Geog0111$_{S}cientific_{C}omputingDocumentation$

Release 1.0

Prof P. Lewis

Jan 04, 2019

Contents:

UCL Geography: Level 7 course, Scientific Computing

CHAPTER 1

Online Notebooks via Binder:

Run the notebooks on Binder server directly by click on different chaper, it may take some time to start but just wait a bit. . . .

Go to full list of notebooks

Go to individual chapter:

Chapter0_help

Chapter1_Python_introduction

Chapter1_Python_introduction_answers

Chapter2_Numpy_matplotlib

Chapter2_Numpy_matplotlib_answers

Chapter3_0_GDAL

Chapter3_1_GDAL

Chapter3_1_GDAL_answers

Chapter3_2_MODIS_download

Chapter3_2_MODIS_download_answers

Chapter3_3_GDAL_masking

Chapter3_4_GDAL_stacking_and_interpolating

Chapter3_4a_GDAL_stacking_and_interpolating-convolution

Chapter3_5_Movies

Chapter3_6A_GDAL_Reconciling_projections_prerequisites

Chapter3_6_GDAL_Reconciling_projections

Chapter4_Practical_Part1

Chapter5_Linear_models

Chapter5_Modelling_and_optimisation

Chapter6_NonLinear_Model_Fitting

Chapter6_NonLinear_Model_Fitting_Solutions

Chapter7_FittingPhenologyModels

Chapter7_FittingPhenologyModels_Solutions

Chapter8_Practical_Part2

Chapter9_Fire_and_Teleconnections

Chapter9_Fire_and_Teleconnections_Solution

Connection

CHAPTER 2

Course information

2.1 Course Convenor

Prof P. Lewis

N.B. 2018-19 Course Convenors: Dr Qingling Wu and Dr. Jose Gomez-Dans

2.2 Course and Contributing Staff

Prof Philip Lewis

Dr. Jose Gomez-Dans

Dr Qingling Wu

Mr Feng Yin

Mr James Brennan

2.3 Purpose of this course

This course, geog0111 Scientific Computing, is a term 1 MSc module worth 15 credits (25% of the term 1 credits) that aims to:

- impart an understanding of scientific computing
- give students a grounding in the basic principles of algorithm development and program construction
- to introduce principles of computer-based image analysis and model development

It is open to students from a number of MSc courses run by the Department of Geography UCL, but the material should be of wider value to others wishing to make use of scientific computing.

The module will cover:

- Computing in Python
- Computing for image analysis
- Computing for environmental modelling
- Data visualisation for scientific applications

2.4 Learning Outcomes

At the end of the module, students should:

- have an understanding of the Python programmibng language and experience of its use
- have an understanding of algorithm development and be able to use widely used scientific computing software to manipulate datasets and accomplish analytical tasks
- have an understanding of the technical issues specific to image-based analysis, model implementation and scientific visualisation

2.5 Timetable

The course takes place over 10 weeks in term 1, in the Geography Department Unix Computing Lab (PB110) in the Pearson Building, UCL.

Classes take place from the second week of term to the final week of term, other than Reading week. See UCL term dates for further information.

The timetable is available on the UCL Academic Calendar

2.6 Assessment

Assessment is through two pieces of coursework, submitted in both paper form and electronically via Moodle.

See the Moodle page for more details.

2.7 Useful links

Course Moodle page

2.8 Python

Python is a high level programming language that is freely available, elatively easy to learn and portable acros

Table of Contents

1.2.3 help

1.3.3 Arrays: np.array

#1. `help`

You can get help on an object using the `help()` method. This will return a full manual page of the class documentation.

```
#the method help()
help(list)
```

```
Help on class list in module builtins:

class list(object)
 |  list() -> new empty list
 |  list(iterable) -> new list initialized from iterable's items
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __iadd__(self, value, /)
 |      Implement self+=value.
 |
 |  __imul__(self, value, /)
 |      Implement self*=value.
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate␣
↪signature.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __reversed__(...)
 |      L.__reversed__() -- return a reverse iterator over the list
 |
 |  __rmul__(self, value, /)
 |      Return value*self.
 |
 |  __setitem__(self, key, value, /)
 |      Set self[key] to value.
 |
 |  __sizeof__(...)
 |      L.__sizeof__() -- size of L in memory, in bytes
 |
 |  append(...)
 |      L.append(object) -> None -- append object to end
 |
 |  clear(...)
 |      L.clear() -> None -- remove all items from L
 |
 |  copy(...)
 |      L.copy() -> list -- a shallow copy of L
 |
 |  count(...)
 |      L.count(value) -> integer -- return number of occurrences of value
 |
 |  extend(...)
 |      L.extend(iterable) -> None -- extend list by appending elements from␣
↪the iterable
 |
 |  index(...)
 |      L.index(value, [start, [stop]]) -> integer -- return first index of␣
↪value.
 |      Raises ValueError if the value is not present.
 |
 |  insert(...)
 |      L.insert(index, object) -- insert object before index
 |
 |  pop(...)
 |      L.pop([index]) -> item -- remove and return item at index (default␣
↪last).
 |      Raises IndexError if list is empty or index is out of range.
 |
 |  remove(...)
 |      L.remove(value) -> None -- remove first occurrence of value.
 |      Raises ValueError if the value is not present.
 |
 |  reverse(...)
 |      L.reverse() -- reverse *IN PLACE
 |
 |  sort(...)
 |      L.sort(key=None, reverse=False) -> None -- stable sort IN PLACE
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __hash__ = None
```

You can get a shorter set of basic help by putting ? after the object.

In a notebook, this will show in a new window at the bottom of the book. You can get rid of this by clicking the x.

```
list?
```

Another useful thing is to see a list of potential methods in a class. This is achieved by hitting the <tab> key, e.g.

```
# place the cursor after the `.` below
# hit the <tab> key, rather than <return> in this cell
# Dont run this cell
list.
```

Really, this is just using the fact that <tab> key performs variable name completion.

This means that if you e.g. have variables called the_long_one and the_long_two set:

```
the_long_one = 1
the_long_two = 2
```

The next time you want to refer to this string in code, you need only type as many letters needed to distinguish this from other variable names, then hit <tab> to complete the name as far as possible.

E1.3.5 Exercise

- in the cell below, place the cursor after the letter t and hit <tab>. It should show you a list of things that begin with t.
- Use this to write the line of code the_long_one = 1000
- in the cell below, place the cursor after the letters th and hit <tab>. It should show you a list of things that begin with th. In this case it should just give you the options of the_long_one or the_long_two.
- If you hit <tab> again, the variable name will be completed as far as it can, here, up to the_long_. Use this to write the line of code the_long_two = 2000

```
# do exercise here ... put the cursor after the t or th and
# use <tab> for completion. Dont run this cell
```

(continues on next page)

(continued from previous page)

```
t
th
```

2.8.1 1. Introduction to Python With ANSWERS

Table of Contents

Python is a popular general purpose, free language. As opposed to other systems that are focused towards a particular task (e.g. R for statistics), Python has a strong following on the web, on systems operations and in data analysis. For scientific computing, a large number of useful add-ons ("libraries") are available to help you analyse and process data. This is an invaluable resource.

In addition to being free, Python is also very portable, and easy to pick up.

The aim of this Chapter is to introduce you to some of the fundamental concepts in Python. Mainly, this is based around fundamental data types in Python (`int`, `float`, `str`, `bool` etc.) and ways to group them (`tuple`, `list`, array, string and `dict`).

Although some of the examples we use are very simple to explain a concept, the more developed ones should be directly applicable to the sort of programming you are likely to need to do.

Further, a more advanced section of the chapter is available, that goes into some more detail and complkications. This too has a set of exercises with worked examples.

In this session, you will be introduced to some of the basic concepts in Python.

The session should last 4 hours (one week).

1.1 Getting Started

1.1.1 Comments and print function

Comments are statements ignored by the language interpreter.

Any text after a # in a *code block* is a comment.

You can 'run' the code in a code block using the 'run' widget (above) or hitting the keys ('typing') and at the same time.

E1.1.1 Exercise

- Try running the code block below
- Explain what happened ('what the computer did')

```
# Hello world
```

ANSWER

Nothing 'apparently' happened, but really, the code block was interpreted as a set of Python commands and executed. As there is only a comment, there was no output.

You can also use text blocks (contained in quotes) to contain comments, but note that if it is the last statement in the code, the text block may be printed out to the terminal.

```
# single line text
'hello world is not printed'

# multi-line text
'''
Hello
world
is printed
'''
```

```
'nHello nworldnis printedn'
```

E1.1.2 Exercise

- Copy the text from above in the window below
- Then put a new text block at the end and note down what happens
- What does the \n mean/do?

```
# do the exercise here
# ANSWER

# single line text
```

(continues on next page)

(continued from previous page)

```
'hello world is not printed'

# multi-line text
'''
Hello
world
is printed
'''

'''
Another text block
'''
```

```
'nAnother text blockn'
```

ANSWER

The last output is printed. In this case, it is the string `'\nAnother text block\n'`.

The `\n` is a newline character. It setrs the text cursor to the start of a new line.

To print some statement (by default, to the screen you are using), use the `print` method:

```
print('hello world')
```

```
hello world
```

```
print('hello','world')
```

```
hello world
```

In Python 3.X, `print` is a function with the *argument(s)* (here, the string you want printed) enclosed in the function's (round) brackets.

E1.1.3 Exercise

- Copy the print statement from above code block.
- Change the words in the quotes and print them out.
- Add some comments to the code block explaining what you have done and seen.

```
# do the exercise here
# ANSWER


'''
The print statement below sends text to the output channel (the 'screen' as it were).
In this case, it prints the string 'good', followed by a space, then the string
↪'morning'
'''
print('good','morning')
```

```
good morning
```

1.1.2 Variables, Values and Data types

The idea of **variables** is fundamental to any programming. You can think of this as the *name* of *something*, so it is a way of allowing us to refer to some object in the language.

What the variable *is* set to is called its **value**.

So let's start with a variable we will call (*declare to be*) `x`.

We will give a *value* of the string `'one'` to this variable:

```
x = 'one'

print(x)
```

```
one
```

E1.1.4 Exercise

- set the a variable called x to some different string (e.g. 'hello world')
- print the value of the variable `x`
- Try this again, putting some 'newlines' (`\n`) in the string

```
# do the exercise here
# ANSWER

x = 'hello world'

print(x)
'''
Note the space before the final
hello world, as there is a comma
between '\n' and x
'''
print(x,'\n',x)
```

```
hello world
hello world
 hello world
```

```
# Now we set x to the value 1

x = 1

print(x,'is type',type(x))
```

```
1 is type <class 'int'>
```

In a computing language, the *sort of thing* the variable can be set to is called its **data type**.

In python, we can access this with the method `type()` as in the example above.

In the example above, the datatype is an **integer** number (e.g. `1, 2, 3, 4`).

In 'natural language', we might read the example above as 'x is one'.

E1.1.5 Exercise

- set the a variable called `x` to the integer `5`
- print the value and type of the variable `x`
- change the data type used for `x` to something else (e.g. a string)

```
# do the exercise here
# ANSWER

x = 5
'''
We use the methods type() and str() here
'''
print(x,type(x),str(x))
```

```
5 <class 'int'> 5
```

Setting `x = 1` is different to:

```
x = 'one'
```

because here we have set value of the variable `x` to a **string** (i.e. some text).

A string is enclosed in quotes, e.g. `"one"` or `'one'`, or even `"'one'"` or `'"one"'`.

```
print ("one")
print ('one')
print ("'one'")
print ('"one"')
```

```
one
one
'one'
"one"
```

E1.1.6 Exercise

- create a variable `name` containing your name, as a string.
- using this variable and the print function, print out a statement such as `my name is Fred` (if your name were `Fred`)

```
# do the exercise here
# ANSWER

name = 'Professor Philip Lewis'
print('My name is',name)
```

```
My name is Professor Philip Lewis
```

Setting `x = 1` or `x = 'one'` is different to:

```
x = 1.0
```

because here we have set value of the variable `x` to a **floating point** number (these are treated and stored differently to integers in computing).

This in turn is different to:

```
x = True
```

where `True` is a **logical** or **boolean** datatype (something is `True` or `False`).

E1.1.7 Exercise

- in the code block below, create a variable called `my_var` and set it to some value (your choice of value, but be clear about the data type you intend)
- print the value of the variable to the screen, along with the data type.

```
# do the exercise here
# ANSWER

# set to integer
my_var = 100
print(my_var,type(my_var))
```

```
100 <class 'int'>
```

We have so far seen four datatypes:

- integer (`int`): 32 bits long on most machines
- (double-precision) floating point (`float`): (64 bits long)
- Boolean (`bool`)
- string (`str`)

but we will come across more (and even create our own!) as we go through the course.

In each of these cases above, we have used the variable `x` to contain these different data types.

As we saw above, if you want to know what the data type of a variable is, you can use the method `type()`

```
print (type(1));
print (type(1.0));
print (type('one'));
print (type(True));
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
```

You can explicitly convert between data types, e.g.:

```
print ('int(1.1) = ',int(1.1))
print ('float(1) = ',float(1))
print ('str(1) = ',str(1))
print ('bool(1) = ',bool(1))
```

```
int(1.1) =  1
float(1) =  1.0
str(1) =  1
bool(1) =  True
```

but only when it makes sense:

```
print ("converting the string '1' to an integer makes sense:",int('1'))
```

```
converting the string '1' to an integer makes sense: 1
```

```
print ("converting the string 'one' to an integer doesn't:",int('one'))
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-23-11bdc0c0e878> in <module>
----> 1 print ("converting the string 'one' to an integer doesn't:",int('one'))


ValueError: invalid literal for int() with base 10: 'one'
```

When you get an error (such as above), you will need to learn to *read* the error message to work out what you did wrong.

E1.1.8 Exercise

- why did the statement above not work?
- type some other data conversions below that *do* work.

```
# do the exercise here
# ANSWER

print(str(1),str('1 2 3'))
print(int('1'))
```

```
1 1 2 3
1
```

We tried to convert a string ('one') to an integer, that doesn't make sense. It *is* ok e.g. for `int('1')` though.

1.1.3 Arithmetic

Often we will want to do some arithmetic with numbers in a program, and we use the 'normal' (derived from C) operators for this.

Note the way this works for integers and floating point representations.

```
'''
    Some examples of arithmetic operations in Python

    Note how, if we mix float and int, the result is raised to float
    (as the more general form)
'''

print (10 + 100)      # int addition
print (10. - 100)     # float subtraction
print (1./2.)         # float division
print (1/2)           # int division
print (10.*20.)       # float multiplication
```

(continues on next page)

(continued from previous page)

```
print (2 ** 3.)       # float exponent

print (65%2)          # int remainder
print (65//2)         # floor operation
```

```
110
-90.0
0.5
0.5
200.0
8.0
1
32
```

E1.1.9 Exercise

- change the numbers in the examples above to make sure you understand these basic operations.
- try combining operations and use brackets `()` to check that that works as expected.
- see what happens when you add (i.e. use `+`) strings together

```
# do the exercise here
# ANSWER

print (1 + 10)      # int addition
print (1. - 10)     # float subtraction
print (3./2.)         # float division
print (3/2)           # int division
print (1.*2.)       # float multiplication
print (2 ** 8.)       # float exponent

print (129%2)          # int remainder
print (129//2)         # floor operation
```

```
11
-9.0
1.5
1.5
2.0
256.0
1
64
```

Most of these are obvious. The integer division in this case returns a float, which is a change from Pythjon 2.

The remainder and floor terms mean that `129 = 64 * 2 + 1`, i.e. `129 = floor * 2 + remainder`

1.1.4 Assignment Operators

```
'''
    Assignment operators

    x = 3   assigns the value 3 to the variable x
    x += 2  adds 2 onto the value of x
```

(continues on next page)

(continued from previous page)

```
            so is the same as x = x + 2
            similarly /=, *=, -=
    x %= 2  is the same as x = x % 2
    x **= 2 is the same as x = x ** 2
    x //= 2 is the same as x = x // 2

    A 'magic' trick
    ===============

    based on
    https://www.wikihow.com/Read-Someone%27s-Mind-With-Math-(Math-Trick)

    whatever you put as myNumber, the answer is 42

    Try this with integers or floating point numbers ...
'''

# pick a number
myNumber = 34.67

x = myNumber

x *= 2

x *= 5

x /= myNumber

x -= 7

x += 39

# The answer will always be 42
print(x)
```

```
42.0
```

E1.1.10 Exercise

- change the number assigned to `myNumber` and check if `42` is still returned
- copy and edit the code to print the value of `x` each time you change it, and add comments explaining what is happening for each line of code. This should allow you to follow more carefully what has happened with the arithmetic and also to simplify the code (use fewer statements to achieve the same thing).

```
# do the exercise here
# ANSWER

myNumber = 27.6
x = myNumber
print(x)
x *= 2
print(x)
x *= 5
print(x)
x /= myNumber
print(x)
```

(continues on next page)

(continued from previous page)

```
x -= 7
print(x)
x += 39
# The answer will always be 42
print(x)
```

```
27.6
55.2
276.0
10.0
3.0
42.0
```

```
# do the exercise here
# ANSWER ... not so magic after all

myNumber = 27.6

x = 10
print(x)
x += 32
# The answer will always be 42
print(x)
```

```
10
42
```

1.1.5 Logical Operators

Logical operators combine boolean variables. Recall from above:

```
print (type(True),type(False));
```

```
<class 'bool'> <class 'bool'>
```

The three main logical operators you will use are:

```
not, and, or
```

The impact of the `not` opeartor should be straightforward to understand, though we can first write it in a 'truth table':

A	not A
T	F
F	T

```
print('not True is',not True)
print('not False is',not False)
```

```
not True is False
not False is True
```

E1.1.11 Exercise:

- write a statement to set a variable `x` to `True` and print the value of `x` and `not x`
- what does `not not x` give? Make sure you understand why

```
# do the exercise here
# ANSWER

x = True
print(x)
print(not x)
print(not not x)
```

```
True
False
True
```

not not cancels out.

The operators `and` and `or` should also be quite straightforward to understand: they have the same meaning as in normal english. Note that `or` is 'inclusive' (so, read `A or B` as 'either A or B or both of them').

```
print ('True and True is',True and True)
print ('True and False is',True and False)
print ('False and True is',False and True)
print ('False and False is',False and False)
```

```
True and True is True
True and False is False
False and True is False
False and False is False
```

So, `A and B` is `True`, if and only if both `A` is `True` and `B` is `True`. Otherwise, it is `False`

We can represent this in a 'truth table':

A	B	A and B
T	T	T
T	F	F
F	T	F
F	F	F

E1.1.12 Exercise:

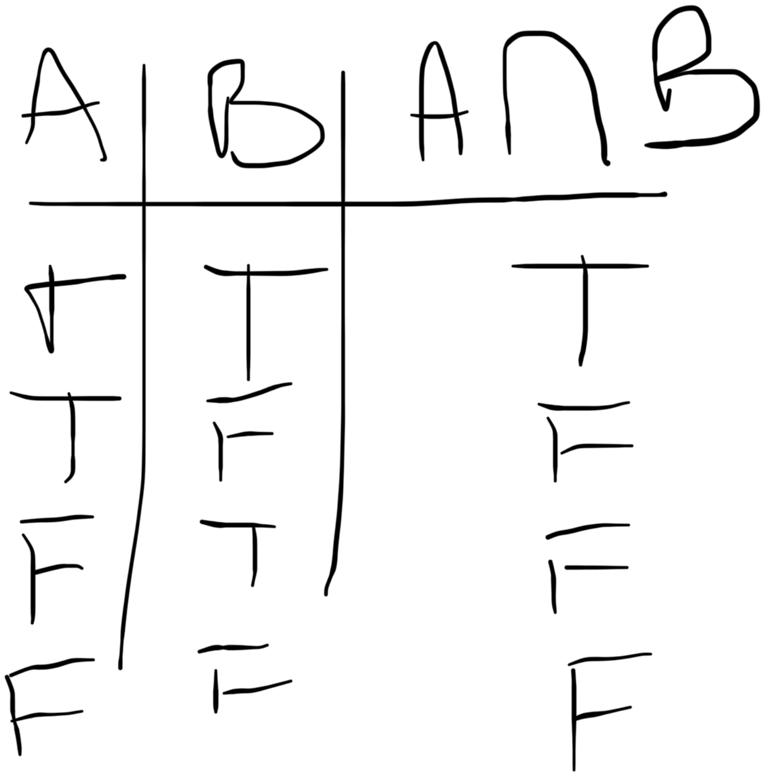
- draw a truth table *on some paper*, label the columns `A`, `B` and `A and B` and fill in the columns `A` and `B` as above
- without looking at the example above, write the value of `A and B` in the third column.
- draw another truth table *on some paper*, label the columns `A`, `B` and `A and B` and fill in the columns `A` and `B` as above
- write the value of `A or B` in the third column.

If you are unsure, test the response using code, below.

```
# do the testing here e.g.
print (True or False)
```

```
True
```

ANSWER



A	B	$A \cap B$
T	T	T
T	F	F
F	T	F
F	F	F

A	B	A∪B
T	T	T
T	F	T
F	T	T
F	F	F

E1.1.13 Exercise

- Copy the following truth table onto paper and fill in the final column:

A	B	C	((A and B) or C)
T	T	T	
T	T	F	
T	F	T	
T	F	F	
F	T	T	
F	T	F	
F	F	T	
F	F	F	

- Try some other compound statements

If you are unsure, or to check your answers, test the response using code, below.

```
# do the testing here e.g.
print ((True and False) or True)
```

```
True
```

ANSWER

Do it one statement at a time:

A	B	C	A and B	(A and B) or C
T	T	T	T	T
T	T	F	T	T
T	F	T	F	T
T	F	F	F	F
F	T	T	F	T
F	T	F	F	F
F	F	T	F	T
F	F	F	F	F

1.1.6 Comparison Operators and `if`

A comparison operator 'compares' two terms (e.g. variables) and returns a boolean data type (`True` or `False`).

For example, to see if the value of some variable `a` is 'the same value as' ('equivalent to') the value of some variable `b`, we use the equivalence operator (==). To test for non equivalence, we use the not equivalent operator `!=` (read the `!` as 'not'):

```
a = 100
b = 10

# Note the use of \n and \t in here
#
print ('a is',a,'and\nb is',b,'\n')
print ('\ta is equivalent to b?',a == b)
```

```
a is 100 and
b is 10

    a is equivalent to b? False
```

E1.1.14 Exercise

- copy the code above and change the values (or type) of the variables `a` and `b` to test their equivalence.
- what does the `\t` in the print statement do?
- add a `print` statement to your code that tests for 'non equivalence'
- write some code to see if `(a or b)` is equivalent to `(b or a)` or not

```
# do the exercise here
# ANSWER

# for logical operators
a = True
b = False

# Note the use of \n and \t in here
#
print ('a is',a,'and\nb is',b,'\n')
print ('\ta is equivalent to b?',a == b)
print ('\ta is not equivalent to b?',a != b)
print ('\t(a or b) == (b or a)?',(a or b) == (b or a))

# for other comparisons ... the or works differently
a = 10
b = 20

# Note the use of \n and \t in here
#
print ('a is',a,'and\nb is',b,'\n')
print ('\ta is equivalent to b?',a == b)
print ('\ta is not equivalent to b?',a != b)
print ('\t(a or b) == (b or a)?',(a or b) == (b or a))

# this just sets to a and b respectively, since True
print((a or b),(b or a))
```

```
a is True and
b is False

    a is equivalent to b? False
    a is not equivalent to b? True
    (a or b) == (b or a)? True
a is 10 and
b is 20

    a is equivalent to b? False
    a is not equivalent to b? True
    (a or b) == (b or a)? False
10 20
```

A full set of comparison operators is:

2.8.2 symbol meaning

== is equivalent to != is not equivalent to > greater than >= greater than or equal to < less than <= less than or equal to ====== ==

so that, for example:

```
# Comparison examples

# is one plus one list identical to two list?
print ([1 + 1] is [2])

# is one plus one list equal to two list?
print ([1 + 1] == [2])

# is one less than or equal to 0.999?
print (1 <= 0.999)

# is one plus one not equal to two?
print (1 + 1 != 2)

# note the use of single quotes inside a double quoted string here
# is 'more' greater than 'less'?
print ("more" > "less")

# "is 100 less than 2?"
print (100 < 2)
```

```
False
True
False
False
True
False
```

Aside on string comparisons

In the case of string comparisons, the ASCII codes of the string characters are compared. So for example the statement "more" > "less" returns True.

Here, the comparison is effectively

```
m > l
```

Since `m` comes after `l` in the alphabet, the ASCII code for `m` (109) is greater than the ASCII code for `l` (108) (see http://www.asciitable.com) so

```
109 > 108
```

returns True. Note that ASCII capital letters come before the lower case letters.

In practice, we mainly avoid string comparisons (other than to confirm equivalence). So there is little direct use of string comparisons other than `==`. It is useful to know how this works however, in case it crops up or happens 'by accident'. It is also worth understanding what ASCII codes are.

Conditional test

One common use of comparisons is for program control, using an `if` statement:

```
if condition1 is True:
    doit1()
elif condition2 is True:
    doit2()
else:
    doit3()
```

where `is` compares identity. This allows us to run blocks of code (e.g. the method `doit1()`) only under a particular condition (or set of conditions).

In Python, the statement(s) we run on condition (here `doit1()` etc.) are *indented*.

The indent can be one or more spaces or a `<tab>` character, the choice is up to the programmer. However, it **must be consistent**.

```
test = [1+1]
print('test is {}'.format(test))

# initialise retval
retval = None

# conduct some tests, and set the
# variable retval to True if we pass
# any test

if test is [2]:
    retval = True
    print('passed test 1: "if test is [2]"')
elif test == [2]:
    retval = True
    print('passed test 2: "if test == [2]"')
else:
    retval = False
    print('failed both tests')

print('retval is',retval)
```

```
test is [2]
passed test 2: "if test == [2]"
retval is True
```

E1.1.15 Exercise

- copy the example above, and change it to use other examples from the 'Comparison examples' code block. Change the value of `test` to get different responses and make notes as to why you get the result you do.
- try out some more complicated conditions, e.g. multipler tests, combined with an `and` operator.

```
# do the exercise here
# ANSWER

test = [3 / 2]
print('test is {}'.format(test))

# initialise retval
retval = None

# conduct some tests, and set the
```

(continues on next page)

(continued from previous page)

```
# variable retval to True if we pass
# any test

if test is [1.5]:
    retval = True
    print('passed test 1: "if test is [1.5]"')
elif test == [1]:
    retval = True
    print('passed test 2: "if test == [1]"')
else:
    retval = False
    print('failed both tests')

print('retval is',retval)
```

```
test is [1.5]
failed both tests
retval is False
```

So be careful how you use `is`. It is not the same as == which only tests for equivalence. `is` is really a pointer comparison, so are they the same object?

In this section, you have had an introduction to the Python programming language, running in a `` `jupyter notebook `` <http://jupyter.org>'__ environment.

You have seen how to write comments in code, how to form `print` statements and basic concepts of variables, values, and data types. You have seen how to maniputae data with arithmetic and assignment operators, as well as the basics in dealing with logic and tests returning logical values.

1.2 Text and looping

In Python, collections of characters (`a`, `b`, `1`, `...`) are called strings. Strings and characters are input by surrounding the relevant text in either double (`"`) or single (`'`) quotes. There are a number of special characters that can be encoded in a string provided they're "escaped". For example, some we have come across are:

- `\n`: the carriage return
- `\t`: a tabulator

```
print ("I'm a happy string")
print ('I\'m a happy string') # the apostrophe has been escaped as not to be confused␣
↪by end of string
print ("\tI'm a happy string")
print ("I'm\na\nhappy\nstring")
```

```
I'm a happy string
I'm a happy string
    I'm a happy string
I'm
a
happy
string
```

We can do a number of things with strings, which are very useful. These so-called string methods are defined on all strings by Python by default, and can be used with every string. For one, we can concatenate strings using the + symbol as we saw above.

1.2.1 `len`

Gives the length of the string as number of characters:

```
t = ''
print ('the length of',t,'is',len(t))


s = "Hello" + "there" + "everyone"

print ('the length of',s,'is',len(s))
```

```
the length of  is 0
the length of Hellothereeveryone is 18
```

Exercise E1.2.1

- what does a zero-length string look like?
- The `Hello there everyone` example above has no spaces between the words. Copy the code to the block below and modify it to have spaces.
- confirm that you get the expected increase in length.

```
# do exercise here
# ANSWER

zero = ''
s = "Hello" + " " + "there" + " " +  "everyone"
print(s)
print ('the length of',s,'is',len(s))
```

```
Hello there everyone
the length of Hello there everyone is 20
```

1.2.2 `for ... in ...` and `enumerate`

Very commonly, we need to iterate or 'loop' over some set of items.

The basic stucture for doing this (in Python, and many other languages) is `for item in group:`, where `item` is the name of some variable and `group` is a set of values.

The loop is run so that `item` takes on the first value in `group`, then the second, etc.

```
# for loop
group = [4,3,2,1]

for item in group:
    '''print counter in loop'''
    print(item)

print ('blast off!')
```

```
4
3
2
```

(continues on next page)

(continued from previous page)

```
1
blast off!
```

The `group` in this example is the list of integer numbers `[4,3,2,1]`. A `list` is a group of comma-separated items contained in square brackets `[]`.

In Python, the statement(s) we run whilst looping (here `print(item)`) are *indented*.

The indent can be one or more spaces or a `<tab>` character, the choice is up to the programmer. However, it **must be consistent**.

Exercise 1.2.1

- generate a list of strings called `group` with the names of (some of) the items in your pocket or bag (or make some up!)
- set up a `for` loop to go through and print each item

```
# do exercise here
# ANSWER

group = ['cat','fish','egg']
for item in group:
    print('I have',item,'in my pocket')
```

```
I have cat in my pocket
I have fish in my pocket
I have egg in my pocket
```

Quite often, we want to keep track of the 'index' of the item in the loop (the 'item number').

One way to do this would be to use a variable (called `count` here).

Before we enter the loop, we initialise the value to zero.

```
# for loop
group = ['hat','dog','keys']

# initialise a variable count
count = 0

for item in group:
    '''print counter in loop'''

    print('item',count,'is',item)

    # add 1 onto count
    count += 1
```

```
item 0 is hat
item 1 is dog
item 2 is keys
```

Exercise 1.2.2

- copy the code above, and check to see if the value of `count` at the end of the loop is the same as the length of the list. Why should this be so?
- change the code so that the counting starts at 1, rather than 0.

```
# do exercise here
# ANSWER

# for loop
group = ['hat','dog','keys']

# initialise a variable count
count = 0

for item in group:
    '''print counter in loop'''

    print('item',count,'is',item)

    # add 1 onto count
    count += 1
'''
Expect the same as we have incremented
count once for each item in group, so it
should be the same as the length of the group!
'''
print(count,len(group))
```

```
item 0 is hat
item 1 is dog
item 2 is keys
3 3
```

Since counting in loops is a common task, we can use the built in method `enumerate()` to achieve the same thing as above. The syntax is then:

```
# for loop
group = ['hat','dog','keys']

for count,item in enumerate(group):
    '''print counter in loop'''
    print('item',count,'is',item)
```

```
item 0 is hat
item 1 is dog
item 2 is keys
```

Exercise 1.2.3

- copy the code above, and check to see if the value of `count` at the end of the loop is the same as the length of the list.
- change the code so that the printed count starts at 1, rather than 0.

Hint: how can you make it print `count+1` rather than `count`?

```
# do exercise here
# ANSWER

# for loop
group = ['hat','dog','keys']

```

(continues on next page)

(continued from previous page)

```
for count,item in enumerate(group):
    '''print counter in loop'''
    print('item',count,'is',item)

'''
In this case, count starts at 0 and takes the
values 0, 1 and 2 as we loop over the 3 items.
'''
print(count,len(group))
'''
print count + 1 !!
'''
print('item',count+1,'is',item)
```

```
item 0 is hat
item 1 is dog
item 2 is keys
2 3
item 3 is keys
```

1.2.3 `slice`

A string can be thought of as an ordered ‘array’ of characters.

So, for example the string `hello` can be thought of as a construct containing `h` then `e`, `l`, `l`, and `o`.

We can index a string, so that e.g. `'hello'[0]` is `h`, `'hello'[1]` is `e` etc.

We have seen above the idea of the ‘length’ of a string. In this example, the length of the string `hello` is 5.

```
string = 'hello'

# length
slen = len(string)
print('length of {} is {}'.format(string,slen))

# select these indices
indices = 0,1,3

# loop over each item in indices
for index in indices:
    print('character {} of {} is {}'.format(index,string,string[index]))
```

```
length of hello is 5
character 0 of hello is h
character 1 of hello is e
character 3 of hello is l
```

Exercise E1.2.4

- copy the code above, and see what happens if you set a value in `indices` that is the value of length of the string. Why does it respond so?
- make the code robust to this issue, but using an `if` statement to test if `index` is in the required range.

```
# do exercise here
# ANSWER

string = 'hello'

# length
slen = len(string)
print('length of {} is {}'.format(string,slen))

# select these indices
indices = 0,1,3,6

# loop over each item in indices
'''
WE expect a failure as we have
asked for an out of range index
'''

for index in indices:
    print('character {} of {} is {}'.format(index,string,string[index]))
```

```
length of hello is 5
character 0 of hello is h
character 1 of hello is e
character 3 of hello is l
```

```
---------------------------------------------------------------------------

IndexError                                Traceback (most recent call last)

<ipython-input-68-88baf263801d> in <module>
     18
     19 for index in indices:
---> 20     print('character {} of {} is {}'.format(index,string,string[index]))
     21


IndexError: string index out of range
```

```
# do exercise here
# ANSWER

string = 'hello'

# length
slen = len(string)
print('length of {} is {}'.format(string,slen))

# select these indices
indices = 0,1,3,6

# loop over each item in indices
'''
Fix by using condition
'''
```

(continues on next page)

(continued from previous page)

```
for index in indices:
    if index < slen:
        print('character {} of {} is {}'.format(index,string,string[index]))
    else:
        print('index {} beyond the string length {}'.format(index,slen))
```

```
length of hello is 5
character 0 of hello is h
character 1 of hello is e
character 3 of hello is l
index 6 beyond the string length 5
```

We can use the idea of a 'slice' to access particular elements within the string.

For a slice, we can specify:

- start index (0 is the first)
- stop index (not including this)
- skip (do every 'skip' character)

When specifying this as array access, this is given as, e.g.:

`array[start:stop:skip]`

- The default start is 0
- The default stop is the length of the array
- The default skip is 1

You can specify a slice with the default values by leaving the terms out:

`array[::2]`

would give values in the array `array` from 0 to the end, in steps of 2.

This idea is fundamental to array processing in Python. We will see later that the same mechanism applies to all ordered groups.

```
s = "Hello World"
print (s,len(s))

start = 0
stop  = 11
skip  = 2
print (s[start:stop:skip])

# use -ve numbers to specify from the end
# use None to take the default value

start = -3
stop  = None
skip  = 1
print (s[start:stop:skip])
```

```
Hello World 11
HloWrd
rld
```

Exercise E1.2.5

The example above allows us to access an individual character(s) of the array.

- copy the example above, and print the string starting from the default start value, up to the default stop value, in steps of `2`.
- write code to print out the 4^{th} letter (character) of the string `s`.

```
# do exercise here
# ANSWER

s = "Hello World"

# take the default start by setting to None
start = None
stop  = None
skip  = 2
print (s[start:stop:skip])

# 4th char ... start at 0 remember!
print(s[3])
```

```
HloWrd
l
```

1.2.4 `replace`

We can replace all occurrences of a string within a string by some other string. We can also replace a string by an empty string, thus in effect removing it:

```
print ("I'm a very happy string".replace("happy", "unhappy"))
```

```
I'm a very unhappy string
```

Exercise E1.2.6

- copy the statement above, and use the `replace` method to make it print out `"I'm a happy string"`.

Hint: you want to replace the string `very` with, effectively, nothing, i.e. a zero-length string.

```
# do exercise here
# ANSWER

# note that we replace 'very '
print ("I'm a very happy string".replace("very ", ""))
```

```
I'm a happy string
```

1.2.5 `find`

Quite often, we might want to find a string inside another string, and potentially give the location (as in characters from the start of the string) where this string occurs. We can use the `find` method, which will return either a `-1` if the string isn't found, or an integer giving the index of where the string starts (for the first time).

```
print ("I'm a very happy string".find("a"))
print ("I'm a very happy string".find("happy"))
```

```
4
11
```

Let's use the idea of `find()` to sort out a messy table of data that we get from a web page.

First, we need to import the package `requests` to access some information from a URL (from a web page). The data we get will be in html.

The data we will examine is a dataset of ENSO values for each month of the year from January 1950 to present, made available by NOAA/

If you visit you will see the data table we are interested in. So, how do we 'grab' this?

The URL points to html code. When you display this in a browser, it is rendered appropriately.

If you access the html directly, you will get the following:

```
# Web scraping example

import requests

url = "http://www.esrl.noaa.gov/psd/enso/mei/table.html"

# This line will pull the URL data as a string
txt = requests.get(url).text

# show the first 1000 characters (see 'slice' above: this is the same as␣
↪[None:1000:None])
print(txt[:1000])
```

```
<html>
<head><title>MEI timeseries from Dec/Jan 1940/50 up to the present</title></head>
<body>
<pre>
MEI Index (updated: 13 October 2018)

Bimonthly MEI values (in 1/1000 of standard deviations), starting with Dec1949/
↪Jan1950, thru last
month.  More information on the MEI can be found on the <a href="mei.html">MEI␣
↪homepage</a>.
Missing values are left blank.  Note that values can still change with each monthly␣
↪update, even
though such changes are typically smaller than +/-0.1.  All values are normalized for␣
↪each bimonthly
season so that the 44 values from 1950 to 1993 have an average of zero and a standard␣
↪deviation of "1".
Responses to 'FAQs' can be found below this table:

YEAR    DECJAN  JANFEB  FEBMAR  MARAPR  APRMAY  MAYJUN  JUNJUL  JULAUG  AUGSEP ␣
↪SEPOCT  OCTNOV  NOVDEC
1950        -1.03   -1.133  -1.283  -1.071  -1.434  -1.412  -1.269  -1.042  -.631   -.
↪406   -1.138  -1.235
1951        -1.049  -1.152  -1.178  -.511   -.374   .288    .679    .818    .726    .
↪768    .726    .504
1952        .433    .138    .071    .224    -.307   -.756   -.305   -.374   .
```

We notice the presence of html codes in the text string (e.g. `<html>`, `<pre>`). There are particular packages for neatly parsing html (scraping information from web pages), one of the most common being BeautifulSoup. This will tend to be more useful if the html is well fomatted, and the data contained in `<table>` sections, or similar structures. Here, we just have a block of text in the `<pre>` section.

If we want to *just* access the dataset here then, we might notice that the data we want to access starts when we see the string `YEAR`.

We can use `find()` to discover the index of this in the string:

```
start = txt.find('YEAR')

print('start of useful data at index {}\n--------------------------------'.
↪format(start))
print(txt[start:start+1000])
```

```
start of useful data at index 688
--------------------------------
YEAR    DECJAN  JANFEB  FEBMAR  MARAPR  APRMAY  MAYJUN  JUNJUL  JULAUG  AUGSEP ␣
↪SEPOCT  OCTNOV  NOVDEC
1950        -1.03   -1.133  -1.283  -1.071  -1.434  -1.412  -1.269  -1.042  -.631   -.
↪406   -1.138  -1.235
1951        -1.049  -1.152  -1.178  -.511   -.374   .288    .679    .818    .726    .
↪768    .726    .504
1952        .433    .138    .071    .224    -.307   -.756   -.305   -.374   .31     .
↪306    -.328   -.098
1953        .044    .401    .277    .687    .756    .191    .382    .209    .483    .
↪124    .099    .351
1954        -.036   -.027   .154    -.616   -1.465  -1.558  -1.355  -1.456  -1.159  -
↪1.32   -1.113  -1.088
1955        -.74    -.669   -1.117  -1.621  -1.653  -2.247  -1.976  -2.05   -1.829  -
↪1.725  -1.813  -1.846
1956        -1.408  -1.275  -1.371  -1.216  -1.304  -1.523  -1.244  -1.118  -1.35   -
↪1.461  -1.014  -.993
1957        -.915   -.348   .108    .383    .813    .73     .926    1.132   1.117   1.
↪114   1.167   1.268
1958        1.473   1.454   1.313   .991    .673    .812    .7      .421    .171    .
↪237    .501    .691
1959        .553    .81     .502    .202    -.025   -.062   -.112   .111    .092    -.
↪038   -.151   -.247
1960        -.287   -.253   -.082   .007    -.322   -.287   -.318   -.25    -.47    -.
↪332   -.308   -.39
1961        -.15    -.235   -.073   .017    -.302   -.185   -.208   -.3     -.3     -.
↪51    -.416   -.60
```

If we look again at the web page http://www.esrl.noaa.gov/psd/enso/mei/table.html, we might notice that the end of the useful data is delimited by two newlines and the string `(1)`, i.e., as a string `\n\n(1)`. So we should be able to use `find()` again to get the location of the end of the data (i.e. `stop`, in the sense of a slice).

Exercise 1.2.6

- use this observation to form a string called `data_table`, containing all of the useful data (i.e. `txt[start:stop]`).
- print the string `data_table`.

```
# do exercise here
# ANSWER
```

(continues on next page)

(continued from previous page)

```
start = txt.find('YEAR')
end = txt.find('\n\n(1)')

print('start of useful data at index {}\n---------------------------------'.
↪format(start))
data_table = txt[start:end]

print(data_table)
```

```
start of useful data at index 688
---------------------------------
YEAR    DECJAN  JANFEB  FEBMAR  MARAPR  APRMAY  MAYJUN  JUNJUL  JULAUG  AUGSEP ␣
↪SEPOCT  OCTNOV  NOVDEC
1950        -1.03   -1.133  -1.283  -1.071  -1.434  -1.412  -1.269  -1.042  -.631   -.
↪406   -1.138  -1.235
1951        -1.049  -1.152  -1.178  -.511   -.374   .288    .679    .818    .726    .
↪768    .726    .504
1952        .433    .138    .071    .224    -.307   -.756   -.305   -.374   .31     .
↪306   -.328   -.098
1953        .044    .401    .277    .687    .756    .191    .382    .209    .483    .
↪124    .099    .351
1954        -.036   -.027   .154    -.616   -1.465  -1.558  -1.355  -1.456  -1.159  -
↪1.32   -1.113  -1.088
1955        -.74    -.669   -1.117  -1.621  -1.653  -2.247  -1.976  -2.05   -1.829  -
↪1.725  -1.813  -1.846
1956        -1.408  -1.275  -1.371  -1.216  -1.304  -1.523  -1.244  -1.118  -1.35   -
↪1.461  -1.014  -.993
1957        -.915   -.348   .108    .383    .813    .73     .926    1.132   1.117   1.
↪114   1.167   1.268
1958        1.473   1.454   1.313   .991    .673    .812    .7      .421    .171    .
↪237    .501    .691
1959        .553    .81     .502    .202    -.025   -.062   -.112   .111    .092    -.
↪038   -.151   -.247
1960        -.287   -.253   -.082   .007    -.322   -.287   -.318   -.25    -.47    -.
↪332   -.308   -.39
1961        -.15    -.235   -.073   .017    -.302   -.185   -.208   -.3     -.3     -.
↪51    -.416   -.608
1962        -1.065  -.963   -.692   -1.04   -.89    -.87    -.683   -.538   -.56    -.
↪642   -.598   -.482
1963        -.718   -.837   -.675   -.761   -.473   -.144   .404    .59     .734    .
↪848    .866    .765
1964        .878    .481    -.256   -.545   -1.234  -1.15   -1.384  -1.486  -1.309  -
↪1.196  -1.211  -.907
1965        -.536   -.329   -.259   .086    .464    .867    1.367   1.425   1.379   1.
↪25    1.378   1.27
1966        1.307   1.186   .689    .515    -.178   -.193   -.116   .152    -.094   -.
↪018   .022    -.181
1967        -.462   -.898   -1.05   -1.03   -.448   -.236   -.492   -.391   -.627   -.
↪655   -.407   -.357
1968        -.602   -.727   -.635   -.944   -1.093  -.812   -.503   -.104   .213    .
↪46    .607    .367
1969        .67     .849    .458    .622    .674    .801    .49     .212    .162    .
↪537    .683    .417
1970        .38     .432    .228    .014    -.099   -.636   -1.055  -1.007  -1.251  -
↪1.068  -1.063  -1.203
1971        -1.204  -1.507  -1.79   -1.839  -1.429  -1.42   -1.207  -1.213  -1.467  -
↪1.399  -1.301  -.969
```

(continues on next page)

(continued from previous page)

```
1972        -.575   -.398   -.256   -.166   .423    .966    1.826   1.8     1.522   1.
↪667   1.74    1.787
1973        1.723   1.515   .87     .491    -.099   -.758   -1.056  -1.334  -1.734  -
↪1.65   -1.482  -1.826
1974        -1.912  -1.768  -1.743  -1.62   -1.048  -.694   -.75    -.664   -.628   -
↪1.031  -1.23   -.886
1975        -.522   -.576   -.85    -.927   -.838   -1.148  -1.497  -1.712  -1.876  -
↪1.968  -1.748  -1.732
1976        -1.587  -1.366  -1.213  -1.157  -.483   .276    .633    .654    1.018   .
↪978    .511    .565
1977        .529    .29     .145    .552    .31     .414    .888    .684    .783    1.
↪016   .99     .877
1978        .777    .912    .935    .199    -.378   -.605   -.42    -.199   -.391   .
↪003    .203    .406
1979        .608    .379    .002    .301    .374    .429    .396    .615    .759    .
↪694    .759    1.004
1980        .677    .601    .684    .912    .958    .911    .769    .329    .263    .
↪223    .27     .111
1981        -.25    -.14    .455    .669    .189    -.024   -.026   -.09    .174    .
↪132    -.021   -.126
1982        -.258   -.125   .1      .008    .443    .93     1.612   1.778   1.783   2.
↪061   2.441   2.425
1983        2.677   2.931   3.008   2.812   2.498   2.235   1.793   1.159   .459    .
↪056    -.115   -.17
1984        -.314   -.509   .151    .39     .18     -.049   -.054   -.155   -.12    .
↪026    -.332   -.585
1985        -.546   -.576   -.696   -.468   -.706   -.166   -.126   -.365   -.535   -.
↪119   -.042   -.279
1986        -.293   -.183   .028    -.107   .353    .279    .401    .765    1.077   1.
↪002   .89     1.202
1987        1.249   1.218   1.716   1.855   2.107   1.958   1.878   1.973   1.851   1.
↪671   1.286   1.293
1988        1.115   .716    .495    .389    .193    -.582   -1.109  -1.295  -1.526  -
↪1.313  -1.447  -1.311
1989        -1.103  -1.241  -1.035  -.76    -.393   -.242   -.43    -.494   -.315   -.
↪319   -.058   .131
1990        .243    .573    .951    .46     .652    .511    .147    .127    .366    .
↪303    .4      .362
1991        .319    .323    .399    .449    .741    1.051   1.044   1.008   .739    1.
↪031   1.202   1.34
1992        1.747   1.886   1.985   2.247   2.085   1.735   1.045   .559    .488    .
↪663    .595    .664
1993        .692    .99     .987    1.408   1.993   1.616   1.204   1.026   .966    1.
↪09    .848    .595
1994        .352    .193    .159    .465    .58     .803    .904    .762    .893    1.
↪437   1.312   1.251
1995        1.219   .959    .845    .453    .57     .507    .234    -.147   -.445   -.
↪461   -.463   -.537
1996        -.597   -.566   -.236   -.391   -.041   .087    -.173   -.374   -.457   -.
↪338   -.134   -.325
1997        -.48    -.605   -.248   .527    1.132   2.275   2.825   3.002   2.99    2.
↪423   2.551   2.344
1998        2.455   2.777   2.751   2.658   2.206   1.336   .392    -.336   -.636   -.
↪789   -1.069  -.908
1999        -1.039  -1.123  -.95    -.88    -.596   -.339   -.478   -.739   -.967   -.
↪954   -1.031  -1.142
2000        -1.122  -1.189  -1.09   -.397   .251    .001    -.159   -.145   -.238   -.
↪367   -.701   -.55
```

(continues on next page)

(continued from previous page)

```
2001         -.496   -.649   -.548   -.05    .282    .052    .297    .332    -.174   -.
↪255   -.138   .033
2002         .017    -.16    -.118   .401    .886    .932    .712    1.002   .881    1.
↪02    1.101   1.156
2003         1.214   .944    .83     .413    .214    .107    .177    .309    .46     .
↪534    .584    .362
2004         .332    .37     -.036   .358    .558    .315    .571    .617    .558    .
↪524    .818    .684
2005         .325    .816    1.057   .626    .885    .589    .519    .343    .296    -.
↪152   -.374   -.55
2006         -.428   -.414   -.521   -.571   .045    .526    .716    .748    .8      .
↪976    1.297   .965
2007         .985    .537    .125    .026    .348    -.155   -.248   -.442   -1.197  -
↪1.204  -1.149  -1.178
2008         -1.006  -1.371  -1.552  -.858   -.345   .142    .088    -.269   -.58    -.
↪681   -.584   -.646
2009         -.714   -.69    -.705   -.106   .326    .751    1.06    1.05    .707    .
↪924    1.134   1.059
2010         1.066   1.526   1.462   .978    .658    -.228   -1.103  -1.671  -1.879  -
↪1.888  -1.472  -1.558
2011         -1.719  -1.544  -1.554  -1.387  -.199   -.003   -.193   -.517   -.778   -.
↪917   -.933   -.945
2012         -.98    -.675   -.382   .11     .757    .842    1.126   .607    .316    .
↪097    .141    .111
2013         .103    -.068   -.026   .09     .205    -.094   -.314   -.481   -.155   .
↪148    -.042   -.234
2014         -.27    -.259   .018    .295    1.001   1.046   .915    .937    .557    .
↪45     .773    .566
2015         .417    .464    .614    .916    1.583   2.097   1.981   2.334   2.479   2.
↪256   2.3     2.12
2016         2.216   2.17    1.963   2.094   1.752   1.053   .352    .167    -.118   -.
↪363   -.197   -.11
2017         -.052   -.043   -.08    .744    1.445   1.039   .456    .009    -.478   -.
↪551   -.277   -.576
2018         -.623   -.731   -.502   -.432   .465    .469    .076    .132    .509
```

This exercise is a very good example of web scraping. Web scraping is often rather messy (you have to work out some 'key' to reliably delimit the information you want) but can be extreemely valuable for accessing datasets that are not cleanly presented. We have only gonbe part of the way to extracting a useful dataset here, because the dataset we are interested in (the ENSO data) are still represented as a string, whereas we really want them to be a set of floating point numbers. We will deal with this later.

1.2.5 `split` and `splitlines`

The first 'line' of `'should contain the 'header' information, i.e. the title of the data columns (`YEAR, DECJAN`etc.). We want to separate the header from the numbers in the data table, so we want to 'split' the string called`data_table' into a header string and data string.

One approach to this would be split the string into 'lines' of text (rather than one block). Effectively that means splitting into multiple strings whenever we hit a `\n` character. Rather than do that explicitly, we use the `splitlines()` method:

```python
import requests
url = "http://www.esrl.noaa.gov/psd/enso/mei/table.html"
```

(continues on next page)

(continued from previous page)

```
txt = requests.get(url).text

# copy the useful data
start = txt.find('YEAR')
stop  = txt.find('\n\n(1)')
data_table = txt[start:stop]

# split into a list of strings
data_lines = data_table.splitlines()

# tell me something useful
print(type(data_lines),len(data_lines))

# loop over some examples
for i in 0,1,len(data_lines)-1:
    print('line {} {}\n\t{}'.format(i,type(data_lines[i]),data_lines[i]))
```

```
<class 'list'> 70
line 0 <class 'str'>
    YEAR    DECJAN   JANFEB   FEBMAR   MARAPR   APRMAY   MAYJUN   JUNJUL   JULAUG   AUGSEP ␣
↪SEPOCT   OCTNOV   NOVDEC
line 1 <class 'str'>
    1950    -1.03    -1.133   -1.283   -1.071   -1.434   -1.412   -1.269   -1.042   -.631    -.
↪406   -1.138   -1.235
line 69 <class 'str'>
    2018    -.623    -.731    -.502    -.432    .465     .469     .076     .132     .509
```

This splits each 'line' of text into an entry in a `list`, so that the header data is now given in the first entry (`data_lines[0]`) and the lines containinmg data, after that.

From the print out above, we notice that the final 'data line' (index `-1`) is shorter than (has fewer entries than) the other lines. This is because we are only part way through this year!.

In 'real' datasets, we quite often have 'messy' lines of data such as this (or data missing for other reasons). How you want to deal with the 'messy bits' depends on the sort of analysis you want to do.

One option (the simplest) would be to simply remove the last line (ignore this year's data):

```
header = data_lines[0]

# select the data block as being from entry 1 to -1
# so, **not including the last row**
data = data_lines[1:-1]

print('header:',header)

for i in 0,1,len(data)-1:
    print('line {} {}\n\t{}'.format(i,type(data[i]),data[i]))
```

```
header: YEAR    DECJAN   JANFEB   FEBMAR   MARAPR   APRMAY   MAYJUN   JUNJUL   JULAUG ␣
↪AUGSEP   SEPOCT   OCTNOV   NOVDEC
line 0 <class 'str'>
    1950    -1.03    -1.133   -1.283   -1.071   -1.434   -1.412   -1.269   -1.042   -.631    -.
↪406   -1.138   -1.235
line 1 <class 'str'>
    1951    -1.049   -1.152   -1.178   -.511    -.374    .288     .679     .818     .726     .
↪768    .726    .504
```

(continues on next page)

(continued from previous page)

```
line 67 <class 'str'>
    2017     -.052    -.043    -.08     .744     1.445    1.039    .456     .009     -.478    -.
↪551   -.277   -.576
```

Exercise 1.2.7

- copy the code from above and explore the response using line indices `-1` and `-2`.

```
# do exercise here
# ANSWER

header = data_lines[0]

# select the data block as being from entry 1 to -1
# so, **not including the last row**
data = data_lines[1:-1]

print('header:',header)

# THE LAST TWO LINES
for i in -1,-2:
    print('line {} {}\n\t{}'.format(i,type(data[i]),data[i]))
```

```
header: YEAR    DECJAN   JANFEB   FEBMAR   MARAPR   APRMAY   MAYJUN   JUNJUL   JULAUG ␣
↪AUGSEP   SEPOCT   OCTNOV   NOVDEC
line -1 <class 'str'>
    2017     -.052    -.043    -.08     .744     1.445    1.039    .456     .009     -.478    -.
↪551   -.277   -.576
line -2 <class 'str'>
    2016     2.216    2.17     1.963    2.094    1.752    1.053    .352     .167     -.118    -.
↪363   -.197   -.11
```

If we want to manipulate or plot the information contained in this (the numbers), we need to convert each of the string representations to a floating point number, e.g. the number `-1.03` rather than the string `'-1.03'`.

Each entry in the list `data` is a string, as we saw above.

We can split an individual string (such as `data[0]` into a list of strings, using the string method `split()`. By default, this splits on 'white space' (i.e. spaces or tab characters), so, e.g.:

```
line = data[0].split()
print(data[0])
print(line,len(line))
```

```
1950        -1.03    -1.133   -1.283   -1.071   -1.434   -1.412   -1.269   -1.042   -.631    -.
↪406   -1.138   -1.235
['1950', '-1.03', '-1.133', '-1.283', '-1.071', '-1.434', '-1.412', '-1.269', '-1.042
↪', '-.631', '-.406', '-1.138', '-1.235'] 13
```

So, we have split the long string into 13 strings in a list.

We want to generate a new list with 13 corresponding floating point values:

```
# split the line on whitespace
line = data[0].split()

# make a new list of the same length
```

(continues on next page)

(continued from previous page)

```
# by copying the variable line
float_data = line.copy()

for index,line_data in enumerate(line):
    # insert the cast float into the list
    # in the right order (use index)
    float_data[index] = float(line_data)

# this is the string list
print(line)

# this is the float list
print(float_data)
```

```
['1950', '-1.03', '-1.133', '-1.283', '-1.071', '-1.434', '-1.412', '-1.269', '-1.042
↪', '-.631', '-.406', '-1.138', '-1.235']
[1950.0, -1.03, -1.133, -1.283, -1.071, -1.434, -1.412, -1.269, -1.042, -0.631, -0.
↪406, -1.138, -1.235]
```

Exercise 1.2.8

- set a variable to be the string `"2, 3, 5, 7, 11, 13, 17, 19, 23, 29"`
- use the approach above to generate a **list of integers** of the first 10 prime numbers.
- print the list with syntax of the pattern of 'prime number 3 is 7'

Make sure you convert each prime number to an integer, rather than leaving it as a string!

Hint: We can still use the method `split()` to do split the string into a list of strings, but this time the separator is a comma, rather than whitespace.

```
# do exercise here
# ANSWER

'''
We recognise these numbers as the first 10 prime numbers!!
'''
pstring = "2, 3, 5, 7, 11, 13, 17, 19, 23, 29"

d = [int(p) for p in pstring.split(',')]
print(d)

# or

d = []
for p in pstring.split(','):
    d.append(int(p))
print(d)

# or, perhaps the neatest, when we know some numpy (later!)
import numpy as np

d = np.array(pstring.split(',')).astype(int)
print(list(d))
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Normally, we wouldn't go to the trouble of first copying the list.

Instead, **where the contents of the loop are simple** (e.g. a single statement) we would use a different way of using a `for` loop, called an **implicit loop**.

In this case:

```
for item in group:
    doit(group)
```

becomes:

```
[doit(group) for item in group]
```

with the additional feature that everything returned by `doit(group)` for each item of `group` is put in a list.

```
# split the line on whitespace

# implicit for loop
float_data = [float(line_data) for line_data in data[0].split()]

# this is the string list
print(line)
# this is the float list
print(float_data)
```

```
['1950', '-1.03', '-1.133', '-1.283', '-1.071', '-1.434', '-1.412', '-1.269', '-1.042
↪', '-.631', '-.406', '-1.138', '-1.235']
[1950.0, -1.03, -1.133, -1.283, -1.071, -1.434, -1.412, -1.269, -1.042, -0.631, -0.
↪406, -1.138, -1.235]
```

The statement:

```
float_data = [float(line_data) for line_data in line]
```

is much more Pythonic than the code above. It is simple, elegant and neat.

We can *nest* for statements, i.e. put one for loop inside another. This allows us to treat data of multiple dimensions.

In the examples above, we converted only the data in `data[0]` to a list of floating point numbers. If we wanted to process *all* lines of data, we would have to loop over them as well, in an 'outer' loop.

```
# use a step of 10 for illustration purposes
# to save space when printing

step = 10

for index,line in enumerate(data_table.splitlines()[1:-1:step]):
    # convert each line to list of floats
    float_data = [float(line_data) for line_data in line.split()]
    print('line {} is {}'.format(index*step,float_data))
```

```
line 0 is [1950.0, -1.03, -1.133, -1.283, -1.071, -1.434, -1.412, -1.269, -1.042, -0.
↪631, -0.406, -1.138, -1.235]
line 10 is [1960.0, -0.287, -0.253, -0.082, 0.007, -0.322, -0.287, -0.318, -0.25, -0.
↪47, -0.332, -0.308, -0.39]
line 20 is [1970.0, 0.38, 0.432, 0.228, 0.014, -0.099, -0.636, -1.055, -1.007, -1.251,
↪ -1.068, -1.063, -1.203]
line 30 is [1980.0, 0.677, 0.601, 0.684, 0.912, 0.958, 0.911, 0.769, 0.329, 0.263, 0.
↪223, 0.27, 0.111]
line 40 is [1990.0, 0.243, 0.573, 0.951, 0.46, 0.652, 0.511, 0.147, 0.127, 0.366, 0.
↪303, 0.4, 0.362]
line 50 is [2000.0, -1.122, -1.189, -1.09, -0.397, 0.251, 0.001, -0.159, -0.145, -0.
↪238, -0.367, -0.701, -0.55]
line 60 is [2010.0, 1.066, 1.526, 1.462, 0.978, 0.658, -0.228, -1.103, -1.671, -1.879,
↪ -1.888, -1.472, -1.558]
```

Note that whilst we have calculated `float_data` in the loop for each line, it gets over-written with each new line as things stand.

We can do the same thing, and generate a list of the responses more neatly, using an implicit loop inside another implicit loop:

```
all_float_data = [[float(line_data) for line_data in line.split()] for line in data_
↪table.splitlines()[1:-1]]
```

The variable `all_float_data` is now a sort of 'two dimensional' list, within which we can refer to individual items as e.g. `all_float_data[10][3]` for row `10`, column `3`.

Let's use this idea to print out column 0 of each row (containing the `YEAR` data). We will use the method `range(nrows)` that (implicitly) generates a list `[0,1,2,3, ..., nrows-1]`.

Notice the use of `end=' '` in the `print` statement. This replaces the usual newline by whatever is specified by the keyword `end`. Note also that we have used `{:.0f}` to specify the format term. This indicates that the term is to be printed as a floating point number (the `f`) with zero numbers after the decimal point (`.0`)

```
nrows = len(all_float_data)
i = 0

print('column {} of the data gives:\n'.format(i))
for row in range(nrows):
    print('{:.0f}'.format(all_float_data[row][i]),end=' ')
```

```
column 0 of the data gives:

1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965 1966␣
↪1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980 1981 1982␣
↪1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998␣
↪1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014␣
↪2015 2016 2017
```

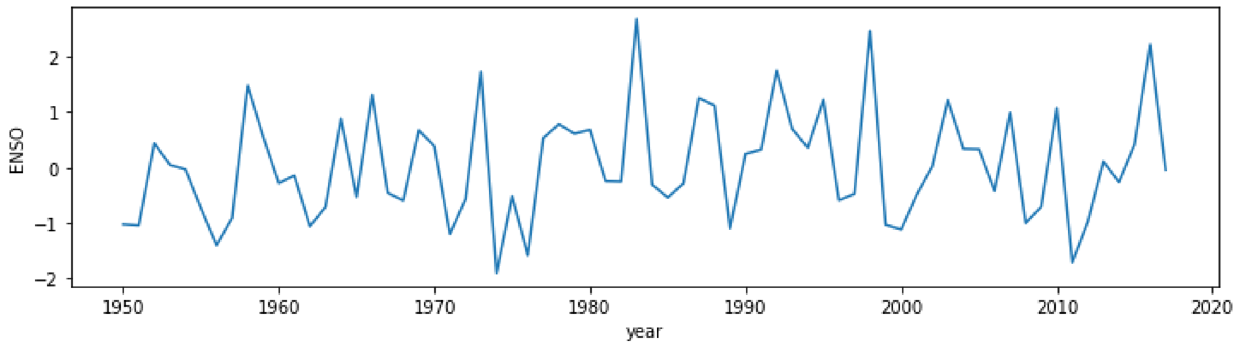
Exercise 1.2.9

- use an implicit loop to create a list of ENSO values in a variable `enso` for the years 1950 up to last year for the period `DECJAN`.
- produce a plot of ENSO for `DECJAN` as a function of year (see below on how to do that).

Hint: check which column in the header is `DECJAN`. To start you off on this, we give you the implicit loop code for extracting the column containing the `YEAR` data (column 0). We also give you the code to achieve the plotting.

```
# do exercise here
# ANSWER

# generate a list called years of column 0 data
years = [all_float_data[row][0] for row in range(nrows)]

# very similar to aboive, but using column 1
enso = [all_float_data[row][1] for row in range(nrows)]

# for plotting
import pylab as plt
%matplotlib inline

#
plt.figure(0,figsize=(12,3))
plt.plot(years,enso)
plt.xlabel('year')
plt.ylabel('ENSO')
```

```
Text(0, 0.5, 'ENSO')
```



1.2.6 Summary

In section 1.2 you have been introduced to text representation in Python, as strings (type `str`), and shown that this sort of variable can be thought of an an 'array', and that it has a length attribute that can be accessed with `len()`.

Other useful string manipulation methods you were introduced to are: `replace()`, `find()`, `split()` and `splitlines()`, though of course there are many more.

In an 'array', we can use an index to refer to a particular item (e.g. index 0 for the first item, 1 for the second, -1 for the last). We can use this idea to manipulate strings.

In a more general sense, we can take a 'slice' of an array, with the syntax `[start:stop:skip]` giving access to a regularly spaced part of an array. We can use this, for example, to print out every 10th value (`skip=10`).

You were also introduced to the idea of looping control structures, using a `for ... in ...:` statement, and the equivalent implicit form. This introduced the idea of indented code blocks and (related) nested structures (loops within loops).

In passing, you have also been shown how to pull html data from a URL (scraping) using the `` `requests `` <http://docs.python-requests.org/en/master/>'__ package, and also how to produce a simple data plot, using `` `pylab `` <https://matplotlib.org/index.html>'__.

1.3. Groups of things

Very often, we will want to group items together. There are several main mechanisms for doing this in Python, known as:

- string e.g. `hello`
- tuple, e.g. `(1, 2, 3)`
- list, e.g. `[1, 2, 3]`
- numpy array e.g. `np.array([1, 2, 3])`

A slightly different form of group is a dictionary:

- dict, e.g. `{1:'one', 2:'two', 3:'three'}`

You will notice that each of the grouping structures tuple, list and dict use a different form of bracket. The numpy array is fundamental to much work that we will do later.

We have dealt with the idea of a string as an ordered collection in the material above, so will deal with the others here.

We noted the concept of length (`len()`), that elements of the ordered collection could be accessed via an index, and came across the concept of a slice. All of these same ideas apply to the first set of groups (string, tuple, list, numpy array) as they are all ordered collections.

A dictionary is not (by default) ordered, however, so indices have no role. Instead, we use 'keys'.

1.3.1 `tuple`

A tuple is a group of items separated by commas. In the case of a tuple, the brackets are optional. You can have a group of diffent types in a tuple (e.g. int, int, str, bool)

```
# load into the tuple
t = (1, 2, 'three', False)

# unload from the tuple
a,b,c,d = t

print(t)
print(a,b,c,d)
```

```
(1, 2, 'three', False)
1 2 three False
```

If there is only one element in a tuple, you must put a comma , at the end, otherwise it is not interpreted as a tuple:

```
t = (1)
print (t,type(t))
t = (1,)
print (t,type(t))
```

```
1 <class 'int'>
(1,) <class 'tuple'>
```

You can have an empty tuple though:

```
t = ()
print (t,type(t))
```

```
() <class 'tuple'>
```

E1.3.1 Exercise

- create a tuple called t that contains the integers 1 to 5 inclusive
- print out the value of t
- use the tuple to set variables a1,a2,a3,a4,a5

```
# do exercise here
# ANSWER

t = (1,2,3,4,5)
print(t)
a1,a2,a3,a4,a5 = t
print(a1,a2,a3,a4,a5)
```

```
(1, 2, 3, 4, 5)
1 2 3 4 5
```

1.3.2 `list`

A `list` is similar to a `tuple`. One main difference is that you can change individual elements in a list but not in a tuple. To convert between a list and tuple, use the 'casting' methods `list()` and `tuple()`:

```
# a tuple
t0 = (1,2,3)

# cast to a list
l = list(t0)

# cast to a tuple
t = tuple(l)

print('type of {} is {}'.format(t,type(t)))
print('type of {} is {}'.format(l,type(l)))
```

```
type of (1, 2, 3) is <class 'tuple'>
type of [1, 2, 3] is <class 'list'>
```

You can concatenate (join) lists or tuples with the + operator:

```
l0 = [1,2,3]
l1 = [4,5,6]

l = l0 + l1
print ('joint list:',l)
```

```
joint list: [1, 2, 3, 4, 5, 6]
```

E1.3.2 Exercise * copy the code from the cell above, but instead of lists, use tuples * loop over each element in the tuple and print out the data type and value of the element

Hint: use a `for ... in ...` construct.

```
# do exercise here
# ANSWER

l0 = (1,2,3)
l1 = (4,5,6)

l = l0 + l1
print ('joint tuple:',l)

for t in l:
    print(type(t),t)
```

```
joint tuple: (1, 2, 3, 4, 5, 6)
<class 'int'> 1
<class 'int'> 2
<class 'int'> 3
<class 'int'> 4
<class 'int'> 5
<class 'int'> 6
```

A common method associated with lists or tuples is: * `index()`

Some useful methods that will operate on lists and tuples are: * `len()` * `sort()` * `min()`, `max()`

```
l0 = (2,8,4,32,16)

# print the index of the item integer 4
# in the tuple / list

item_number = 4

# Note the dot . here
# as index is a method of the class list
ind  = l0.index(item_number)

# notice that this is different
# as len() is not a list method, but
# does operatate on lists/tuples
# Note: do not use len as a variable name!
llen = len(l0)

# note the use of integers in the braces e.g. {0}
# rather than empty braces as before. This allows us to
# refer to particular items in the format argument list
print('the index of {0} in {1} is {2}'.format(item_number,l0,ind))
print('the length of the {0} {1} is {2}'.format(type(l0),l0,llen))
```

```
the index of 4 in (2, 8, 4, 32, 16) is 2
the length of the <class 'tuple'> (2, 8, 4, 32, 16) is 5
```

E1.3.3 Exercise

- copy the code to the block below, and test that this works with lists, as well as tuples
- find the index of the integer 16 in the tuple/list
- what is the index of the first item?
- what is the length of the tuple/list?

- what is the index of the last item?

```
# do exercise here
# ANSWER

# change this to a list
l0 = [2,8,4,32,16]

# print the index of the item integer 4
# in the tuple / list

item_number = 4

# Note the dot . here
# as index is a method of the class list
ind  = l0.index(item_number)

# notice that this is different
# as len() is not a list method, but
# does operatate on lists/tuples
# Note: do not use len as a variable name!
llen = len(l0)

# note the use of integers in the braces e.g. {0}
# rather than empty braces as before. This allows us to
# refer to particular items in the format argument list
print('the index of {0} in {1} is {2}'.format(item_number,l0,ind))
print('the length of the {0} {1} is {2}'.format(type(l0),l0,llen))

'''
find the index of the integer 16 in the tuple/list
what is the index of the first item?
what is the length of the tuple/list?
what is the index of the last item?
'''

print('16 is item number',l0.index(16))
print('the first item has index 0:',l0.index(2))
print('list length',len(l0))
print('the last item has index 4 (len - 1):',l0.index(16))
```

```
the index of 4 in [2, 8, 4, 32, 16] is 2
the length of the <class 'list'> [2, 8, 4, 32, 16] is 5
16 is item number 4
the first item has index 0: 0
list length 5
the last item has index 4 (len - 1): 4
```

A list has a much richer set of methods than a tuple. This is because we can add or remove list items (but not tuple).

- `insert(i,j)` : insert `j` beore item `i` in the list
- `append(j)` : append `j` to the end of the list
- `sort()` : sort the list

This shows that tuples and lists are 'ordered' (i.e. they maintain the order they are loaded in) so that indiviual elements may be accessed through an 'index'. The index values start at 0 as we saw above. The index of the last element in a list/tuple is the length of the group, minus 1. This can also be referred to an index `-1`.

```
l0 = [2,8,4,32,16]

# insert 64 at the begining (before item 0)
# Note that this inserts 'in place'
# i.e. the list is changed by calling this
l0.insert(0,64)


# insert 128 *before* the last item (item -1)
l0.insert(-1,128)

# append 256 on the end
l0.append(256)

# copy the list
# and sort the copy
# Note the use of the copy() method here
# to create a copy
l1 = l0.copy()

# Note that this sorts 'in place'
# i.e. the list is changed by calling this
l1.sort()

print('the list {0} once sorted is {1}'.format(l0,l1))
```

```
the list [64, 2, 8, 4, 32, 128, 16, 256] once sorted is [2, 4, 8, 16, 32, 64, 128,␣
↪256]
```

E1.3.4 Exercise

- copy the above code and try out some different locations for inserting values (e.g. what does index `-2` mean?)
- what happens if you take off the `.copy()` statement in the line `l1 = l0.copy()`, i.e. just use `l1 = l0`? Why is this?

```
# do exercise here
# ANSWER

l0 = [2,8,4,32,16]

# insert 64 at index -2 (2 from end)
# Note that this inserts 'in place'
# i.e. the list is changed by calling this
l0.insert(-2,64)
print(l0)

# copy the list
# and sort the copy
# Note the use of the copy() method here
# to create a copy
l1 = l0.copy()

# Note that this sorts 'in place'
# i.e. the list is changed by calling this
l1.sort()

print('the list {0} once sorted is {1}'.format(l0,l1))
```

```
[2, 8, 4, 64, 32, 16]
the list [2, 8, 4, 64, 32, 16] once sorted is [2, 4, 8, 16, 32, 64]
```

```
# do exercise here
# ANSWER

l0 = [2,8,4,32,16]

# insert 64 at index -2 (2 from end)
# Note that this inserts 'in place'
# i.e. the list is changed by calling this
l0.insert(-2,64)
print(l0)

# copy the list
# and sort the copy
# Note the use of the copy() method here
# to create a copy
l1 = l0

# Note that this sorts 'in place'
# i.e. the list is changed by calling this
l1.sort()

'''
Note that without the copy, we have altered l0
even though we did l1.sort()
'''

print('the list {0} once sorted is {1}'.format(l0,l1))

'''
For not in place sort, use sorted()
'''

l0 = [2,8,4,32,16]
print('the list {0} once sorted is {1}'.format(l0,sorted(l0)))
```

```
[2, 8, 4, 64, 32, 16]
the list [2, 4, 8, 16, 32, 64] once sorted is [2, 4, 8, 16, 32, 64]
the list [2, 8, 4, 32, 16] once sorted is [2, 4, 8, 16, 32]
```

1.3.3 `np.array`

An array is a group of objects of the same type. Because they are of the same type, they can be stored efficiently in compter memory, and also accessed efficiently.

Whilst there are different ways of forming arrays, the most common is to use numpy arrays, using the package `numpy`. To use this, we must first import the package into the current workspace. We do this with the `import` method. Using the optional `as` statement allows us to use a shorter (or more suitable) name for the package. We will generally call numpy `np`, so we use:

```
import numpy as np
```

to import ('load') the numpy package.

Often, we will read data from a file/URL as we did above for the ENSO dataset. In that case, we had to step through each item to convert from string form to floating point number.

This sort of thing is much more simply done using methods associated with numpy arrays.

A particularly useful numpy method is `np.loadtxt(file)` that loads an ASCII table of data straight into a numpy array.

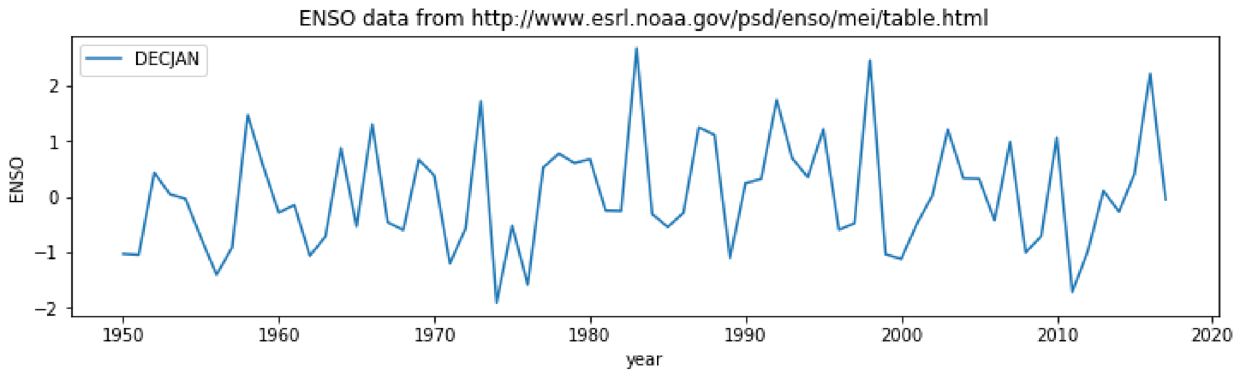
Whilst this is designed to load data from a file, we can use `io.StringIO()` from the `io` package to make data that we already have as a string seem to `np.loadtxt` as if it were a file. This is a useful 'trick' for using methods that expect data in a file. The `unpack=True` option makes sure the data array is compoised the way we would expect it. The `usecols` option lets us select only those data columns we wish to read (0 and 1 here).

An alternative to `np.loadtxt()` is `np.genfromtxt()`. This has some additional features, such the `invalid_raise` flag. If this is set `False`, the loading is made somewhat tolerant to data errors (e.g. inconsistent number of columns). Further, we can explicitly set what will indicate `missing_values` in the input and what we would like to replace them with (`filling_values`) which can be useful for tidying up datasets.

```
import requests
import numpy as np
import io

# access dataset as above
url = "http://www.esrl.noaa.gov/psd/enso/mei/table.html"
txt = requests.get(url).text

# copy the useful data
start_head = txt.find('YEAR')
start_data = txt.find('1950\t')
stop_data  = txt.find('2018\t')

# select a data column
data_column = 1

header = txt[start_head:start_data].split()
data = np.loadtxt(io.StringIO(txt[start_data:stop_data]),unpack=True,usecols=[0,data_
↪column])

# so data[0] is the year data
#    data[1] is the enso data for column data_column
# print some attributes of the data array

print('array type',type(data))
print('data type',data.dtype)
print('number of dimensions',data.ndim)
print('data shape',data.shape)
print('data size',data.size)

# for plotting
import pylab as plt
%matplotlib inline

#
plt.figure(0,figsize=(12,3))
plt.plot(data[0],data[1],label=header[data_column])
plt.xlabel('year')
plt.ylabel('ENSO')
plt.title('ENSO data from {0}'.format(url))
plt.legend(loc='best')
```

```
array type <class 'numpy.ndarray'>
data type float64
number of dimensions 2
data shape (2, 68)
data size 136
```

```
<matplotlib.legend.Legend at 0x11b8b2668>
```



We saw in the example above that a numpy array (`<class 'numpy.ndarray'>`) has a set of attributes that include `shape`, `ndim`, `dtype` and `size` that we can use to query information about the array. We will learn morre about processing data with numpy arrays later in the course, but you should already see that they are a useful construct for manipulating multi-dimensional datasets.

Exercise 1.3.4

- copy the code from the block above and modify it to plot the ENSO data for the period `FEBMAR`. Check this by looking at the data in the original table.
- modify the code to produce a plot of *all* periods (so the graph should have 12 lines, correctly labelled)

Hint: You will need to consider what, if anything to set of `usecols` (what happends if you don't set `usecols`?) and provide a looping structure for the plotting.

```
# do exercise here
#ANSWER

import requests
import numpy as np
import io

# access dataset as above
url = "http://www.esrl.noaa.gov/psd/enso/mei/table.html"
txt = requests.get(url).text

# copy the useful data
start_head = txt.find('YEAR')
start_data = txt.find('1950\t')
stop_data  = txt.find('2018\t')

# select a data column
'''
Use column 3 for FEBMAR

The code is written so that we only need
```

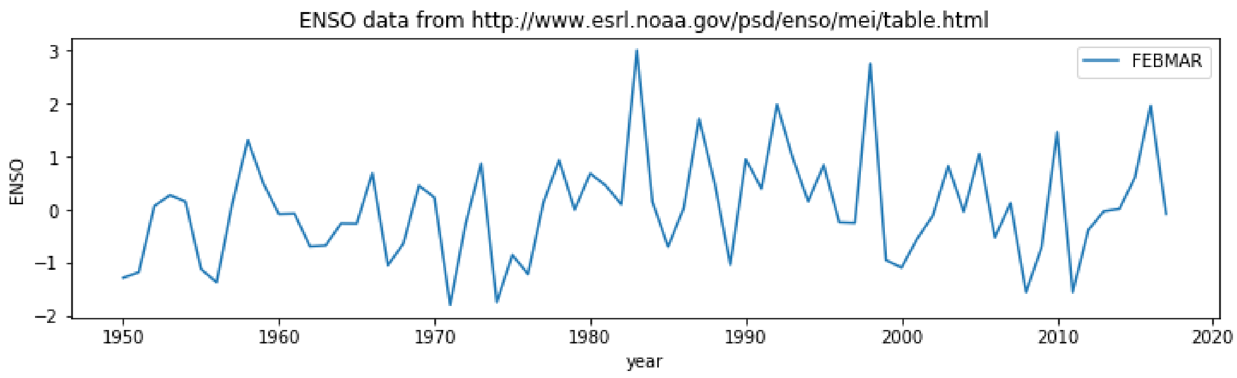
(continues on next page)

(continued from previous page)

```
change this one variable
'''
data_column = 3

header = txt[start_head:start_data].split()
data = np.loadtxt(io.StringIO(txt[start_data:stop_data]),unpack=True,usecols=[0,data_
↪column])

# so data[0] is the year data
#    data[1] is the enso data for column data_column
# print some attributes of the data array

print('array type',type(data))
print('data type',data.dtype)
print('number of dimensions',data.ndim)
print('data shape',data.shape)
print('data size',data.size)

# for plotting
import pylab as plt
%matplotlib inline

#
plt.figure(0,figsize=(12,3))

plt.plot(data[0],data[1],label=header[data_column])
plt.xlabel('year')
plt.ylabel('ENSO')
plt.title('ENSO data from {0}'.format(url))
plt.legend(loc='best')
```

```
array type <class 'numpy.ndarray'>
data type float64
number of dimensions 2
data shape (2, 68)
data size 136
```

```
<matplotlib.legend.Legend at 0x11bcb0a20>
```



1.3.4 dict

The collections we have used so far have all been ordered. This means that we can refer to a particular element in the group by an index, e.g. `array[10]`.

A dictionary is not (by default) ordered. Instead of indices, we use 'keys' to refer to elements: each element has a key associated with it. It can be very useful for data organisation (e.g. databases) to have a key to refer to, rather than e.g. some arbitrary column number in a gridded dataset.

A dictionary is defined as a group in braces (curley brackets). For each elerment, we specify the key and then the value, separated by `:`.

```
a = {'one': 1, 'two': 2, 'three': 3}

# we then refer to the keys and values in the dict as:

print ('a:\n\t',a)
print ('a.keys():\n\t',a.keys())      # the keys
print ('a.values():\n\t',a.values()) # returns the values
print ('a.items():\n\t',a.items())    # returns a list of tuples
```

```
a:
     {'one': 1, 'two': 2, 'three': 3}
a.keys():
     dict_keys(['one', 'two', 'three'])
a.values():
     dict_values([1, 2, 3])
a.items():
     dict_items([('one', 1), ('two', 2), ('three', 3)])
```

Because dictionaries are not ordered, we cannot guarantee the order they will come out in a `for` loop, but we will often use such a loop to iterate over the items in a dictionary.

```
for key,value in a.items():
    print(key,value)
```

```
one 1
two 2
three 3
```

We refer to specific items using the key e.g.:

```
print(a['one'])
```

```
1
```

You can add to a dictionary:

```
a.update({'four':4,'five':5})
print(a)

# or for a single value
a['six'] = 6
print(a)
```

```
{'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5}
{'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5, 'six': 6}
```

Quite often, you find that you have the keys you want to use in a dictionary as a list or array, and the values in another list.

In such a case, we can use the method `zip(keys,values)` to load into the dictionary. For example:

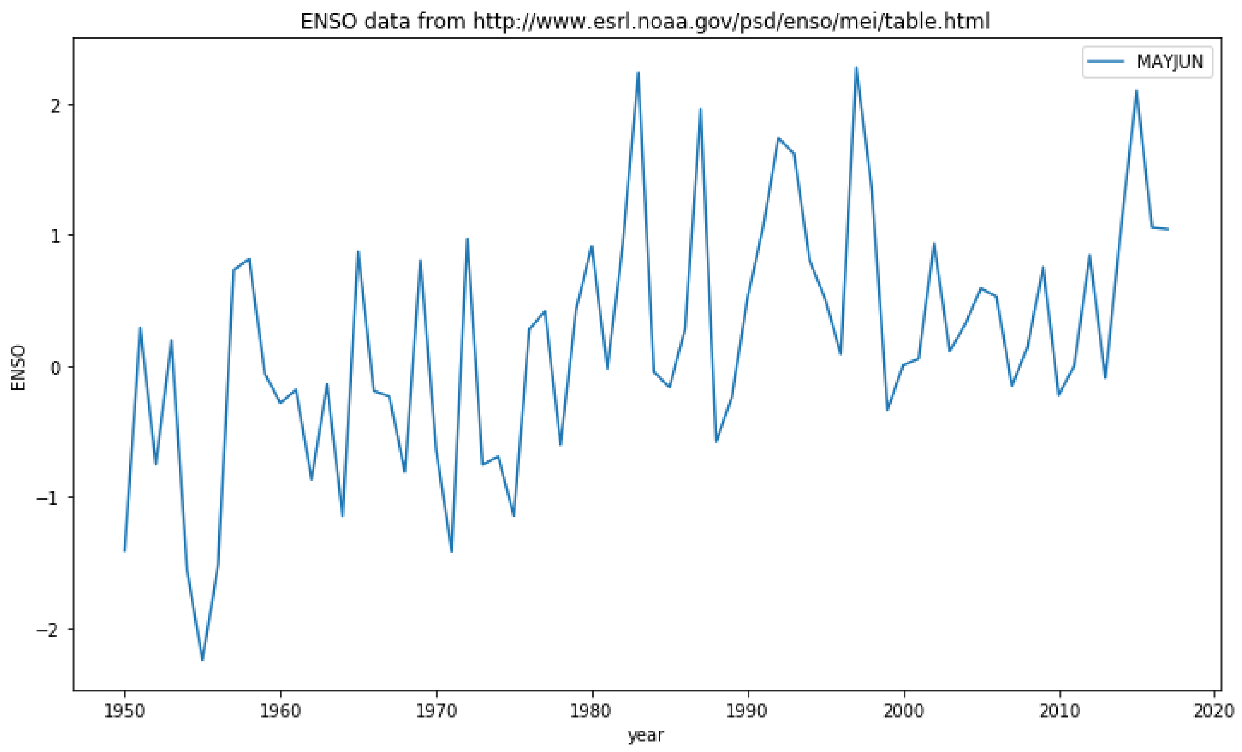
```
values = [1,2,3,4]
keys = ['one','two','three','four']

a = dict(zip(keys,values))

print(a)
```

```
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

We will use this idea to make a dictionary of our ENSO dataset, using the items in the header for the keys. In this way, we obtain a more elegant representation of the dataset, and can refer to items by names (keys) instead of column numbers.

```
import requests
import numpy as np
import io

# access dataset as above
url = "http://www.esrl.noaa.gov/psd/enso/mei/table.html"
txt = requests.get(url).text

# copy the useful data
start_head = txt.find('YEAR')
start_data = txt.find('1950\t')
stop_data  = txt.find('2018\t')

header = txt[start_head:start_data].split()
data = np.loadtxt(io.StringIO(txt[start_data:stop_data]),unpack=True)

# use zip to load into a dictionary
data_dict = dict(zip(header, data))

key = 'MAYJUN'
# plot data
plt.figure(0,figsize=(12,7))
plt.title('ENSO data from {0}'.format(url))
plt.plot(data_dict['YEAR'],data_dict[key],label=key)
plt.xlabel('year')
plt.ylabel('ENSO')
plt.legend(loc='best')
```

```
<matplotlib.legend.Legend at 0x11bdc31d0>
```



Exercise 1.3.5

- copy the code above, and modify so that datasets for months `['MAYJUN','JUNJUL','JULAUG']` are plotted on the graph

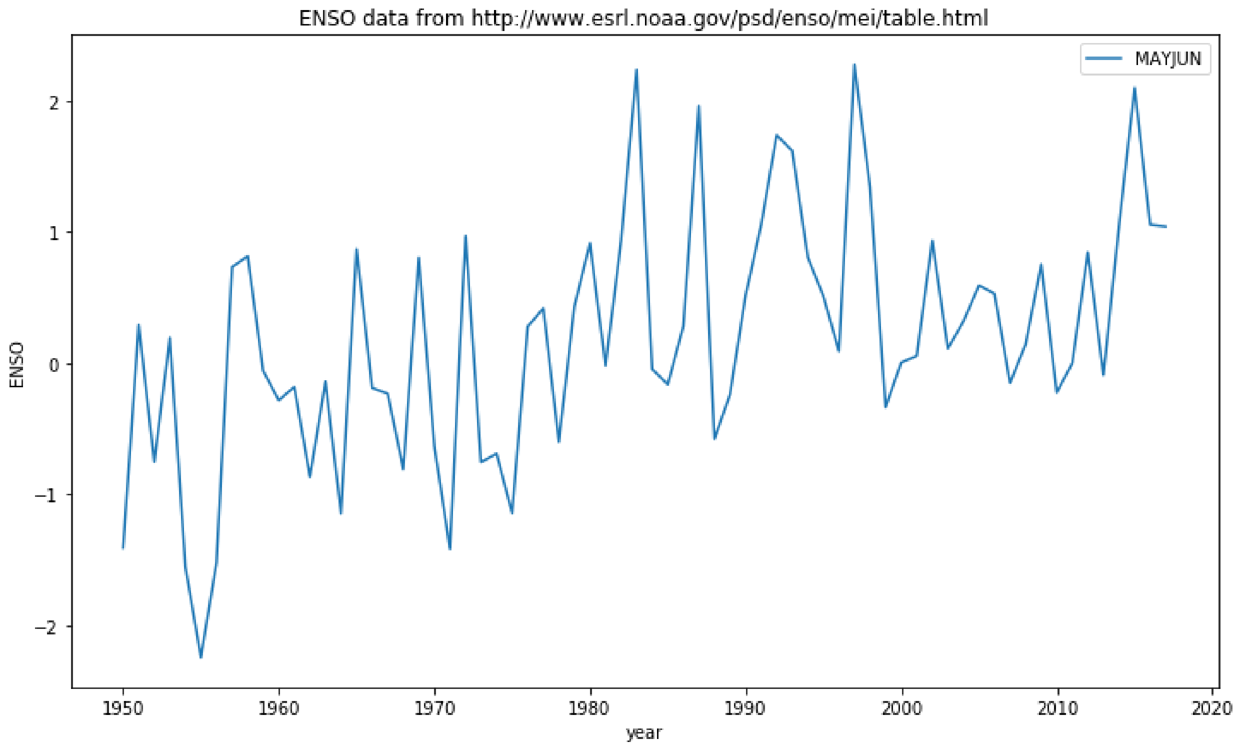
Hint: use a for loop

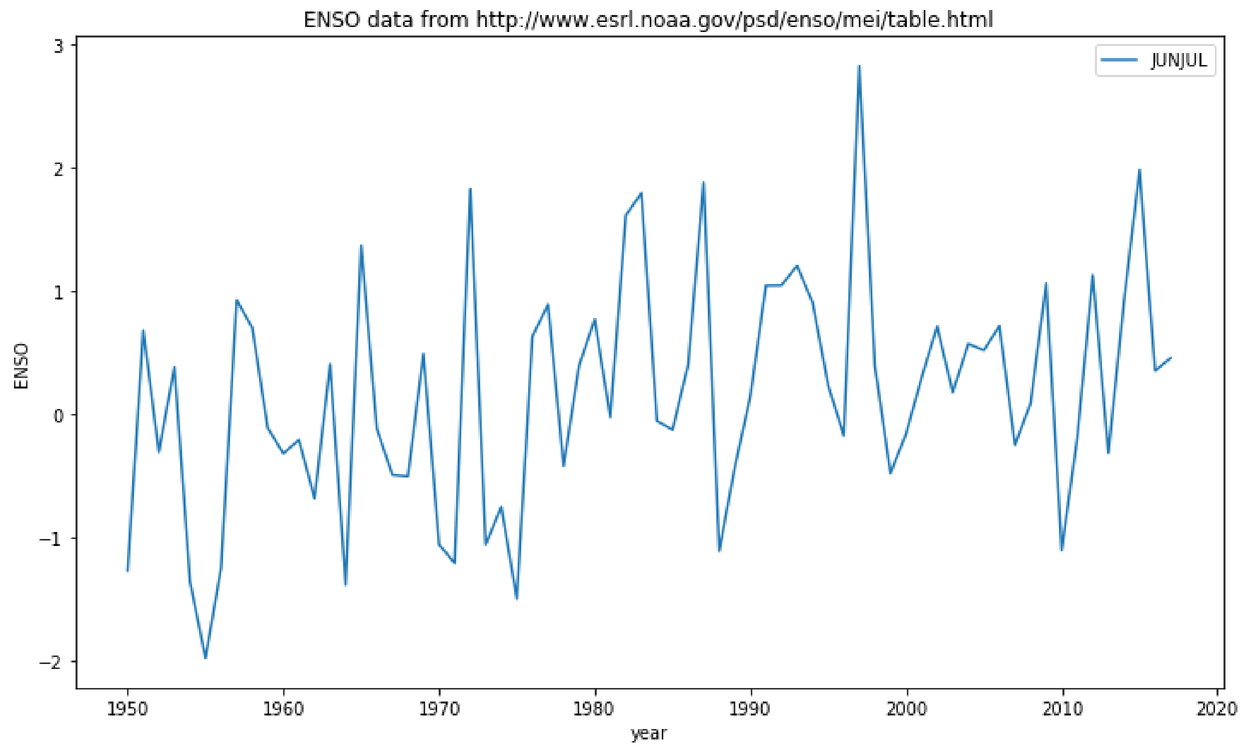
```
# do exercise here
# ANSWER

import requests
import numpy as np
import io

# access dataset as above
url = "http://www.esrl.noaa.gov/psd/enso/mei/table.html"
txt = requests.get(url).text

# copy the useful data
start_head = txt.find('YEAR')
start_data = txt.find('1950\t')
stop_data  = txt.find('2018\t')

header = txt[start_head:start_data].split()
data = np.loadtxt(io.StringIO(txt[start_data:stop_data]),unpack=True)

# use zip to load into a dictionary
data_dict = dict(zip(header, data))


'''
Do the loop here
```

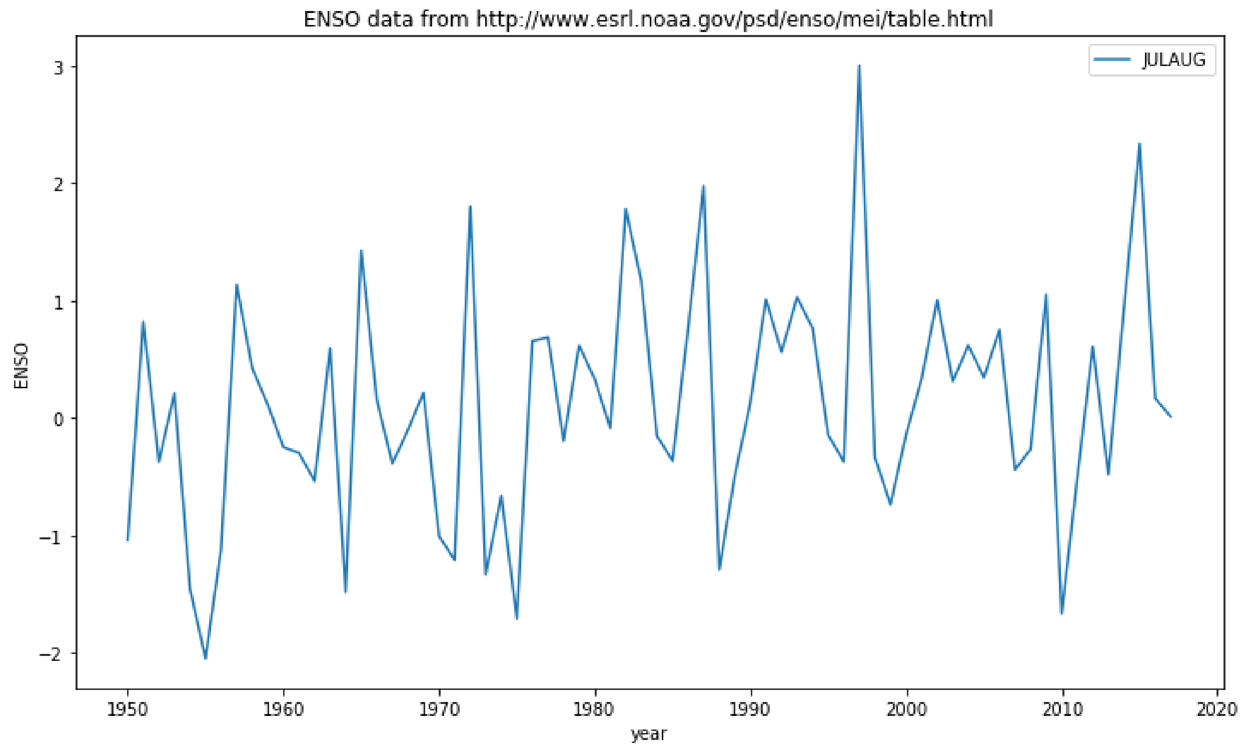
(continues on next page)

(continued from previous page)

```
'''
for i,key in enumerate(['MAYJUN','JUNJUL','JULAUG']):
    # plot data
    '''
    Use enumeration i as figure number
    '''
    plt.figure(i,figsize=(12,7))
    plt.title('ENSO data from {0}'.format(url))
    plt.plot(data_dict['YEAR'],data_dict[key],label=key)
    plt.xlabel('year')
    plt.ylabel('ENSO')
    plt.legend(loc='best')
```







We can also usefully use a dictionary with a printing format statement. In that case, we refer directly to the key in ther format string. This can make printing statements much easier to read. We don;'t directly pass the dictionary to the `fortmat` staterment, but rather `**dict`, where `**dict` means "treat the key-value pairs in the dictionary as additional named arguments to this function call".

So, in the example:

```python
import requests
import numpy as np
import io

# access dataset as above
url = "http://www.esrl.noaa.gov/psd/enso/mei/table.html"
txt = requests.get(url).text

# copy the useful data
start_head = txt.find('YEAR')
start_data = txt.find('1950\t')
stop_data  = txt.find('2018\t')

header = txt[start_head:start_data].split()
data = np.loadtxt(io.StringIO(txt[start_data:stop_data]),unpack=True)

# use zip to load into a dictionary
data_dict = dict(zip(header, data))
print(data_dict.keys())

# print the data for MAYJUN
print('data for MAYJUN: {MAYJUN}'.format(**data_dict))
```

```
dict_keys(['YEAR', 'DECJAN', 'JANFEB', 'FEBMAR', 'MARAPR', 'APRMAY', 'MAYJUN', 'JUNJUL
↪', 'JULAUG', 'AUGSEP', 'SEPOCT', 'OCTNOV', 'NOVDEC'])
data for MAYJUN: [-1.412e+00  2.880e-01 -7.560e-01  1.910e-01 -1.558e+00 -2.247e+00
 -1.523e+00  7.300e-01  8.120e-01 -6.200e-02 -2.870e-01 -1.850e-01
 -8.700e-01 -1.440e-01 -1.150e+00  8.670e-01 -1.930e-01 -2.360e-01
 -8.120e-01  8.010e-01 -6.360e-01 -1.420e+00  9.660e-01 -7.580e-01
 -6.940e-01 -1.148e+00  2.760e-01  4.140e-01 -6.050e-01  4.290e-01
  9.110e-01 -2.400e-02  9.300e-01  2.235e+00 -4.900e-02 -1.660e-01
  2.790e-01  1.958e+00 -5.820e-01 -2.420e-01  5.110e-01  1.051e+00
  1.735e+00  1.616e+00  8.030e-01  5.070e-01  8.700e-02  2.275e+00
  1.336e+00 -3.390e-01  1.000e-03  5.200e-02  9.320e-01  1.070e-01
  3.150e-01  5.890e-01  5.260e-01 -1.550e-01  1.420e-01  7.510e-01
 -2.280e-01 -3.000e-03  8.420e-01 -9.400e-02  1.046e+00  2.097e+00
  1.053e+00  1.039e+00]
```

The line `print('data for MAYJUN: {MAYJUN}'.format(**data_dict))` is equivalent to writing:

```python
print('data for {MAYJUN}'.format(YEAR=data_dict[YEAR],DECJAN=data_dict[DECJAN], ...))
```

In this way, we use the keys in the dictionary as keywords to pass to a method.

Another useful example of such a use of a dictionary is in saving a numpy dataset to file.

If the data are numpy arrays in a dictionary as above, we can store the dataset using:

```python
import requests
import numpy as np
import io

# access dataset as above
url = "http://www.esrl.noaa.gov/psd/enso/mei/table.html"
txt = requests.get(url).text

# copy the useful data
start_head = txt.find('YEAR')
```

(continues on next page)

(continued from previous page)

```
start_data = txt.find('1950\t')
stop_data  = txt.find('2018\t')

header = txt[start_head:start_data].split()
data = np.loadtxt(io.StringIO(txt[start_data:stop_data]),unpack=True)

# use zip to load into a dictionary
data_dict = dict(zip(header, data))

filename = 'enso_mei.npz'

# save the dataset
np.savez_compressed(filename,**data_dict)
```

What we load from the file is a dictionary-like object `<class 'numpy.lib.npyio.NpzFile'>`.

If needed, we can cast this to a dictionary with `dict()`, but it is generally more efficient to keep the original type.

```
# load the dataset

filename = 'enso_mei.npz'

loaded_data = np.load(filename)

print(type(loaded_data))

# test they are the same using np.array_equal
for k in loaded_data.keys():
    print('\t',k,np.array_equal(data_dict[k], loaded_data[k]))
```

```
<class 'numpy.lib.npyio.NpzFile'>
     YEAR True
     DECJAN True
     JANFEB True
     FEBMAR True
     MARAPR True
     APRMAY True
     MAYJUN True
     JUNJUL True
     JULAUG True
     AUGSEP True
     SEPOCT True
     OCTNOV True
     NOVDEC True
```

Exercise 1.3.6

- Using what you have learned above, access the Met Office data file `` `https://www.metoffice.gov.uk/hadobs/hadukp/data/monthly/HadSEEP_monthly_qc.txt `` <https://www.metoffice.gov.uk/hadobs/hadukp/data/monthly/HadSEEP_monthly_qc.txt>'__ and create a 'data package' in a numpy `.npz` file that has keys of `YEAR` and each month in the year, with associated datasets of Monthly Southeast England precipitation (mm).
- confirm that tha data in your `npz` file is the same as in your original dictionary
- produce a plot of October rainfall using these data for the years 1900 onwards

```
# do exercise here
# ANSWER

'''
Exploration of dataset shows:


Monthly Southeast England precipitation (mm). Daily automated values used after 1996.
Wigley & Jones (J.Climatol.,1987), Gregory et al. (Int.J.Clim.,1991)
Jones & Conway (Int.J.Climatol.,1997), Alexander & Jones (ASL,2001). Values may␣
↪change after QC.
YEAR   JAN   FEB   MAR   APR   MAY   JUN   JUL   AUG   SEP   OCT   NOV   DEC   ANN
 1873  87.1  50.4  52.9  19.9  41.1  63.6  53.2  56.4  62.0  86.0  59.4  15.7  647.7
 1874  46.8  44.9  15.8  48.4  24.1  49.9  28.3  43.6  79.4  96.1  63.9  52.3  593.5

so we have 3 lines of header
then the column titles
then the data

we can define these as before using

txt.find('YEAR')
start_data = txt.find('1873')
stop_data = None


Other than the filenames then, the code
is identical
'''

import requests
import numpy as np
import io

# access dataset as above
url = "https://www.metoffice.gov.uk/hadobs/hadukp/data/monthly/HadSEEP_monthly_qc.txt"
txt = requests.get(url).text

# copy the useful data
start_head = txt.find('YEAR')
start_data = txt.find('1873')
stop_data  = None

header = txt[start_head:start_data].split()
data = np.loadtxt(io.StringIO(txt[start_data:stop_data]),unpack=True)

# use zip to load into a dictionary
data_dict = dict(zip(header, data))

filename = 'HadSEEP_monthly_qc.npz'

# save the dataset
np.savez_compressed(filename,**data_dict)
```

```
# ANSWER

loaded_data = np.load(filename)
```

(continues on next page)

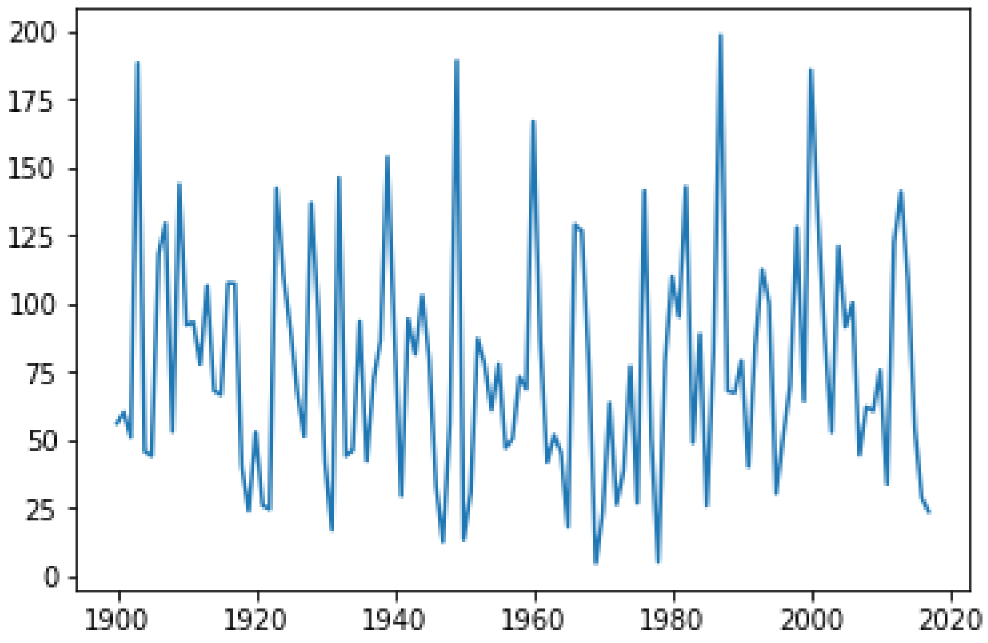
(continued from previous page)

```
print(type(loaded_data))

# test they are the same using np.array_equal
for k in loaded_data.keys():
    print('\t',k,np.array_equal(data_dict[k], loaded_data[k]))
```

```
<class 'numpy.lib.npyio.NpzFile'>
     YEAR True
     JAN True
     FEB True
     MAR True
     APR True
     MAY True
     JUN True
     JUL True
     AUG True
     SEP True
     OCT True
     NOV True
     DEC True
     ANN True
```

```
# ANSWER

'''
October rainfall, 1900+
'''

year = loaded_data['YEAR']

# mask where years match
mask = year  >= 1900

oct = loaded_data['OCT']

# set invalid data points to nan
oct[oct<0] = np.nan

plt.plot(year[mask],oct[mask])
```

```
[<matplotlib.lines.Line2D at 0x11c76a9e8>]
```



1.3.5 Summary

In this section, we have extended the types of data we might come across to include groups . We dealt with ordered groups of various types (`tuple`, `list`), and introduced the numpy package for numpy arrays (`np.array`). We saw dictionaries as collections with which we refer to individual items with a key.

We learned in the previous section how to pull apart a dataset presented as a string using loops and various using methods and to construct a useful dataset 'by hand' in a list or similar structure. It is useful, when learning to program, to know how to do this.

Here, we saw that packages such as numpy provide higher level routines that make reading data easier, and we would generally use these in practice. We saw how we can use `zip()` to help load a dataset from arrays into a dictionary, and also the value of using a dictionary representation when saving numpy files.

2.8.3 1. Introduction to Python

Table of Contents

Python is a popular general purpose, free language. As opposed to other systems that are focused towards a particular task (e.g. R for statistics), Python has a strong following on the web, on systems operations and in data analysis. For scientific computing, a large number of useful add-ons ("libraries") are available to help you analyse and process data. This is an invaluable resource.

In addition to being free, Python is also very portable, and easy to pick up.

The aim of this Chapter is to introduce you to some of the fundamental concepts in Python. Mainly, this is based around fundamental data types in Python (`int`, `float`, `str`, `bool` etc.) and ways to group them (`tuple`, `list`, array, string and `dict`).

Although some of the examples we use are very simple to explain a concept, the more developed ones should be directly applicable to the sort of programming you are likely to need to do.

Further, a more advanced section of the chapter is available, that goes into some more detail and complkications. This too has a set of exercises with worked examples.

In this session, you will be introduced to some of the basic concepts in Python.

The session should last 4 hours (one week).

1.1 Getting Started

1.1.1 Comments and print function

Comments are statements ignored by the language interpreter.

Any text after a # in a *code block* is a comment.

You can 'run' the code in a code block using the 'run' widget (above) or hitting the keys ('typing') and at the same time.

E1.1.1 Exercise

- Try running the code block below
- Explain what happened ('what the computer did')

```
# Hello world
```

You can also use text blocks (contained in quotes) to contain comments, but note that if it is the last statement in the code, the text block may be printed out to the terminal.

```
# single line text
'hello world is not printed'

# multi-line text
'''
Hello
world
is printed
'''
```

```
'nHello nworldnis printedn'
```

E1.1.2 Exercise

- Copy the text from above in the window below
- Then put a new text block at the end and note down what happens
- What does the `\n` mean/do?

```
# do the exercise here
```

To print some statement (by default, to the screen you are using), use the `print` method:

```
print('hello world')
```

```
hello world
```

```
print('hello','world')
```

```
hello world
```

In Python 3.X, `print` is a function with the *argument(s)* (here, the string you want printed) enclosed in the function's (round) brackets.

E1.1.3 Exercise

- Copy the print statement from above code block.
- Change the words in the quotes and print them out.
- Add some comments to the code block explaining what you have done and seen.

```
# do the exercise here
```

1.1.2 Variables, Values and Data types

The idea of **variables** is fundamental to any programming. You can think of this as the *name* of *something*, so it is a way of allowing us to refer to some object in the language.

What the variable *is* set to is called its **value**.

So let's start with a variable we will call (*declare to be*) `x`.

We will give a *value* of the string `'one'` to this variable:

```
x = 'one'

print(x)
```

```
one
```

E1.1.4 Exercise

- set the a variable called x to some different string (e.g. 'hello world')
- print the value of the variable `x`
- Try this again, putting some 'newlines' (`\n`) in the string

```
# do the exercise here
```

```
# Now we set x to the value 1

x = 1

print(x,'is type',type(x))
```

```
1 is type <class 'int'>
```

In a computing language, the *sort of thing* the variable can be set to is called its **data type**.

In python, we can access this with the method `type()` as in the example above.

In the example above, the datatype is an **integer** number (e.g. `1, 2, 3, 4`).

In 'natural language', we might read the example above as 'x is one'.

E1.1.5 Exercise

- set the a variable called `x` to the integer `5`
- print the value and type of the variable `x`
- change the data type used for `x` to something else (e.g. a string)

```
# do the exercise here
```

Setting `x = 1` is different to:

```
x = 'one'
```

because here we have set value of the variable `x` to a **string** (i.e. some text).

A string is enclosed in quotes, e.g. `"one"` or `'one'`, or even `"'one'"` or `'"one"'`.

```
print ("one")
print ('one')
print ("'one'")
print ('"one"')
```

```
one
one
'one'
"one"
```

E1.1.6 Exercise

- create a variable `name` containing your name, as a string.
- using this variable and the print function, print out a statement such as `my name is Fred` (if your name were `Fred`)

```
# do the exercise here
```

Setting `x = 1` or `x = 'one'` is different to:

```
x = 1.0
```

because here we have set value of the variable `x` to a **floating point** number (these are treated and stored differently to integers in computing).

This in turn is different to:

```
x = True
```

where `True` is a **logical** or **boolean** datatype (something is `True` or `False`).

E1.1.7 Exercise

- in the code block below, create a variable called `my_var` and set it to some value (your choice of value, but be clear about the data type you intend)
- print the value of the variable to the screen, along with the data type.

```
# do the exercise here
```

We have so far seen four datatypes:

- integer (`int`): 32 bits long on most machines
- (double-precision) floating point (`float`): (64 bits long)
- Boolean (`bool`)
- string (`str`)

but we will come across more (and even create our own!) as we go through the course.

In each of these cases above, we have used the variable `x` to contain these different data types.

As we saw above, if you want to know what the data type of a variable is, you can use the method `type()`

```
print (type(1));
print (type(1.0));
print (type('one'));
print (type(True));
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
```

You can explicitly convert between data types, e.g.:

```
print ('int(1.1) = ',int(1.1))
print ('float(1) = ',float(1))
```

(continues on next page)

(continued from previous page)

```
print ('str(1) = ',str(1))
print ('bool(1) = ',bool(1))
```

```
int(1.1) =  1
float(1) =  1.0
str(1) =  1
bool(1) =  True
```

but only when it makes sense:

```
print ("converting the string '1' to an integer makes sense:",int('1'))
```

```
converting the string '1' to an integer makes sense: 1
```

```
print ("converting the string 'one' to an integer doesn't:",int('one'))
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-74-11bdc0c0e878> in <module>()
----> 1 print ("converting the string 'one' to an integer doesn't:",int('one'))


ValueError: invalid literal for int() with base 10: 'one'
```

When you get an error (such as above), you will need to learn to *read* the error message to work out what you did wrong.

E1.1.8 Exercise

- why did the statement abovce not work?
- type some other data conversions below that *do* work.

```
# do the exercise here
```

1.1.3 Arithmetic

Often we will want to do some arithmetic with numbers in a program, and we use the 'normal' (derived from C) operators for this.

Note the way this works for integers and floating point representations.

```
'''
    Some examples of arithmetic operations in Python

    Note how, if we mix float and int, the result is raised to float
    (as the more general form)
'''

print (10 + 100)      # int addition
print (10. - 100)     # float subtraction
print (1./2.)         # float division
```

(continues on next page)

(continued from previous page)

```
print (1/2)          # int division
print (10.*20.)      # float multiplication
print (2 ** 3.)      # float exponent

print (65%2)         # int remainder
print (65//2)        # floor operation
```

```
110
-90.0
0.5
0.5
200.0
8.0
1
32
```

E1.1.9 Exercise

- change the numbers in the examples above to make sure you understand these basic operations.
- try combining operations and use brackets `()` to check that that works as expected.
- see what happens when you add (i.e. use +) strings together

```
# do the exercise here
```

1.1.4 Assignment Operators

```
'''
    Assignment operators

    x = 3   assigns the value 3 to the variable x
    x += 2  adds 2 onto the value of x
            so is the same as x = x + 2
            similarly /=, *=, -=
    x %= 2  is the same as x = x % 2
    x **= 2 is the same as x = x ** 2
    x //= 2 is the same as x = x // 2

    A 'magic' trick
    ===============

    based on
    https://www.wikihow.com/Read-Someone%27s-Mind-With-Math-(Math-Trick)

    whatever you put as myNumber, the answer is 42

    Try this with integers or floating point numbers ...
'''

# pick a number
myNumber = 34.67

x = myNumber
```

(continues on next page)

(continued from previous page)

```
x *= 2

x *= 5

x /= myNumber

x -= 7

x += 39

# The answer will always be 42
print(x)
```

```
42.0
```

E1.1.10 Exercise

- change the number assigned to `myNumber` and check if `42` is still returned
- copy and edit the code to print the value of `x` each time you change it, and add comments explaining what is happening for each line of code. This should allow you to follow more carefully what has happened with the arithmetic and also to simplify the code (use fewer statements to achieve the same thing).

```
# do the exercise here
```

1.1.5 Logical Operators

Logical operators combine boolean variables. Recall from above:

```
print (type(True),type(False));
```

```
<class 'bool'> <class 'bool'>
```

The three main logical operators you will use are:

```
not, and, or
```

The impact of the `not` opeartor should be straightforward to understand, though we can first write it in a 'truth table':

A	not A
T	F
F	T

```
print('not True is',not True)
print('not False is',not False)
```

```
not True is False
not False is True
```

E1.1.11 Exercise:

- write a statement to set a variable `x` to `True` and print the value of `x` and `not x`
- what does `not not x` give? Make sure you understand why

```
# do the exercise here
```

The operators `and` and `or` should also be quite straightforward to understand: they have the same meaning as in normal english. Note that `or` is 'inclusive' (so, read `A or B` as 'either A or B or both of them').

```
print ('True and True is',True and True)
print ('True and False is',True and False)
print ('False and True is',False and True)
print ('False and False is',False and False)
```

```
True and True is True
True and False is False
False and True is False
False and False is False
```

So, `A and B` is `True`, if and only if both `A` is `True` and `B` is `True`. Otherwise, it is `False`

We can represent this in a 'truth table':

A	B	A and B
T	T	T
T	F	F
F	T	F
F	F	F

E1.1.12 Exercise:

- draw a truth table *on some paper*, label the columns `A`, `B` and `A and B` and fill in the columns `A` and `B` as above
- without looking at the example above, write the value of `A and B` in the third column.
- draw another truth table *on some paper*, label the columns `A`, `B` and `A and B` and fill in the columns `A` and `B` as above
- write the value of `A or B` in the third column.

If you are unsure, test the response using code, below.

```
# do the testing here e.g.
print (True or False)
```

```
True
```

E1.1.13 Exercise

- Copy the following truth table onto paper and fill in the final column:

A	B	C	((A and B) or C)
T	T	T	
T	T	F	
T	F	T	
T	F	F	
F	T	T	
F	T	F	
F	F	T	
F	F	F	

- Try some other compound statements

If you are unsure, or to check your answers, test the response using code, below.

```
# do the testing here e.g.
print ((True and False) or True)
```

```
True
```

1.1.6 Comparison Operators and `if`

A comparison operator 'compares' two terms (e.g. variables) and returns a boolean data type (`True` or `False`).

For example, to see if the value of some variable `a` is 'the same value as' ('equivalent to') the value of some variable `b`, we use the equivalence operator (==). To test for non equivalence, we use the not equivalent operator `!=` (read the `!` as 'not'):

```
a = 100
b = 10

# Note the use of \n and \t in here
#
print ('a is',a,'and\nb is',b,'\n')
print ('\ta is equivalent to b?',a == b)
```

```
a is 100 and
b is 10

    a is equivalent to b? False
```

E1.1.14 Exercise

- copy the code above and change the values (or type) of the variables `a` and `b` to test their equivalence.
- what does the `\t` in the print statement do?
- add a `print` statement to your code that tests for 'non equivalence'
- write some code to see if `(a or b)` is equivalent to `(b or a)` or not

```
# do the exercise here
```

A full set of comparison operators is:

2.8.4 symbol meaning

== is equivalent to != is not equivalent to > greater than >= greater than or equal to < less than <= less than or equal to ====== ==

so that, for example:

```
# Comparison examples

# is one plus one list identical to two list?
print ([1 + 1] is [2])

```

(continues on next page)

(continued from previous page)

```
# is one plus one list equal to two list?
print ([1 + 1] == [2])

# is one less than or equal to 0.999?
print (1 <= 0.999)

# is one plus one not equal to two?
print (1 + 1 != 2)

# note the use of single quotes inside a double quoted string here
# is 'more' greater than 'less'?
print ("more" > "less")

# "is 100 less than 2?"
print (100 < 2)
```

```
False
True
False
False
True
False
```

Aside on string comparisons

In the case of string comparisons, the ASCII codes of the string characters are compared. So for example the statement "more" > "less" returns True.

Here, the comparison is effectively

```
m > l
```

Since `m` comes after `l` in the alphabet, the ASCII code for `m` (109) is greater than the ASCII code for `l` (108) (see http://www.asciitable.com) so

```
109 > 108
```

returns True. Note that ASCII capital letters come before the lower case letters.

In practice, we mainly avoid string comparisons (other than to confirm equivalence). So there is little direct use of string comparisons other than ==. It is useful to know how this works however, in case it crops up or happens 'by accident'. It is also worth understanding what ASCII codes are.

Conditional test

One common use of comparisons is for program control, using an `if` statement:

```
if condition1 is True:
    doit1()
elif condition2 is True:
    doit2()
else:
    doit3()
```

where `is` compares identity. This allows us to run blocks of code (e.g. the method `doit1()`) only under a particular condition (or set of conditions).

In Python, the statement(s) we run on condition (here `doit1()` etc.) are *indented*.

The indent can be one or more spaces or a `<tab>` character, the choice is up to the programmer. However, it **must be consistent**.

```
test = [1+1]
print('test is {}'.format(test))

# initialise retval
retval = None

# conduct some tests, and set the
# variable retval to True if we pass
# any test

if test is [2]:
    retval = True
    print('passed test 1: "if test is [2]"')
elif test == [2]:
    retval = True
    print('passed test 2: "if test == [2]"')
else:
    retval = False
    print('failed both tests')

print('retval is',retval)
```

```
test is [2]
passed test 2: "if test == [2]"
retval is True
```

E1.1.15 Exercise

- copy the example above, and change it to use other examples from the 'Comparison examples' code block. Change the value of `test` to get different responses and make notes as to why you get the result you do.
- try out some more complicated conditions, e.g. multipler tests, combined with an `and` operator.

```
# do the exercise here
```

In this section, you have had an introduction to the Python programming language, running in a ``jupyter notebook` <http://jupyter.org>`__ environment.

You have seen how to write comments in code, how to form `print` statements and basic concepts of variables, values, and data types. You have seen how to maniputae data with arithmetic and assignment operators, as well as the basics in dealing with logic and tests returning logical values.

1.2 Text and looping

In Python, collections of characters (`a`, `b`, `1`, `...`) are called strings. Strings and characters are input by surrounding the relevant text in either double (`"`) or single (`'`) quotes. There are a number of special characters that can be encoded in a string provided they're "escaped". For example, some we have come across are:

- `\n`: the carriage return
- `\t`: a tabulator

```
print ("I'm a happy string")
print ('I\'m a happy string') # the apostrophe has been escaped as not to be confused␣
↪by end of string
```

(continues on next page)

(continued from previous page)

```
print ("\tI'm a happy string")
print ("I'm\na\nhappy\nstring")
```

```
I'm a happy string
I'm a happy string
    I'm a happy string
I'm
a
happy
string
```

We can do a number of things with strings, which are very useful. These so-called string methods are defined on all strings by Python by default, and can be used with every string. For one, we can concatenate strings using the + symbol as we saw above.

1.2.1 `len`

Gives the length of the string as number of characters:

```
t = ''
print ('the length of',t,'is',len(t))


s = "Hello" + "there" + "everyone"

print ('the length of',s,'is',len(s))
```

```
the length of  is 0
the length of Hellothereeveryone is 18
```

Exercise E1.2.1

- what does a zero-length string look like?
- The `Hello there everyone` example above has no spaces between the words. Copy the code to the block below and modify it to have spaces.
- confirm that you get the expected increase in length.

```
# do exercise here
```

1.2.2 `for ... in ...` and `enumerate`

Very commonly, we need to iterate or 'loop' over some set of items.

The basic stucture for doing this (in Python, and many other languages) is `for item in group:`, where `item` is the name of some variable and `group` is a set of values.

The loop is run so that `item` takes on the first value in `group`, then the second, etc.

```
# for loop
group = [4,3,2,1]

for item in group:
```

(continues on next page)

(continued from previous page)

```
    '''print counter in loop'''
    print(item)

print ('blast off!')
```

```
4
3
2
1
blast off!
```

The `group` in this example is the list of integer numbers `[4,3,2,1]`. A `list` is a group of comma-separated items contained in square brackets `[]`.

In Python, the statement(s) we run whilst looping (here `print(item)`) are *indented*.

The indent can be one or more spaces or a `<tab>` character, the choice is up to the programmer. However, it **must be consistent**.

Exercise 1.2.1

- generate a list of strings called `group` with the names of (some of) the items in your pocket or bag (or make some up!)
- set up a `for` loop to go through and print each item

```
# do exercise here
```

Quite often, we want to keep track of the 'index' of the item in the loop (the 'item number').

One way to do this would be to use a variable (called `count` here).

Before we enter the loop, we initialise the value to zero.

```
# for loop
group = ['hat','dog','keys']

# initialise a variable count
count = 0

for item in group:
    '''print counter in loop'''

    print('item',count,'is',item)

    # add 1 onto count
    count += 1
```

```
item 0 is hat
item 1 is dog
item 2 is keys
```

Exercise 1.2.2

- copy the code above, and check to see if the value of `count` at the end of the loop is the same as the length of the list. Why should this be so?
- change the code so that the counting starts at 1, rather than 0.

```
# do exercise here
```

Since counting in loops is a common task, we can use the built in method `enumerate()` to achieve the same thing as above. The syntax is then:

```
# for loop
group = ['hat','dog','keys']

for count,item in enumerate(group):
    '''print counter in loop'''
    print('item',count,'is',item)
```

```
item 0 is hat
item 1 is dog
item 2 is keys
```

Exercise 1.2.3

- copy the code above, and check to see if the value of `count` at the end of the loop is the same as the length of the list.
- change the code so that the printed count starts at 1, rather than 0.

Hint: how can you make it print `count+1` rather than `count`?

```
# do exercise here
```

1.2.3 `slice`

A string can be thought of as an ordered 'array' of characters.

So, for example the string `hello` can be thought of as a construct containing `h` then `e`, `l`, `l`, and `o`.

We can index a string, so that e.g. `'hello'[0]` is `h`, `'hello'[1]` is `e` etc.

We have seen above the idea of the 'length' of a string. In this example, the length of the string `hello` is 5.

```
string = 'hello'

# length
slen = len(string)
print('length of {} is {}'.format(string,slen))

# select these indices
indices = 0,1,3

# loop over each item in indices
for index in indices:
    print('character {} of {} is {}'.format(index,string,string[index]))
```

```
length of hello is 5
character 0 of hello is h
character 1 of hello is e
character 3 of hello is l
```

Exercise E1.2.4

- copy the code above, and see what happens if you set a value in `indices` that is the value of length of the string. Why does it respond so?
- make the code robust to this issue, but using an `if` statement to test if `index` is in the required range.

```
# do exercise here
```

We can use the idea of a 'slice' to access particular elements within the string.

For a slice, we can specify:

- start index (0 is the first)
- stop index (not including this)
- skip (do every 'skip' character)

When specifying this as array access, this is given as, e.g.:

`array[start:stop:skip]`

- The default start is 0
- The default stop is the length of the array
- The default skip is 1

You can specify a slice with the default values by leaving the terms out:

`array[::2]`

would give values in the array `array` from 0 to the end, in steps of 2.

This idea is fundamental to array processing in Python. We will see later that the same mechanism applies to all ordered groups.

```
s = "Hello World"
print (s,len(s))

start = 0
stop  = 11
skip  = 2
print (s[start:stop:skip])

# use -ve numbers to specify from the end
# use None to take the default value

start = -3
stop  = None
skip  = 1
print (s[start:stop:skip])
```

```
Hello World 11
HloWrd
rld
```

Exercise E1.2.5

The example above allows us to access an individual character(s) of the array.

- copy the example above, and print the string starting from the default start value, up to the default stop value, in steps of `2`.
- write code to print out the 4^{th} letter (character) of the string `s`.

```
# do exercise here
```

1.2.4 `replace`

We can replace all occurrences of a string within a string by some other string. We can also replace a string by an empty string, thus in effect removing it:

```
print ("I'm a very happy string".replace("happy", "unhappy"))
```

```
I'm a very unhappy string
```

Exercise E1.2.6

- copy the statement above, and use the `replace` method to make it print out `"I'm a happy string"`.

Hint: you want to replace the string `very` with, effectively, nothing, i.e. a zero-length string.

```
# do exercise here
```

1.2.5 `find`

Quite often, we might want to find a string inside another string, and potentially give the location (as in characters from the start of the string) where this string occurs. We can use the `find` method, which will return either a `-1` if the string isn't found, or an integer giving the index of where the string starts (for the first time).

```
print ("I'm a very happy string".find("a"))
print ("I'm a very happy string".find("happy"))
```

```
4
11
```

Let's use the idea of `find()` to sort out a messy table of data that we get from a web page.

First, we need to import the package `requests` to access some information from a URL (from a web page). The data we get will be in html.

The data we will examine is a dataset of ENSO values for each month of the year from January 1950 to present, made available by NOAA/

If you visit you will see the data table we are interested in. So, how do we 'grab' this?

The URL points to html code. When you display this in a browser, it is rendered appropriately.

If you access the html directly, you will get the following:

```
# Web scraping example

import requests

url = "http://www.esrl.noaa.gov/psd/enso/mei/table.html"

# This line will pull the URL data as a string
txt = requests.get(url).text

```

(continues on next page)

(continued from previous page)

```
# show the first 1000 characters (see 'slice' above: this is the same as␣
↪[None:1000:None])
print(txt[:1000])
```

```
<html>
<head><title>MEI timeseries from Dec/Jan 1940/50 up to the present</title></head>
<body>
<pre>
MEI Index (updated: 7 September 2018)

Bimonthly MEI values (in 1/1000 of standard deviations), starting with Dec1949/
↪Jan1950, thru last
month.  More information on the MEI can be found on the <a href="mei.html">MEI␣
↪homepage</a>.
Missing values are left blank.  Note that values can still change with each monthly␣
↪update, even
though such changes are typically smaller than +/-0.1.  All values are normalized for␣
↪each bimonthly
season so that the 44 values from 1950 to 1993 have an average of zero and a standard␣
↪deviation of "1".
Responses to 'FAQs' can be found below this table:

YEAR    DECJAN  JANFEB  FEBMAR  MARAPR  APRMAY  MAYJUN  JUNJUL  JULAUG  AUGSEP ␣
↪SEPOCT  OCTNOV  NOVDEC
1950        -1.03   -1.133  -1.283  -1.071  -1.434  -1.412  -1.269  -1.042  -.597   -.
↪406   -1.138  -1.235
1951        -1.049  -1.152  -1.178  -.511   -.374   .288    .679    .818    .773    .
↪768    .726    .504
1952        .433    .138    .071    .224    -.307   -.756   -.305   -.374
```

We notice the presence of html codes in the text string (e.g. `<html>`, `<pre>`). There are particular packages for neatly parsing html (scraping information from web pages), one of the most common being BeautifulSoup. This will tend to be more useful if the html is well fomatted, and the data contained in `<table>` sections, or similar structures. Here, we just have a block of text in the `<pre>` section.

If we want to *just* access the dataset here then, we might notice that the data we want to access starts when we see the string `YEAR`.

We can use `find()` to discover the index of this in the string:

```
start = txt.find('YEAR')

print('start of useful data at index {}\n---------------------------------'.
↪format(start))
print(txt[start:start+1000])
```

```
start of useful data at index 689
---------------------------------
YEAR    DECJAN  JANFEB  FEBMAR  MARAPR  APRMAY  MAYJUN  JUNJUL  JULAUG  AUGSEP ␣
↪SEPOCT  OCTNOV  NOVDEC
1950        -1.03   -1.133  -1.283  -1.071  -1.434  -1.412  -1.269  -1.042  -.597   -.
↪406   -1.138  -1.235
1951        -1.049  -1.152  -1.178  -.511   -.374   .288    .679    .818    .773    .
↪768    .726    .504
1952        .433    .138    .071    .224    -.307   -.756   -.305   -.374   .347    .
↪306    -.328   -.098
```

(continues on next page)

(continued from previous page)

```
1953        .044    .401    .277    .687    .756    .191    .382    .209    .527    .
↪124    .099    .351
1954        -.036   -.027   .154    -.616   -1.465  -1.558  -1.355  -1.456  -1.138  -
↪1.32   -1.113  -1.088
1955        -.74    -.669   -1.117  -1.621  -1.653  -2.247  -1.976  -2.05   -1.803  -
↪1.725  -1.813  -1.846
1956        -1.408  -1.275  -1.371  -1.216  -1.304  -1.523  -1.244  -1.118  -1.327  -
↪1.461  -1.014  -.993
1957        -.915   -.348   .108    .383    .813    .73     .926    1.132   1.158   1.
↪114   1.167   1.268
1958        1.473   1.454   1.313   .991    .673    .812    .7      .421    .209    .
↪237    .501    .691
1959        .553    .81     .502    .202    -.025   -.062   -.112   .111    .126    -.
↪038   -.151   -.247
1960        -.287   -.253   -.082   .007    -.322   -.287   -.318   -.25    -.439   -.
↪332   -.308   -.39
1961        -.15    -.235   -.073   .017    -.302   -.185   -.208   -.3     -.271   -.
↪51    -.416
```

If we look again at the web page http://www.esrl.noaa.gov/psd/enso/mei/table.html, we might notice that the end of the useful data is delimited by two newlines and the string `(1)`, i.e., as a string `\n\n(1)`. So we should be able to use `find()` again to get the location of the end of the data (i.e. `stop`, in the sense of a slice).

Exercise 1.2.6

- use this observation to form a string called `data_table`, containing all of the useful data (i.e. `txt[start:stop]`).
- print the string `data_table`.

```
# do exercise here
```

This exercise is a very good example of web scraping. Web scraping is often rather messy (you have to work out some 'key' to reliably delimit the information you want) but can be extreemely valuable for accessing datasets that are not cleanly presented. We have only gonbe part of the way to extracting a useful dataset here, because the dataset we are interested in (the ENSO data) are still represented as a string, whereas we really want them to be a set of floating point numbers. We will deal with this later.

1.2.5 `split` and `splitlines`

The first 'line' of `'should contain the 'header' information, i.e. the title of the data columns (`YEAR, DECJAN etc.`). We want to separate the header from the numbers in the data table, so we want to 'split' the string called`data_table' into a header string and data string.

One approach to this would be split the string into 'lines' of text (rather than one block). Effectively that means splitting into multiple strings whenever we hit a `\n` character. Rather than do that explicitly, we use the `splitlines()` method:

```
import requests
url = "http://www.esrl.noaa.gov/psd/enso/mei/table.html"
txt = requests.get(url).text

# copy the useful data
start = txt.find('YEAR')
```

(continues on next page)

(continued from previous page)

```
stop  = txt.find('\n\n(1)')
data_table = txt[start:stop]

# split into a list of strings
data_lines = data_table.splitlines()

# tell me something useful
print(type(data_lines),len(data_lines))

# loop over some examples
for i in 0,1,len(data_lines)-1:
    print('line {} {}\n\t{}'.format(i,type(data_lines[i]),data_lines[i]))
```

```
<class 'list'> 70
line 0 <class 'str'>
    YEAR    DECJAN   JANFEB   FEBMAR   MARAPR   APRMAY   MAYJUN   JUNJUL   JULAUG   AUGSEP ␣
↪SEPOCT   OCTNOV   NOVDEC
line 1 <class 'str'>
    1950    -1.03    -1.133   -1.283   -1.071   -1.434   -1.412   -1.269   -1.042   -.597    -.
↪406   -1.138   -1.235
line 69 <class 'str'>
    2018    -.623    -.731    -.502    -.432    .465     .469     .076     .132
```

This splits each 'line' of text into an entry in a `list`, so that the header data is now given in the first entry (`data_lines[0]`) and the lines containinmg data, after that.

From the print out above, we notice that the final 'data line' (index `-1`) is shorter than (has fewer entries than) the other lines. This is because we are only part way through this year!.

In 'real' datasets, we quite often have 'messy' lines of data such as this (or data missing for other reasons). How you want to deal with the 'messy bits' depends on the sort of analysis you want to do.

One option (the simplest) would be to simply remove the last line (ignore this year's data):

```
header = data_lines[0]

# select the data block as being from entry 1 to -1
# so, **not including the last row**
data = data_lines[1:-1]

print('header:',header)

for i in 0,1,len(data)-1:
    print('line {} {}\n\t{}'.format(i,type(data[i]),data[i]))
```

```
header: YEAR    DECJAN   JANFEB   FEBMAR   MARAPR   APRMAY   MAYJUN   JUNJUL   JULAUG ␣
↪AUGSEP   SEPOCT   OCTNOV   NOVDEC
line 0 <class 'str'>
    1950    -1.03    -1.133   -1.283   -1.071   -1.434   -1.412   -1.269   -1.042   -.597    -.
↪406   -1.138   -1.235
line 1 <class 'str'>
    1951    -1.049   -1.152   -1.178   -.511    -.374    .288     .679     .818     .773     .
↪768    .726     .504
line 67 <class 'str'>
    2017    -.052    -.043    -.08     .744     1.445    1.039    .456     .009     -.449    -.
↪551   -.277    -.576
```

Exercise 1.2.7

- copy the code from above and explore the response using line indices `-1` and `-2`.

```
# do exercise here
```

If we want to manipulate or plot the information contained in this (the numbers), we need to convert each of the string representations to a floating point number, e.g. the number `-1.03` rather than the string `'-1.03'`.

Each entry in the list `data` is a string, as we saw above.

We can split an individual string (such as `data[0]` into a list of strings, using the string method `split()`. By default, this splits on 'white space' (i.e. spaces or tab characters), so, e.g.:

```
line = data[0].split()
print(data[0])
print(line,len(line))
```

```
1950        -1.03   -1.133  -1.283  -1.071  -1.434  -1.412  -1.269  -1.042  -.597   -.
↪406   -1.138  -1.235
['1950', '-1.03', '-1.133', '-1.283', '-1.071', '-1.434', '-1.412', '-1.269', '-1.042
↪', '-.597', '-.406', '-1.138', '-1.235'] 13
```

So, we have split the long string into 13 strings in a list.

We want to generate a new list with 13 corresponding floating point values:

```
# split the line on whitespace
line = data[0].split()

# make a new list of the same length
# by copying the variable line
float_data = line.copy()

for index,line_data in enumerate(line):
    # insert the cast float into the list
    # in the right order (use index)
    float_data[index] = float(line_data)

# this is the string list
print(line)

# this is the float list
print(float_data)
```

```
['1950', '-1.03', '-1.133', '-1.283', '-1.071', '-1.434', '-1.412', '-1.269', '-1.042
↪', '-.597', '-.406', '-1.138', '-1.235']
[1950.0, -1.03, -1.133, -1.283, -1.071, -1.434, -1.412, -1.269, -1.042, -0.597, -0.
↪406, -1.138, -1.235]
```

Exercise 1.2.8

- set a variable to be the string `"2, 3, 5, 7, 11, 13, 17, 19, 23, 29"`
- use the approach above to generate a **list of integers** of the first 10 prime numbers.
- print the list with syntax of the pattern of 'prime number 3 is 7'

Make sure you convert each prime number to an integer, rather than leaving it as a string!

Hint: We can still use the method `split()` to do split the string into a list of strings, but this time the separator is a comma, rather than whitespace.

```
# do exercise here
pstring = "2, 3, 5, 7, 11, 13, 17, 19, 23, 29"
```

Normally, we wouldn't go to the trouble of first copying the list.

Instead, **where the contents of the loop are simple** (e.g. a single statement) we would use a different way of using a `for` loop, called an **implicit loop**.

In this case:

```
for item in group:
    doit(group)
```

becomes:

```
[doit(group) for item in group]
```

with the additional feature that everything returned by `doit(group)` for each item of `group` is put in a list.

```
# split the line on whitespace

# implicit for loop
float_data = [float(line_data) for line_data in data[0].split()]

# this is the string list
print(line)
# this is the float list
print(float_data)
```

```
2010        1.066   1.526   1.462   .978    .658    -.228   -1.103  -1.671  -1.86   -
↪1.888  -1.472  -1.558
[1950.0, -1.03, -1.133, -1.283, -1.071, -1.434, -1.412, -1.269, -1.042, -0.597, -0.
↪406, -1.138, -1.235]
```

The statement:

```
float_data = [float(line_data) for line_data in line]
```

is much more Pythonic than the code above. It is simple, elegant and neat.

We can *nest* for statements, i.e. put one for loop inside another. This allows us to treat data of multiple dimensions.

In the examples above, we converted only the data in `data[0]` to a list of floating point numbers. If we wanted to process *all* lines of data, we would have to loop over them as well, in an 'outer' loop.

```
# use a step of 10 for illustration purposes
# to save space when printing

step = 10

for index,line in enumerate(data_table.splitlines()[1:-1:step]):
    # convert each line to list of floats
    float_data = [float(line_data) for line_data in line.split()]
    print('line {} is {}'.format(index*step,float_data))
```

```
line 0 is [1950.0, -1.03, -1.133, -1.283, -1.071, -1.434, -1.412, -1.269, -1.042, -0.
↪597, -0.406, -1.138, -1.235]
line 10 is [1960.0, -0.287, -0.253, -0.082, 0.007, -0.322, -0.287, -0.318, -0.25, -0.
↪439, -0.332, -0.308, -0.39]
line 20 is [1970.0, 0.38, 0.432, 0.228, 0.014, -0.099, -0.636, -1.055, -1.007, -1.226,
↪ -1.068, -1.063, -1.203]
line 30 is [1980.0, 0.677, 0.601, 0.684, 0.912, 0.958, 0.911, 0.769, 0.329, 0.302, 0.
↪223, 0.27, 0.111]
line 40 is [1990.0, 0.243, 0.573, 0.951, 0.46, 0.652, 0.511, 0.147, 0.127, 0.398, 0.
↪303, 0.4, 0.362]
line 50 is [2000.0, -1.122, -1.189, -1.09, -0.397, 0.251, 0.001, -0.159, -0.145, -0.
↪21, -0.367, -0.701, -0.55]
line 60 is [2010.0, 1.066, 1.526, 1.462, 0.978, 0.658, -0.228, -1.103, -1.671, -1.86,␣
↪-1.888, -1.472, -1.558]
```

Note that whilst we have calculated `float_data` in the loop for each line, it gets over-written with each new line as things stand.

We can do the same thing, and generate a list of the responses more neatly, using an implicit loop inside another implicit loop:

```
all_float_data = [[float(line_data) for line_data in line.split()] for line in data_
↪table.splitlines()[1:-1]]
```

The variable `all_float_data` is now a sort of 'two dimensional' list, within which we can refer to individual items as e.g. `all_float_data[10][3]` for row `10`, column `3`.

Let's use this idea to print out column 0 of each row (containing the `YEAR` data). We will use the method `range(nrows)` that (implicitly) generates a list `[0,1,2,3, ..., nrows-1]`.

Notice the use of `end=' '` in the `print` statement. This replaces the usual newline by whatever is specified by the keyword `end`. Note also that we have used `{:.0f}` to specify the format term. This indicates that the term is to be printed as a floating point number (the `f`) with zero numbers after the decimal point (`.0`)

```
nrows = len(all_float_data)
i = 0

print('column {} of the data gives:\n'.format(i))
for row in range(nrows):
    print('{:.0f}'.format(all_float_data[row][i]),end=' ')
```

```
column 0 of the data gives:

1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965 1966␣
↪1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980 1981 1982␣
↪1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998␣
↪1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014␣
↪2015 2016 2017
```

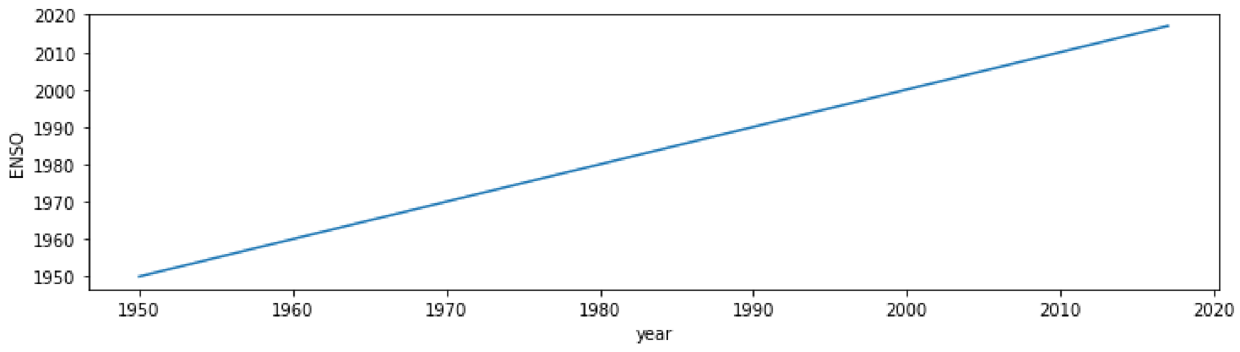
Exercise 1.2.9

- use an implicit loop to create a list of ENSO values in a variable `enso` for the years 1950 up to last year for the period `DECJAN`.
- produce a plot of ENSO for `DECJAN` as a function of year (see below on how to do that).

Hint: check which column in the header is `DECJAN`. To start you off on this, we give you the implicit loop code for extracting the column containing the `YEAR` data (column 0). We also give you the code to achieve the plotting.

```
# do exercise here

# generate a list called years of column 0 data
years = [all_float_data[row][0] for row in range(nrows)]

# you need to put the enso data in here!
# this is put in as a dummy that should plot a straight line!
enso = years.copy()

# for plotting
import pylab as plt
%matplotlib inline

#
plt.figure(0,figsize=(12,3))
plt.plot(years,enso)
plt.xlabel('year')
plt.ylabel('ENSO')
```

```
Text(0,0.5,'ENSO')
```



1.2.6 Summary

In section 1.2 you have been introduced to text representation in Python, as strings (type `str`), and shown that this sort of variable can be thought of an an 'array', and that it has a length attribute that can be accessed with `len()`.

Other useful string manipulation methods you were introduced to are: `replace()`, `find()`, `split()` and `splitlines()`, though of course there are many more.

In an 'array', we can use an index to refer to a particular item (e.g. index 0 for the first item, 1 for the second, -1 for the last). We can use this idea to manipulate strings.

In a more general sense, we can take a 'slice' of an array, with the syntax `[start:stop:skip]` giving access to a regularly spaced part of an array. We can use this, for example, to print out every 10th value (`skip=10`).

You were also introduced to the idea of looping control structures, using a `for ... in ...:` statement, and the equivalent implicit form. This introduced the idea of indented code blocks and (related) nested structures (loops within loops).

In passing, you have also been shown how to pull html data from a URL (scraping) using the ``requests` <http://docs.python-requests.org/en/master/>'__ package, and also how to produce a simple data plot, using ``pylab` <https://matplotlib.org/index.html>'__.

1.3. Groups of things

Very often, we will want to group items together. There are several main mechanisms for doing this in Python, known as:

- string e.g. `hello`
- tuple, e.g. `(1, 2, 3)`
- list, e.g. `[1, 2, 3]`
- numpy array e.g. `np.array([1, 2, 3])`

A slightly different form of group is a dictionary:

- dict, e.g. `{1:'one', 2:'two', 3:'three'}`

You will notice that each of the grouping structures tuple, list and dict use a different form of bracket. The numpy array is fundamental to much work that we will do later.

We have dealt with the idea of a string as an ordered collection in the material above, so will deal with the others here.

We noted the concept of length (`len()`), that elements of the ordered collection could be accessed via an index, and came across the concept of a slice. All of these same ideas apply to the first set of groups (string, tuple, list, numpy array) as they are all ordered collections.

A dictionary is not (by default) ordered, however, so indices have no role. Instead, we use 'keys'.

1.3.1 `tuple`

A tuple is a group of items separated by commas. In the case of a tuple, the brackets are optional. You can have a group of differnt types in a tuple (e.g. int, int, str, bool)

```
# load into the tuple
t = (1, 2, 'three', False)

# unload from the tuple
a,b,c,d = t

print(t)
print(a,b,c,d)
```

```
(1, 2, 'three', False)
1 2 three False
```

If there is only one element in a tuple, you must put a comma , at the end, otherwise it is not interpreted as a tuple:

```
t = (1)
print (t,type(t))
t = (1,)
print (t,type(t))
```

```
1 <class 'int'>
(1,) <class 'tuple'>
```

You can have an empty tuple though:

```
t = ()
print (t,type(t))
```

```
() <class 'tuple'>
```

E1.3.1 Exercise

- create a tuple called t that contains the integers 1 to 5 inclusive
- print out the value of t
- use the tuple to set variables a1,a2,a3,a4,a5

```
# do exercise here
```

1.3.2 `list`

A `list` is similar to a `tuple`. One main difference is that you can change individual elements in a list but not in a tuple. To convert between a list and tuple, use the 'casting' methods `list()` and `tuple()`:

```
# a tuple
t0 = (1,2,3)

# cast to a list
l = list(t0)

# cast to a tuple
t = tuple(l)

print('type of {} is {}'.format(t,type(t)))
print('type of {} is {}'.format(l,type(l)))
```

```
type of (1, 2, 3) is <class 'tuple'>
type of [1, 2, 3] is <class 'list'>
```

You can concatenate (join) lists or tuples with the + operator:

```
l0 = [1,2,3]
l1 = [4,5,6]

l = l0 + l1
print ('joint list:',l)
```

```
joint list: [1, 2, 3, 4, 5, 6]
```

E1.3.2 Exercise * copy the code from the cell above, but instead of lists, use tuples * loop over each element in the tuple and print out the data type and value of the element

Hint: use a `for ... in ...` construct.

```
# do exercise here
```

A common method associated with lists or tuples is: * `index()`

Some useful methods that will operate on lists and tuples are: * `len()` * `sort()` * `min()`,`max()`

```
l0 = (2,8,4,32,16)

# print the index of the item integer 4
```

(continues on next page)

(continued from previous page)

```
# in the tuple / list

item_number = 4

# Note the dot . here
# as index is a method of the class list
ind  = l0.index(item_number)

# notice that this is different
# as len() is not a list method, but
# does operatate on lists/tuples
# Note: do not use len as a variable name!
llen = len(l0)

# note the use of integers in the braces e.g. {0}
# rather than empty braces as before. This allows us to
# refer to particular items in the format argument list
print('the index of {0} in {1} is {2}'.format(item_number,l0,ind))
print('the length of the {0} {1} is {2}'.format(type(l0),l0,llen))
```

```
the index of 4 in (2, 8, 4, 32, 16) is 2
the length of the <class 'tuple'> (2, 8, 4, 32, 16) is 5
```

E1.3.3 Exercise

- copy the code to the block below, and test that this works with lists, as well as tuples
- find the index of the integer 16 in the tuple/list
- what is the index of the first item?
- what is the length of the tuple/list?
- what is the index of the last item?

```
# do exercise here
```

A list has a much richer set of methods than a tuple. This is because we can add or remove list items (but not tuple).

- `insert(i,j)` : insert `j` beore item `i` in the list
- `append(j)` : append `j` to the end of the list
- `sort()` : sort the list

This shows that tuples and lists are 'ordered' (i.e. they maintain the order they are loaded in) so that indiviual elements may be accessed through an 'index'. The index values start at 0 as we saw above. The index of the last element in a list/tuple is the length of the group, minus 1. This can also be referred to an index `-1`.

```
l0 = [2,8,4,32,16]

# insert 64 at the begining (before item 0)
# Note that this inserts 'in place'
# i.e. the list is changed by calling this
l0.insert(0,64)


# insert 128 *before* the last item (item -1)
l0.insert(-1,128)
```

(continues on next page)

(continued from previous page)

```
# append 256 on the end
l0.append(256)

# copy the list
# and sort the copy
# Note the use of the copy() method here
# to create a copy
l1 = l0.copy()

# Note that this sorts 'in place'
# i.e. the list is changed by calling this
l1.sort()

print('the list {0} once sorted is {1}'.format(l0,l1))
```

```
the list [64, 2, 8, 4, 32, 128, 16, 256] once sorted is [2, 4, 8, 16, 32, 64, 128,␣
↪256]
```

E1.3.4 Exercise

- copy the above code and try out some different locations for inserting values (e.g. what does index `-2` mean?)
- what happens if you take off the `.copy()` statement in the line `l1 = l0.copy()`, i.e. just use `l1 = l0`? Why is this?

```
# do exercise here
```

1.3.3 `np.array`

An array is a group of objects of the same type. Because they are of the same type, they can be stored efficiently in compter memory, and also accessed efficiently.

Whilst there are different ways of forming arrays, the most common is to use numpy arrays, using the package `numpy`. To use this, we must first import the package into the current workspace. We do this with the `import` method. Using the optional `as` statement allows us to use a shorter (or more suitable) name for the package. We will generally call numpy `np`, so we use:

```
import numpy as np
```

to import ('load') the numpy package.

Often, we will read data from a file/URL as we did above for the ENSO dataset. In that case, we had to step through each item to convert from string form to floating point number.

This sort of thing is much more simply done using methods associated with numpy arrays.

A particularly useful numpy method is `np.loadtxt(file)` that loads an ASCII table of data straight into a numpy array.

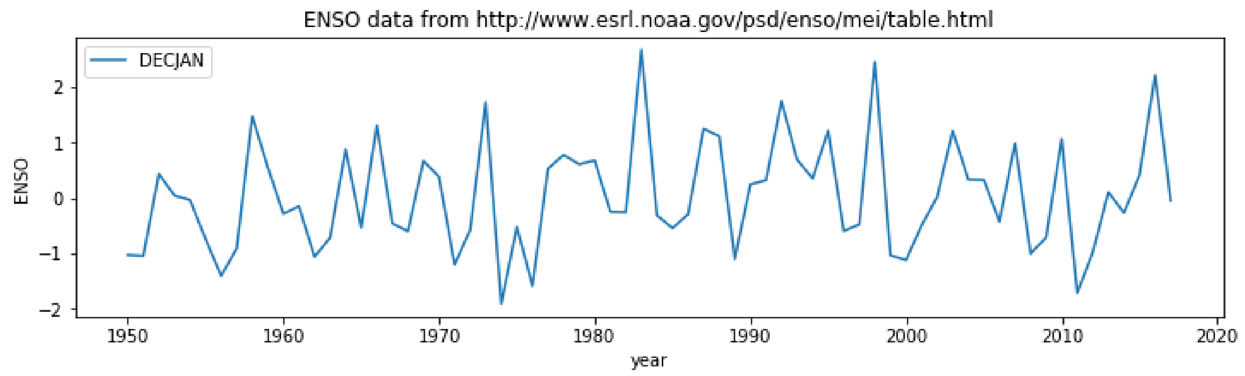
Whilst this is designed to load data from a file, we can use `io.StringIO()` from the `io` package to make data that we already have as a string seem to `np.loadtxt` as if it were a file. This is a useful 'trick' for using methods that expect data in a file. The `unpack=True` option makes sure the data array is composed the way we would expect it. The `usecols` option lets us select only those data columns we wish to read (0 and 1 here).

An alternative to `np.loadtxt()` is `np.genfromtxt()`. This has some additional features, such the `invalid_raise` flag. If this is set `False`, the loading is made somewhat tolerant to data errors (e.g. inconsistent number of columns). Further, we can explicitly set what will indicate `missing_values` in the input and what we would like to replace them with (`filling_values`) which can be useful for tidying up datasets.

```
import requests
import numpy as np
import io

# access dataset as above
url = "http://www.esrl.noaa.gov/psd/enso/mei/table.html"
txt = requests.get(url).text

# copy the useful data
start_head = txt.find('YEAR')
start_data = txt.find('1950\t')
stop_data  = txt.find('2018\t')

# select a data column
data_column = 1

header = txt[start_head:start_data].split()
data = np.loadtxt(io.StringIO(txt[start_data:stop_data]),unpack=True,usecols=[0,data_
↪column])

# so data[0] is the year data
#    data[1] is the enso data for column data_column
# print some attributes of the data array

print('array type',type(data))
print('data type',data.dtype)
print('number of dimensions',data.ndim)
print('data shape',data.shape)
print('data size',data.size)

# for plotting
import pylab as plt
%matplotlib inline

#
plt.figure(0,figsize=(12,3))
plt.plot(data[0],data[1],label=header[data_column])
plt.xlabel('year')
plt.ylabel('ENSO')
plt.title('ENSO data from {0}'.format(url))
plt.legend(loc='best')
```

```
array type <class 'numpy.ndarray'>
data type float64
number of dimensions 2
data shape (2, 68)
data size 136
```

```
<matplotlib.legend.Legend at 0x11ea0dda0>
```



We saw in the example above that a numpy array (`<class 'numpy.ndarray'>`) has a set of attributes that include `shape`, `ndim`, `dtype` and `size` that we can use to query information about the array. We will learn morre about processing data with numpy arrays later in the course, but you should already see that they are a useful construct for manipulating multi-dimensional datasets.

Exercise 1.3.4

- copy the code from the block above and modify it to plot the ENSO data for the period `FEBMAR`. Check this by looking at the data in the original table.
- modify the code to produce a plot of *all* periods (so the graph should have 12 lines, correctly labelled)

Hint: You will need to consider what, if anything to set of `usecols` (what happends if you don't set `usecols`?) and provide a looping structure for the plotting.

```
# do exercise here
```

1.3.4 dict

The collections we have used so far have all been ordered. This means that we can refer to a particular element in the group by an index, e.g. `array[10]`.

A dictionary is not (by default) ordered. Instead of indices, we use 'keys' to refer to elements: each element has a key associated with it. It can be very useful for data organisation (e.g. databases) to have a key to refer to, rather than e.g. some arbitrary column number in a gridded dataset.

A dictionary is defined as a group in braces (curley brackets). For each elerment, we specify the key and then the value, separated by `:`.

```
a = {'one': 1, 'two': 2, 'three': 3}

# we then refer to the keys and values in the dict as:

print ('a:\n\t',a)
print ('a.keys():\n\t',a.keys())      # the keys
print ('a.values():\n\t',a.values()) # returns the values
print ('a.items():\n\t',a.items())    # returns a list of tuples
```

```
a:
     {'one': 1, 'two': 2, 'three': 3}
a.keys():
     dict_keys(['one', 'two', 'three'])
a.values():
```

(continues on next page)

(continued from previous page)

```
     dict_values([1, 2, 3])
a.items():
     dict_items([('one', 1), ('two', 2), ('three', 3)])
```

Because dictionaries are not ordered, we cannot guarantee the order they will come out in a `for` loop, but we will often use such a loop to iterate over the items in a dictionary.

```
for key,value in a.items():
    print(key,value)
```

```
one 1
two 2
three 3
```

We refer to specific items using the key e.g.:

```
print(a['one'])
```

```
1
```

You can add to a dictionary:

```
a.update({'four':4,'five':5})
print(a)

# or for a single value
a['six'] = 6
print(a)
```

```
{'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5}
{'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5, 'six': 6}
```

Quite often, you find that you have the keys you want to use in a dictionary as a list or array, and the values in another list.

In such a case, we can use the method `zip(keys,values)` to load into the dictionary. For example:

```
values = [1,2,3,4]
keys = ['one','two','three','four']

a = dict(zip(keys,values))

print(a)
```

```
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

We will use this idea to make a dictionary of our ENSO dataset, using the items in the header for the keys. In this way, we obtain a more elegant representation of the dataset, and can refer to items by names (keys) instead of column numbers.

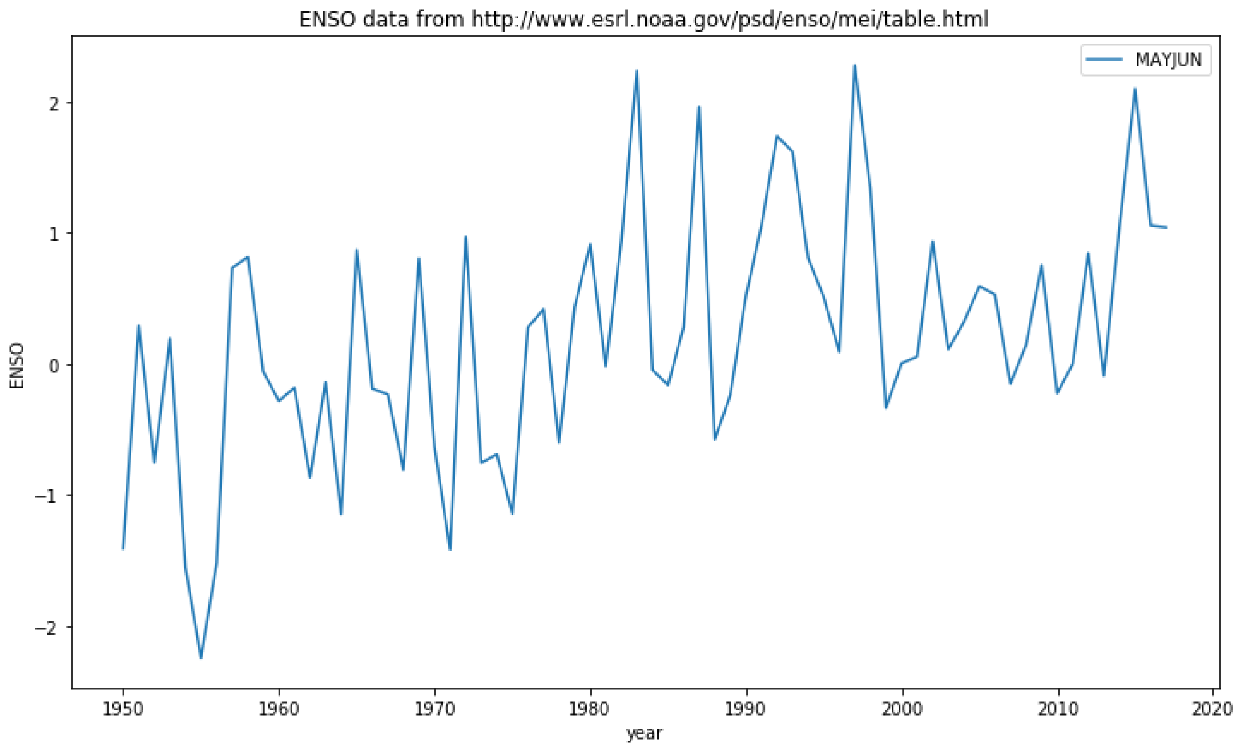
```
import requests
import numpy as np
import io

# access dataset as above
```

(continues on next page)

(continued from previous page)

```
url = "http://www.esrl.noaa.gov/psd/enso/mei/table.html"
txt = requests.get(url).text

# copy the useful data
start_head = txt.find('YEAR')
start_data = txt.find('1950\t')
stop_data  = txt.find('2018\t')

header = txt[start_head:start_data].split()
data = np.loadtxt(io.StringIO(txt[start_data:stop_data]),unpack=True)

# use zip to load into a dictionary
data_dict = dict(zip(header, data))

key = 'MAYJUN'
# plot data
plt.figure(0,figsize=(12,7))
plt.title('ENSO data from {0}'.format(url))
plt.plot(data_dict['YEAR'],data_dict[key],label=key)
plt.xlabel('year')
plt.ylabel('ENSO')
plt.legend(loc='best')
```

```
<matplotlib.legend.Legend at 0x11fceb320>
```



Exercise 1.3.5

- copy the code above, and modify so that datasets for months `['MAYJUN','JUNJUL','JULAUG']` are plotted on the graph

Hint: use a for loop

```
# do exercise here
```

We can also usefully use a dictionary with a printing format statement. In that case, we refer directly to the key in ther format string. This can make printing statements much easier to read. We don;'t directly pass the dictionary to the `fortmat` staterment, but rather `**dict`, where `**dict` means "treat the key-value pairs in the dictionary as additional named arguments to this function call".

So, in the example:

```
import requests
import numpy as np
import io

# access dataset as above
url = "http://www.esrl.noaa.gov/psd/enso/mei/table.html"
txt = requests.get(url).text

# copy the useful data
start_head = txt.find('YEAR')
start_data = txt.find('1950\t')
stop_data  = txt.find('2018\t')

header = txt[start_head:start_data].split()
data = np.loadtxt(io.StringIO(txt[start_data:stop_data]),unpack=True)

# use zip to load into a dictionary
data_dict = dict(zip(header, data))
print(data_dict.keys())

# print the data for MAYJUN
print('data for MAYJUN: {MAYJUN}'.format(**data_dict))
```

```
dict_keys(['YEAR', 'DECJAN', 'JANFEB', 'FEBMAR', 'MARAPR', 'APRMAY', 'MAYJUN', 'JUNJUL
↪', 'JULAUG', 'AUGSEP', 'SEPOCT', 'OCTNOV', 'NOVDEC'])
data for MAYJUN: [-1.412e+00  2.880e-01 -7.560e-01  1.910e-01 -1.558e+00 -2.247e+00
 -1.523e+00  7.300e-01  8.120e-01 -6.200e-02 -2.870e-01 -1.850e-01
 -8.700e-01 -1.440e-01 -1.150e+00  8.670e-01 -1.930e-01 -2.360e-01
 -8.120e-01  8.010e-01 -6.360e-01 -1.420e+00  9.660e-01 -7.580e-01
 -6.940e-01 -1.148e+00  2.760e-01  4.140e-01 -6.050e-01  4.290e-01
  9.110e-01 -2.400e-02  9.300e-01  2.235e+00 -4.900e-02 -1.660e-01
  2.790e-01  1.958e+00 -5.820e-01 -2.420e-01  5.110e-01  1.051e+00
  1.735e+00  1.616e+00  8.030e-01  5.070e-01  8.700e-02  2.275e+00
  1.336e+00 -3.390e-01  1.000e-03  5.200e-02  9.320e-01  1.070e-01
  3.150e-01  5.890e-01  5.260e-01 -1.550e-01  1.420e-01  7.510e-01
 -2.280e-01 -3.000e-03  8.420e-01 -9.400e-02  1.046e+00  2.097e+00
  1.053e+00  1.039e+00]
```

The line `print('data for MAYJUN: {MAYJUN}'.format(**data_dict))` is equivalent to writing:

```
print('data for {MAYJUN}'.format(YEAR=data_dict[YEAR],DECJAN=data_dict[DECJAN], ...))
```

In this way, we use the keys in the dictionary as keywords to pass to a method.

Another useful example of such a use of a dictionary is in saving a numpy dataset to file.

If the data are numpy arrays in a dictionary as above, we can store the dataset using:

```
import requests
import numpy as np
import io

# access dataset as above
url = "http://www.esrl.noaa.gov/psd/enso/mei/table.html"
txt = requests.get(url).text

# copy the useful data
start_head = txt.find('YEAR')
start_data = txt.find('1950\t')
stop_data  = txt.find('2018\t')

header = txt[start_head:start_data].split()
data = np.loadtxt(io.StringIO(txt[start_data:stop_data]),unpack=True)

# use zip to load into a dictionary
data_dict = dict(zip(header, data))

filename = 'enso_mei.npz'

# save the dataset
np.savez_compressed(filename,**data_dict)
```

What we load from the file is a dictionary-like object `<class 'numpy.lib.npyio.NpzFile'>`.

If needed, we can cast this to a dictionary with `dict()`, but it is generally more efficient to keep the original type.

```
# load the dataset

filename = 'enso_mei.npz'

loaded_data = np.load(filename)

print(type(loaded_data))

# test they are the same using np.array_equal
for k in loaded_data.keys():
    print('\t',k,np.array_equal(data_dict[k], loaded_data[k]))
```

```
<class 'numpy.lib.npyio.NpzFile'>
	 YEAR True
	 DECJAN True
	 JANFEB True
	 FEBMAR True
	 MARAPR True
	 APRMAY True
	 MAYJUN True
	 JUNJUL True
	 JULAUG True
	 AUGSEP True
	 SEPOCT True
	 OCTNOV True
	 NOVDEC True
```

Exercise 1.3.6

- Using what you have learned above, access the Met Office data file (`https://www.metoffice.gov.`

uk/hadobs/hadukp/data/monthly/HadSEEP_monthly_qc.txt)[https://www.metoffice.gov.uk/hadobs/hadukp/data/monthly/HadSEEP_monthly_qc.txt] and create a 'data package' in a numpy.npz file that has keys of YEAR and each month in the year, with associated datasets of Monthly Southeast England precipitation (mm).

- confirm that tha data in your npz file is the same as in your original dictionary
- produce a plot of October rainfall using these data for the years 1900 onwards

```
# do exercise here
```

1.3.5 Summary

In this section, we have extended the types of data we might come across to include groups . We dealt with ordered groups of various types (tuple, list), and introduced the numpy package for numpy arrays (np.array). We saw dictionaries as collections with which we refer to individual items with a key.

We learned in the previous section how to pull apart a dataset presented as a string using loops and various using methods and to construct a useful dataset 'by hand' in a list or similar structure. It is useful, when learning to program, to know how to do this.

Here, we saw that packages such as numpy provide higher level routines that make reading data easier, and we would generally use these in practice. We saw how we can use zip() to help load a dataset from arrays into a dictionary, and also the value of using a dictionary representation when saving numpy files.

2.8.5 2. Manipulating and plotting data in Python: numpy, and matplotlib libraries With ANSWERS

Table of Contents

While Python has a rich set of modules and data types by default, for numerical computing you'll be using two main libraries that conform the backbone of the Python scientific stack. These libraries implement a great deal of functionality related to mathematical operations and efficient computations on large data volumes. These libraries are `numpy <http://numpy.org>'__ and `scipy <http://scipy.org>'__. numpy, which we will concentrate on in this section, deals with efficient arrays, similar to lists, that simplify many common processing operations. Of course, just doing calculations isn't much fun if you can't plot some results. To do this, we use the `matplotlib <http://matplotlib.org>'__ library.

But first, we'll see the concept of *functions*. . . .

2.1 Functions

A function is a collection of Python statements that do something (usually on some data). For example, you may want to convert from Fahrenheit to Centigrade. The conversion is

$$^{\circ}C = (^{\circ}F - 32) \cdot \frac{5}{9}$$

A Python function will have a name (and we hope that the name is self-explanatory as to what the function does), and a set of input parameters. In the case above, the function would look like this:

```
def fahrenheit_to_centigrade(deg_fahrenheit):
    """A function to convert from degrees Fahrenheit to degrees Centigrade

    Parameters
```

(continues on next page)

(continued from previous page)

```
    ----------
    deg_fahrenheit: float
        Temperature in degrees F

    Returns
    -------
    Temperature converted to degrees C
    """
    deg_c = (deg_fahrenheit - 32.)*5./9.
    return deg_c
```

We see that the function has a name (`fahrenheit_to_centigrade`), and takes one parameter (`deg_fahrenheit`).

The main body of the function is indented (like `if` and `for` statements). There is first a comment string, that describes what the function does, as well as what the inputs are, and what the output is. This is just useful documentation of the code.

The main body of the function calculates `deg_C` from the given input, and **returns** it back to the user.

Notice that the document string `"""A function to convert from temperature... """` is what is printed when you request `help` on the function:

```
help(fahrenheit_to_centigrade)
```

```
Help on function fahrenheit_to_centigrade in module __main__:

fahrenheit_to_centigrade(deg_fahrenheit)
    A function to convert from degrees Fahrenheit to degrees Centigrade

    Parameters
    ----------
    deg_fahrenheit: float
        Temperature in degrees F

    Returns
    -------
    Temperature converted to degrees C
```

E2.1.1 Exercise

- In the vein of converting units, write functions that convert from
  - inches to m (and back)
  - kg to stones (and back)

Hint: A stone is equal to 14 pounds, and a pound is equal to 0.45359237 kg.

Ensure that your functions are clearly named, have sensible variable names, a brief docmentation string, and remember to test the functions work: just demonstrate running the function with some input pairs where you know the output and checking it makese sense.

```
# Space for your solution
# ANSWER
# conversion factors found from
# googling
```

(continues on next page)

(continued from previous page)

```
def inches_to_metre(value_inches):
    '''
    convert input value in inches
    to metres
    '''
    return (value_inches * 0.0254)

def metre_to_inches(value_metre):
    '''
    convert input value in metres
    to inches
    '''
    return (value_metre / 0.0254)

def kg_to_stone(value_kg):
    '''
    convert input value in Kg to stone
    '''
    return(value_kg*0.157473)

def stone_to_kg(value_stone):
    '''
    convert input value in stone to Kg
    '''
    return(value_stone/0.157473)

# test:

print('6 inches is',inches_to_metre(6),'m')
print('70 Kg is',kg_to_stone(70),'stone')
```

```
6 inches is 0.15239999999999998 m
70 Kg is 11.02311 stone
```

2.2 numpy

2.2.1 arrays

You import the `numpy` library using

```
import numpy as np
```

This means that all the functionality of `numpy` is accessed by the prefix `np.`: e.g. `np.array`. The main element of `numpy` is the numpy array. An array is like a list, but unlike a list, all the elements are of the same type, floating point numbers for example.

Let's see some arrays in action…

```
import numpy as np  # Import the numpy library

# An array with 5 ones
arr = np.ones(5)
print(arr)
print(type(arr))
```

(continues on next page)

(continued from previous page)

```
# An array started from a list of **integers**
arr = np.array([1, 2, 3, 4])
print(arr)

# An array started from a list of numbers, what's the difference??
arr = np.array([1., 2, 3, 4])
print(arr)
```

```
[1. 1. 1. 1. 1.]
<class 'numpy.ndarray'>
[1 2 3 4]
[1. 2. 3. 4.]
```

In the example above we have generated an array where all the elements are `1.0`, using `` `np.ones `` <https://docs.scipy.org/doc/numpy/reference/generated/numpy.ones.html>'__, and then we have been able to generate arrays from lists using the `` `np.array `` <https://docs.scipy.org/doc/numpy/reference/generated/numpy.array.html>'__ function. The difference between the 2nd and 3rd examples is that in the 2nd example, all the elements of the list are integers, and in the 3rd example, one is a floating point number. `numpy` automatically makes the array floating point by converting the integers to floating point numbers.

What can we do with arrays? We can efficiently operate on individual elements without loops:

```
arr = np.ones(10)
print(2 * arr)
```

```
[2. 2. 2. 2. 2. 2. 2. 2. 2. 2.]
```

`numpy` is clever enough to figure out that the 2 multiplying the array is applied to all elements of the array, and returns an array of the same size as `arr` with the elements of `arr` multiplied by 2. We can also multiply two arrays of the same size. So let's create an array with the numbers 0 to 9 and one with the numbers 9 to 0 and do a times table:

```
arr1 = 9 * np.ones(10)
arr2 = np.arange(1, 11)  # arange gives an array from 1 to 11, 11 not included

print(arr1)
print(arr2)

print(arr1 * arr2)
```

```
[9. 9. 9. 9. 9. 9. 9. 9. 9. 9.]
[ 1  2  3  4  5  6  7  8  9 10]
[ 9. 18. 27. 36. 45. 54. 63. 72. 81. 90.]
```

E2.2.1 Exercise

- Using code similar to the above and a `for` loop, write the times tables for 2 to 10. The solution you're looking for should look a bit like this:

```
[ 2  4  6  8 10 12 14 16 18 20]
[ 3  6  9 12 15 18 21 24 27 30]
[ 4  8 12 16 20 24 28 32 36 40]
[ 5 10 15 20 25 30 35 40 45 50]
[ 6 12 18 24 30 36 42 48 54 60]
[ 7 14 21 28 35 42 49 56 63 70]
```

(continues on next page)

(continued from previous page)

```
[ 8 16 24 32 40 48 56 64 72 80]
[ 9 18 27 36 45 54 63 72 81 90]
[ 10  20  30  40  50  60  70  80  90 100]
```

```
# Your solution here
# ANSWER

a = np.arange(1, 11)

for b in range(2,11):
    print(a * b)
```

```
[ 2  4  6  8 10 12 14 16 18 20]
[ 3  6  9 12 15 18 21 24 27 30]
[ 4  8 12 16 20 24 28 32 36 40]
[ 5 10 15 20 25 30 35 40 45 50]
[ 6 12 18 24 30 36 42 48 54 60]
[ 7 14 21 28 35 42 49 56 63 70]
[ 8 16 24 32 40 48 56 64 72 80]
[ 9 18 27 36 45 54 63 72 81 90]
[ 10  20  30  40  50  60  70  80  90 100]
```

If the arrays are of the same *shape*, you can do standard operations between them **element-wise**:

```
arr1 = np.array([3, 4, 5, 6.])
arr2 = np.array([30, 40, 50, 60.])

print(arr2 - arr1)
print(arr1 * arr2)

print("Array shapes:")
print("arr1: ", arr1.shape)
print("arr2: ", arr2.shape)
```

```
[27. 36. 45. 54.]
[ 90. 160. 250. 360.]
Array shapes:
arr1:  (4,)
arr2:  (4,)
```

The `numpy` documenation is huge. There's an user's guide, as well as a reference to all the contents of the library. There's even a tutorial availabe if you get bored with this one.

2.2.2 More detail about `numpy.arrays`

So far, we have seen a 1D array, which is the equivalent to a vector. But arrays can have more dimensions: a 2D array would be equivalent to a matrix (or an image, with rows and columns), and a 3D array would be a volume split into voxels, as seen below

So a 1D array has one axis, a 2D array has 2 axes, a 3D array 3, and so on. The `shape` of the array provides a tuple with the number of elements along each axis. Let's see this with some generally useful array creation options:

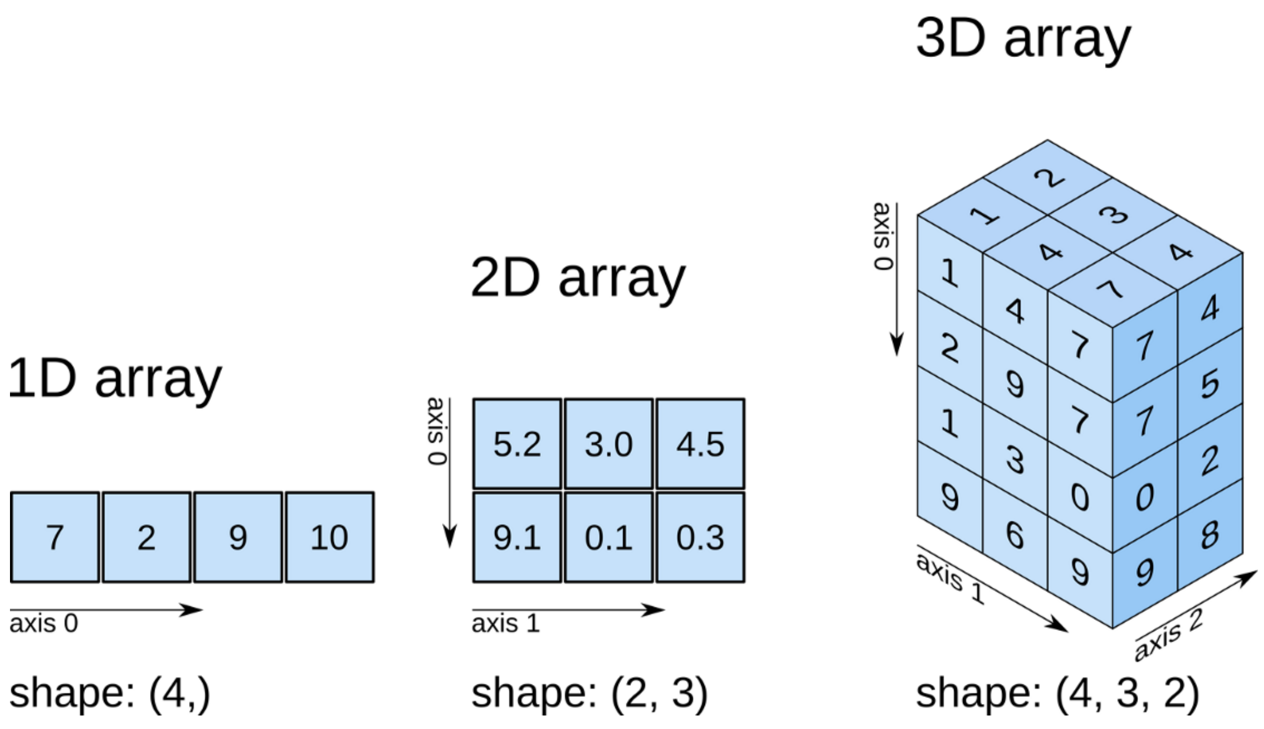


Fig. 1: numpy arrays

```
# Create a 2D array from a list of rows. Note that the 3 rows have the same number of␣
↪elements!
arr1 = np.array([[0, 1, 2, 3, 4], [5, 6, 7, 8, 9], [10, 11, 12, 13, 14]])
# A 2D array from a list of tuples.
# We're specifically asking for floating point numbers
arr2 = np.array([(1.5, 2, 3), (4, 5, 6)], dtype=np.float)
print("3*5 array:")
print(arr1)
print("2*3 array:")
print(arr2)
```

```
3*5 array:
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
2*3 array:
[[1.5 2.  3. ]
 [4.  5.  6. ]]
```

2.2.3 Array creators

Quite often, we will want to initialise an array to be all the same number. The methods for doing this as 0,1 and unspecified in `numpy` are `np.zeros()`, `np.ones()`, `np.empty()` respectively.

```
# Creates a 3*4 array of 0s
arr = np.zeros((3, 4))
print("3*4 array of 0s")
```

(continues on next page)

(continued from previous page)

```
print(arr)

# Creates a 2x3x4 array of int 1's
print("2*3*4 array of 1s (integers)")
arr = np.ones((2, 3, 4), dtype=np.int)
print(arr)

# Creates an empty (e.g. uninitialised) 2x3 array. Elements are random
print("2*3 empty array (contents could be anything)")
arr = np.empty((2, 3))
print(arr)
```

```
3*4 array of 0s
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
2*3*4 array of 1s (integers)
[[[1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]]

 [[1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]]]
2*3 empty array (contents could be anything)
[[1.5 2.  3. ]
 [4.  5.  6. ]]
```

Exercise E2.2.2

- write a function that does the following:
  - create a 2-D tuple called `indices` containing the integers `((0, 1, 2, 3, 4),(5, 6, 7, 8, 9))`
  - create a 2-D numpy array called `data` of shape `(5,10)`, data type `int`, initialised with zero
  - set the value of `data[r,c]` to be `1` for each of the 5 row,column pairs specified in `indices`.
  - return the data array
- print out the result returned

The result should look like:

```
[[0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 0 1]]
```

Hint: You could use a `for` loop, but what does `data[indices]` give?

```
# do exercise here
# ANSWER

def doit():
    indices = ((0, 1, 2, 3, 4),(5, 6, 7, 8, 9))
    data = np.zeros((5,10),dtype=np.int)
```

(continues on next page)

(continued from previous page)

```
    data[indices] = 1
    return(data)

print(doit())
```

```
[[0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 0 1]]
```

Exercise 2.2.3

- write a more flexible version of you function above where `indices`, the value you want to set (`1` above) and the desired shape of `data` are specified through function keyword arguments (e.g. `indices=((0, 1, 2, 3, 4),(5, 6, 7, 8, 9)),value=1,shape=(5,10))`

```
# do exercise here
# ANSWER

def doit(indices = ((0, 1, 2, 3, 4),(5, 6, 7, 8, 9)),\
         shape   = (5,10),\
         value   = 1):

    data = np.zeros(shape,dtype=type(value))
    data[indices] = value
    return(data)

print(doit(value=10.5))
```

```
[[ 0.   0.   0.   0.   0.  10.5  0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.  10.5  0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.  10.5  0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   0.  10.5  0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.  10.5]]
```

As well as initialising arrays with the same number as above, we often also want to initialise with common data patterns. This includes simple integer ranges `(start, stop, skip)` in a similar fashion to slicing in the last session, or variations on this theme:

```
### array creators

print("1D array of numbers from 0 to 2 in increments of 0.3")
start = 0
stop  = 2.0
skip  = 0.3

arr = np.arange(start,stop,skip)
print(f'arr of shape {arr.shape}:\n\t{arr}')

start  = 0
stop   = 34
nsamp  = 9
arr = np.linspace(start,stop,nsamp)
print(f"array of shape {arr.shape} numbers equally spaced from {start} to {stop}:\n\t
↪{arr}")
```

(continues on next page)

(continued from previous page)

```
np.linspace(stop,start,9)
```

```
1D array of numbers from 0 to 2 in increments of 0.3
arr of shape (7,):
    [0.  0.3 0.6 0.9 1.2 1.5 1.8]
array of shape (9,) numbers equally spaced from 0 to 34:
    [ 0.    4.25  8.5  12.75 17.   21.25 25.5  29.75 34.  ]
```

```
array([34.  , 29.75, 25.5 , 21.25, 17.  , 12.75,  8.5 ,  4.25,  0.  ])
```

Exercise E2.2.4

- print an array of integer numbers from 100 to 1
- print an array with 9 numbers equally spaced between 100 and 1

Hint: what value of skip would be appropriate here? what about `start` and `stop`?

```
# do exercise here
print(np.arange(100,0,-1))
print(np.linspace(100,0,9))
```

```
[100  99  98  97  96  95  94  93  92  91  90  89  88  87  86  85  84  83
  82  81  80  79  78  77  76  75  74  73  72  71  70  69  68  67  66  65
  64  63  62  61  60  59  58  57  56  55  54  53  52  51  50  49  48  47
  46  45  44  43  42  41  40  39  38  37  36  35  34  33  32  31  30  29
  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11
  10   9   8   7   6   5   4   3   2   1]
[100.   87.5  75.   62.5  50.   37.5  25.   12.5   0. ]
```

2.2.4 Summary statistics

Below are some typical arithmetic operations that you can use on arrays. Remember that they happen **elementwise** (i.e. to the whole array).

```
b = np.arange(4)
print(f'{b}^2 = {b**2}\n')

a = np.array([20, 30, 40, 50])
print(f"assuming in radians,\n10*sin({a}) = {10 * np.sin(a)}")

print("\nSome useful numpy array methods for summary statistics...\n")
print("Find the maximum of an array: a.max(): ", a.max())
print("Find the minimum of an array: a.min(): ", a.min())
print("Find the sum of an array: a.sum(): ", a.sum())
print("Find the mean of an array: a.mean(): ", a.mean())
print("Find the standard deviation of an array: a.std(): ", a.std())
```

```
[0 1 2 3]^2 = [0 1 4 9]

assuming in radians,
10*sin([20 30 40 50]) = [ 9.12945251 -9.88031624  7.4511316  -2.62374854]
```

(continues on next page)

(continued from previous page)

```
Some useful numpy array methods for summary statistics...

Find the maximum of an array: a.max():  50
Find the minimum of an array: a.min():  20
Find the sum of an array: a.sum():  140
Find the mean of an array: a.mean():  35.0
Find the standard deviation of an array: a.std():  11.180339887498949
```

Let's access an interesting dataset on the frequency of satellite launches to illustrate this.

Fig. 2: SpaceX landing

```
from geog0111.nsat import nsat

'''
This dataset gives the number of
satellites launched per month and year
data from https://www.n2yo.com
'''
# We use the code supplied in nsat.py
# to generate the dataset (takes time)
# or to load it if it exists
data,years = nsat().data,nsat().years

print(f'data shape {data.shape}')

print(f'some summary statistics over the period {years[0]} to {years[1]}:')
print(f'The total number of launches is {data.sum():d}')
print(f'The mean number of launches is {data.mean():.3f} per month')
```

```
data shape (12, 62)
some summary statistics over the period 1957 to 2019:
The total number of launches is 43611
The mean number of launches is 58.617 per month
```

Exercise E2.2.5

- copy the code above but generate a fuller set of summary statistics including the standard deviation, minimum and maximum.

```
# do exercise here
# ANSWER

data,years = nsat().data,nsat().years

print(f'data shape {data.shape}')

print(f'some summary statistics over the period {years[0]} to {years[1]}:')

print(f'The total number of launches is {data.sum():d}')
print(f'The mean number of launches is {data.mean():.3f} per month')
print(f'The minimum number of launches is {data.min():.3f} per month')
print(f'The maximum number of launches is {data.max():.3f} per month')
print(f'The std dev number of launches is {data.std():.3f} per month')
print(f'The median number of launches is {np.median(data):.3f} per month')
```

```
data shape (12, 62)
some summary statistics over the period 1957 to 2019:
The total number of launches is 43611
The mean number of launches is 58.617 per month
The minimum number of launches is 0.000 per month
The maximum number of launches is 3476.000 per month
The std dev number of launches is 161.904 per month
The median number of launches is 30.000 per month
```

Whilst we have generated some interesting summary statistics on the dataset, it's not really enough to give us a good idea of the data characteristics.

To do that, we want to be able to ask somewhat more complex questions of the data, such as, which *year* has the most/least launches? which month do most launches happen in? which month in which year had the most launches? which years had more than 100 launches?

To be able to address these, we need some new concepts:

- methods `argmin()` and `argmax()` that provide the *index* where the min/max occurs
- filtering and the related method `where()`
- `axis` methods: the dataset is two-dimensional, and for some questions we need to operate only over one of these

To illustrate:

```
from geog0111.nsat import nsat
import numpy as np

data,years = nsat().data,nsat().years

year = np.arange(years[0],years[1],dtype=np.int)

# sum the data over all months (axis 0)
sum_per_year = data.sum(axis=0)

imax = np.argmax(sum_per_year)
imin = np.argmin(sum_per_year)

# filtering
# high(low) is an array set to True where the condition
# is True, and False otherwise
high = sum_per_year>=1000
low  = sum_per_year<=300

print(f'the year with most launches was {year[imax]} with {sum_per_year[imax]}')
print(f'the year with fewest launches was {year[imin]} with {sum_per_year[imin]}')

print('\nThe years with >= 1000 launches are:')
print(year[high],'\nvalues:\n',sum_per_year[high])
print('The years with <= 300 launches are:')
print(year[low],'\nvalues:\n',sum_per_year[low])
```

```
the year with most launches was 1999 with 4195
the year with fewest launches was 1957 with 3

The years with >= 1000 launches are:
[1965 1975 1976 1981 1986 1987 1993 1994 1999 2006]
```

(continues on next page)

(continued from previous page)

```
values:
 [1527 1195 1264 1190 1375 1130 2131 1166 4195 1158]
The years with <= 300 launches are:
[1957 1958 1959 1960 1962 1996 2002 2003 2004 2005]
values:
 [  3  11  22  52 207 246 277 243 209 192]
```

Exercise E2.2.6

- copy the code above, and modify it to find the total launches *per month* (over all years)
- show these data in a table
- which month do launches mostly take place in? which month do launches most seldom take place in?

```
# do exercise here
# ANSWER

from geog0111.nsat import nsat
import numpy as np
from datetime import datetime

data,years = nsat().data,nsat().years

year = np.arange(years[0],years[1],dtype=np.int)

month = np.arange(0,12)
# OR BETTER AS STRINGS ... use datetime for things like this
# see http://blog.e-shell.org/94
month = np.array([datetime(2018, i, 1).strftime('%B') for i in range(1,13)])

# sum the data over all years (axis 1)
sum_per_month = data.sum(axis=1)

imax = np.argmax(sum_per_month)
imin = np.argmin(sum_per_month)

# filtering
# high(low) is an array set to True where the condition
# is True, and False otherwise
high = sum_per_month>=5000
low  = sum_per_month<=3000

print(high,low)

print(f'the month with most launches was {month[imax]} with {sum_per_month[imax]}')
print(f'the month with fewest launches was {month[imin]} with {sum_per_month[imin]}')

print('\nThe months with >= 5000 launches are:')
print(month[high],'\nvalues:\n',sum_per_month[high])
print('The months with <= 3000 launches are:')
print(month[low],'\nvalues:\n',sum_per_month[low])
```

```
[False False False False  True  True False False False False False False] [ True␣
↪False  True False False False False  True False False False False]
the month with most launches was May with 6504
the month with fewest launches was January with 1868
```

(continues on next page)

(continued from previous page)

```
The months with >= 5000 launches are:
['May' 'June']
values:
 [6504 5559]
The months with <= 3000 launches are:
['January' 'March' 'August']
values:
 [1868 2771 2313]
```

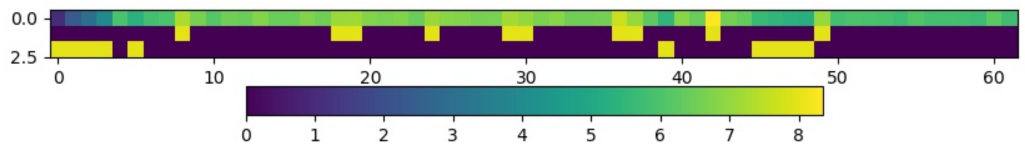
The form of filtering above (`high = sum_per_year>=1000`) produces a numpy array of the same shape as that operated on (`sum_per_year` here) of `bool` data type. It has entries of `True` where the condition is met, and `False` where it is not met.

```
from geog0111.nsat import nsat
# sum the data over all months (axis 0)
sum_per_year = nsat().data.sum(axis=0)

high = sum_per_year>=1000
low  = sum_per_year<=300

print(f'type(sum_per_year): {type(sum_per_year)}, sum_per_year.shape: {sum_per_year.
↪shape}, ' \
        + f'sum_per_year.dtype: {sum_per_year.dtype}')
print(f'type(high): {type(high)}, high.shape: {high.shape}, high.dtype: {high.dtype}\n
↪')

print(f'sum_per_year: {sum_per_year}')
print(f'high: {high}')
print(f'low: {low}')
```

```
type(sum_per_year): <class 'numpy.ndarray'>, sum_per_year.shape: (62,), sum_per_year.
↪dtype: int64
type(high): <class 'numpy.ndarray'>, high.shape: (62,), high.dtype: bool

sum_per_year: [   3   11   22   52  396  207  346  401 1527  786  466  690  641  906
  636  654  875  694 1195 1264  891  783  857  637 1190  946  884  760
  788 1375 1130  814  950  691  691  740 2131 1166  534  246  960  651
 4195  730  582  277  243  209  192 1158  349  406  378  373  315  435
  352  355  335  308  512  320]
high: [False False False False False False False False  True False False False
 False False False False False False  True  True False False False False
  True False False False False  True  True False False False False False
  True  True False False False False  True False False False False False
 False  True False False False False False False False False False False
 False False]
low: [ True  True  True  True False  True False False False False False False
 False False False False False False False False False False False False
 False False False False False False False False False False False False
 False False False  True False False False False False  True  True  True
  True False False False False False False False False False False False
 False False]
```

We can think of this logical array as a 'data mask' that we use to select (filter) entries.

The figure shows `log(sum_per_year)` in the top line of the image (numbers represented by colour shown in colourbar), then a representation of the `bool` arrays high and low. Where the bool value is shown yellow, the 'data mask' is true.



```
print(f'{sum_per_year[high]}')
print(f'{sum_per_year[low]}')
```

```
[1527 1195 1264 1190 1375 1130 2131 1166 4195 1158]
[   3   11   22   52 207 246 277 243 209 192]
```

Sometimes, instead of just applying the filter as above, we want to know the indices of the filtered values.

To do this, we can use the `np.where()` method. This takes a `bool` array as its argument (such as our data masks or other conditions) and returns a tuple of the indices where this is set `True`.

```
from geog0111.nsat import nsat
data,years = nsat().data,nsat().years
# where :
# which months in the dataset were particularly busy ..
# we select data > 400 as a condition

indices = np.where(data > 400)
print(f'indices:\n{indices[0]}\n{indices[1]}')
print(f'\ntype(indices): {type(indices)}')
print(f'len(indices): {len(indices)}, len(indices[0]): {len(indices[0])}')
print(f'type(indices[0][0]): {type(indices[0][0])}')

year = np.arange(years[0],years[1],dtype=np.int)
month = np.arange(12)

nsamp = len(indices[0])

# loop over the entries in the tuple
print('*'*23)
print('busy months')
print('*'*23)
for i in range(nsamp):
    print(f'{i:04d} month {month[indices[0][i]]:02d}'+\
                  f' year {year[indices[1][i]]:04d}')
print('*'*23)
```

```
indices:
[1 3 4 4 5 5 5 6 8 8 9 9]
[29 13 37 42 24 36 49 19 40 43  8 42]

type(indices): <class 'tuple'>
len(indices): 2, len(indices[0]): 12
type(indices[0][0]): <class 'numpy.int64'>
*******************
busy months
*******************
```

```
0000 month 01 year 1986
0001 month 03 year 1970
0002 month 04 year 1994
0003 month 04 year 1999
0004 month 05 year 1981
0005 month 05 year 1993
0006 month 05 year 2006
0007 month 06 year 1976
0008 month 08 year 1997
0009 month 08 year 2000
0010 month 09 year 1965
0011 month 09 year 1999
*******************
```

Exercise E2.2.7

- Using code from the sections above, print out a table with the busiest launch months with an additional column stating the number of launches

Hint: this is just adding another column to the print statement in the for loop

```python
# do exercise here
# ANSWER

from geog0111.nsat import nsat
data,years = nsat().data,nsat().years
# where :
# which months in the dataset were particularly busy ..
# we select data > 400 as a condition

indices = np.where(data > 400)
print(f'indices:\n{indices[0]}\n{indices[1]}')
print(f'\ntype(indices): {type(indices)}')
print(f'len(indices): {len(indices)}, len(indices[0]): {len(indices[0])}')
print(f'type(indices[0][0]): {type(indices[0][0])}')

year = np.arange(years[0],years[1],dtype=np.int)
month = np.arange(12)

nsamp = len(indices[0])

# loop over the entries in the tuple
print('*'*23)
print('busy months')
print('*'*23)
for i in range(nsamp):
    '''
    Add in extra column here
    '''
    m = month[indices[0][i]]
    y = year[indices[1][i]]
    print(f'{i:04d} month {m:02d}'+\
                  f' year {y:04d}'+\
                  f' n_launches {data[m-1,y-years[0]]}')
print('*'*23)
```

```
indices:
[1 3 4 4 5 5 5 6 8 8 9 9]
```

```
[29 13 37 42 24 36 49 19 40 43  8 42]

type(indices): <class 'tuple'>
len(indices): 2, len(indices[0]): 12
type(indices[0][0]): <class 'numpy.int64'>
*******************
busy months
*******************
0000 month 01 year 1986 n_launches 38
0001 month 03 year 1970 n_launches 18
0002 month 04 year 1994 n_launches 21
0003 month 04 year 1999 n_launches 30
0004 month 05 year 1981 n_launches 60
0005 month 05 year 1993 n_launches 28
0006 month 05 year 2006 n_launches 16
0007 month 06 year 1976 n_launches 59
0008 month 08 year 1997 n_launches 36
0009 month 08 year 2000 n_launches 24
0010 month 09 year 1965 n_launches 34
0011 month 09 year 1999 n_launches 39
*******************
```

You might notice the indices in the tuple derived above using `where` are *ordered*, but the effect of this is that the months are in sequential order, rather than the years. We have

```
month[indices[0][i]]
year[indices[1][i]]
```

If we want to put the data in year order, there are several ways we could go about this. An insteresting one, following the ideas in `argmax()` and `argmin()` above is to use `argsort()`. This gives the *indices* of the sorted array, rather than the values.

So here, we can find the *indices* of the year-sorted array, and apply them to both `month` and `year` datasets:

```
# prepare data as above
from geog0111.nsat import nsat
data,years = nsat().data,nsat().years
indices = np.where(data > 400)
year = np.arange(years[0],years[1],dtype=np.int)
month = np.arange(12,dtype=np.int)

# store the months and years
# in their unsorted (original) form
unsorted_months = month[indices[0]]
unsorted_years = year[indices[1]]
print(f'years not in order: {unsorted_years}')
print(f'but months are:     {unsorted_months}\n')


# get the indices to put years in order
year_order = np.argsort(indices[1])

# apply this to months and years
print(f'year order: {year_order}\n')
print(f'years in order: {unsorted_years[year_order]}')
print(f'months in year order: {unsorted_months[year_order]}')
```

```
years not in order: [1986 1970 1994 1999 1981 1993 2006 1976 1997 2000 1965 1999]
but months are:     [1 3 4 4 5 5 5 6 8 8 9 9]

year order: [10  1  7  4  0  5  2  8  3 11  9  6]

years in order: [1965 1970 1976 1981 1986 1993 1994 1997 1999 1999 2000 2006]
months in year order: [9 3 6 5 1 5 4 8 4 9 8 5]
```

Exercise E2.2.8

- Use this example of `argsort()` to redo Exercise E2.2.7, putting the data in correct year order

```
# do exercise here
# ANSWER
'''
This is quite tricky to get right ...

first, work through the example above then apply
what you have learned

Dont forget to check the results against the
table above!

'''

from geog0111.nsat import nsat
data,years = nsat().data,nsat().years
# where :
# which months in the dataset were particularly busy ..
# we select data > 400 as a condition

indices = np.where(data > 400)
print(f'indices:\n{indices[0]}\n{indices[1]}')
print(f'\ntype(indices): {type(indices)}')
print(f'len(indices): {len(indices)}, len(indices[0]): {len(indices[0])}')
print(f'type(indices[0][0]): {type(indices[0][0])}')

year = np.arange(years[0],years[1],dtype=np.int)
month = np.arange(12)

'''
store unsorted data
'''
unsorted_months = month[indices[0]]
unsorted_years = year[indices[1]]

# get the indices to put years in order
year_order = np.argsort(indices[1])

nsamp = len(indices[0])

# loop over the entries in the tuple
print('*'*23)
print('busy months')
print('*'*23)
for i in range(nsamp):
    '''
    Add in extra column here
```

(continues on next page)

(continued from previous page)

```
    '''
    m = unsorted_months[year_order[i]]
    y = unsorted_years[year_order[i]]
    print(f'{i:04d} month {m:02d}'+\
                    f' year {y:04d}'+\
                    f' n_launches {data[m-1,y-years[0]]}')
print('*'*23)
```

```
indices:
[1 3 4 4 5 5 5 6 8 8 9 9]
[29 13 37 42 24 36 49 19 40 43  8 42]

type(indices): <class 'tuple'>
len(indices): 2, len(indices[0]): 12
type(indices[0][0]): <class 'numpy.int64'>
*******************
busy months
*******************
0000 month 09 year 1965 n_launches 34
0001 month 03 year 1970 n_launches 18
0002 month 06 year 1976 n_launches 59
0003 month 05 year 1981 n_launches 60
0004 month 01 year 1986 n_launches 38
0005 month 05 year 1993 n_launches 28
0006 month 04 year 1994 n_launches 21
0007 month 08 year 1997 n_launches 36
0008 month 04 year 1999 n_launches 30
0009 month 09 year 1999 n_launches 39
0010 month 08 year 2000 n_launches 24
0011 month 05 year 2006 n_launches 16
*******************
```

2.2.5 Summary

In this section, you have been introduced to more detail on arrays in `numpy`. The big advantages of `numpy` are that you can easily perform array operators (such as adding two arrays together), and that `numpy` has a large number of useful functions for manipulating N-dimensional data in array form. This makes it particularly appropriate for raster geospatial data processing.

We have seen how to create various forms of array (e.g. `np.ones()`, `np.arange()`), how to calculate some basic statistics (`min()`, `max()` etc), and finding the array index where some pattern occurs (e.g. `argmin()`, `argsort()` or `where()`).

2.3 Plotting with Matplotlib

There are quite a few graphical libraries for Python, but matplotlib is probably the most famous one. It does pretty much all you need in terms of 2D plots, and simple 3D plots, and is fairly straightforward to use. Have a look at the matplotlib gallery for a fairly comprehensive list of examples of what the library can do as well as the code that was used in the examples.

Importing matplotlib

You can import matplotlib with

```
import matplotlib.pyplot as plt
```

As with `numpy`, it's custom to use the `plt` prefix to call matplotlib commands. In the notebook, you should also issue the following command just after the import

```
%matplotlib notebook
```

or

```
%matplotlib inline
```

The former command will make the plots in the notebook interactive (i.e. point-and-click-ey), and the second will just stick the plots into the notebook as PNG files.

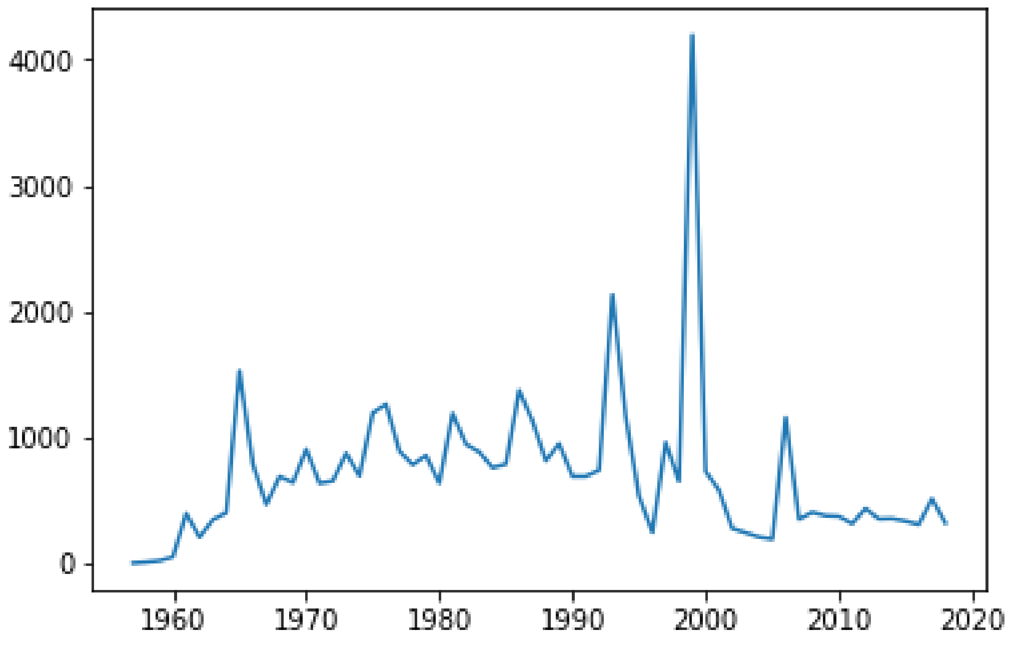
Simple 2D plots

The most basic plots are 2D plots (e.g. x and y).

```
import matplotlib.pyplot as plt
%matplotlib inline

from geog0111.nsat import nsat
import numpy as np
'''
This dataset gives the number of
satellites launched per month and year

data from https://www.n2yo.com
'''
data,years = nsat().data,nsat().years
year = np.arange(years[0],years[1],dtype=np.int)
# sum the data over all months (axis 0)
sum_per_year = data.sum(axis=0)

print(f'data shape {data.shape}')

# plot x as year
# plot y as the number of satellites per year
plt.plot(year,sum_per_year,label='launches per year')
```

```
data shape (12, 62)
```

```
[<matplotlib.lines.Line2D at 0x11d5c13c8>]
```



Whilst this plot is fine, there are a few simple things we could do improve it.

We will go through some of the options below, but to get a taste of improved ploitting, lets use e.g.:

- reset the image shape/size
  - `plt.figure(figsize=(13,3))`
- plot the mean value (as a red dashed line) for comparison
  - `plt.plot([year[0],year[-1]],[mean,mean],'r--',label='mean')`
- limit the dataset to range of variable `year`
  - `plt.xlim(year[0],year[-1])`
- put labels on the x and y axes
  - `plt.xlabel('year')`
  - `plt.ylabel('# satellite launches')`
- set a title
  - `plt.title('data from https://www.n2yo.com')`
- use a legend (in conjunction with `label=` using `plot`)
  - `plt.legend(loc='best')`
- use a log scale in the y-axis
  - `plt.semilogy()`

What you choose to do will depend on what you want to show on the graph, but the examples above are quite common.

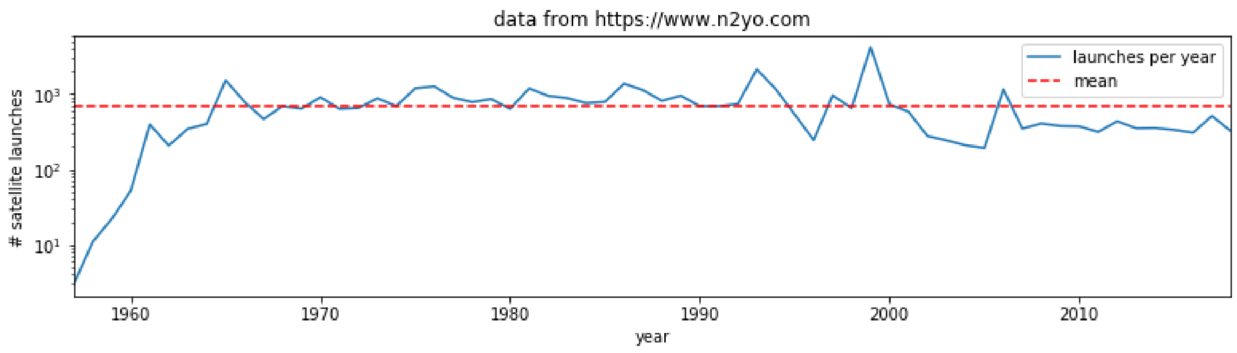
```
import matplotlib.pyplot as plt
%matplotlib inline
from geog0111.nsat import nsat
import numpy as np
```

(continues on next page)

(continued from previous page)

```
'''data as above'''
data,years = nsat().data,nsat().years
year = np.arange(years[0],years[1],dtype=np.int)
sum_per_year = data.sum(axis=0)

# calculate mean of sum_per_year
mean = sum_per_year.mean()

plt.figure(figsize=(13,3))
plt.plot(year,sum_per_year,label='launches per year')
plt.plot([year[0],year[-1]],[mean,mean],'r--',label='mean')
plt.xlim(year[0],year[-1])
plt.xlabel('year')
plt.ylabel('# satellite launches')
plt.title('data from https://www.n2yo.com')
plt.legend(loc='best')
plt.semilogy()
```

```
[]
```



Exercise E2.3.1

- produce a plot showing launches per year as a function of year, showing data for selected months individually.

Hint: do a simple plot first, then add some improvements gradually. You might set up a list of months to process and use a loop to go over each month.

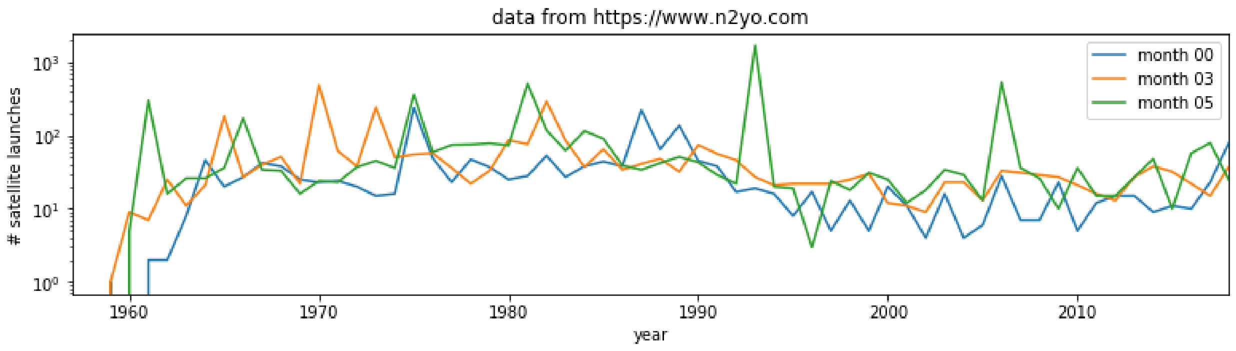
```
# do exercise here
# ANSWER

import matplotlib.pyplot as plt
%matplotlib inline
from geog0111.nsat import nsat
import numpy as np
'''data as above'''
data,years = nsat().data,nsat().years
year = np.arange(years[0],years[1],dtype=np.int)

# data contains the info we want
#sum_per_year = data.sum(axis=0)

plt.figure(figsize=(13,3))
# loop over the months we want
for month in [0,3,5]:
```

(continues on next page)

(continued from previous page)

```
    plt.plot(year,data[month],label=f'month {month:02d}')

# better still would be to use month names
# as in example above

plt.xlim(year[0],year[-1])
plt.xlabel('year')
plt.ylabel('# satellite launches')
plt.title('data from https://www.n2yo.com')
plt.legend(loc='best')
plt.semilogy()
```

```
[]
```



Exercise 2.3.2

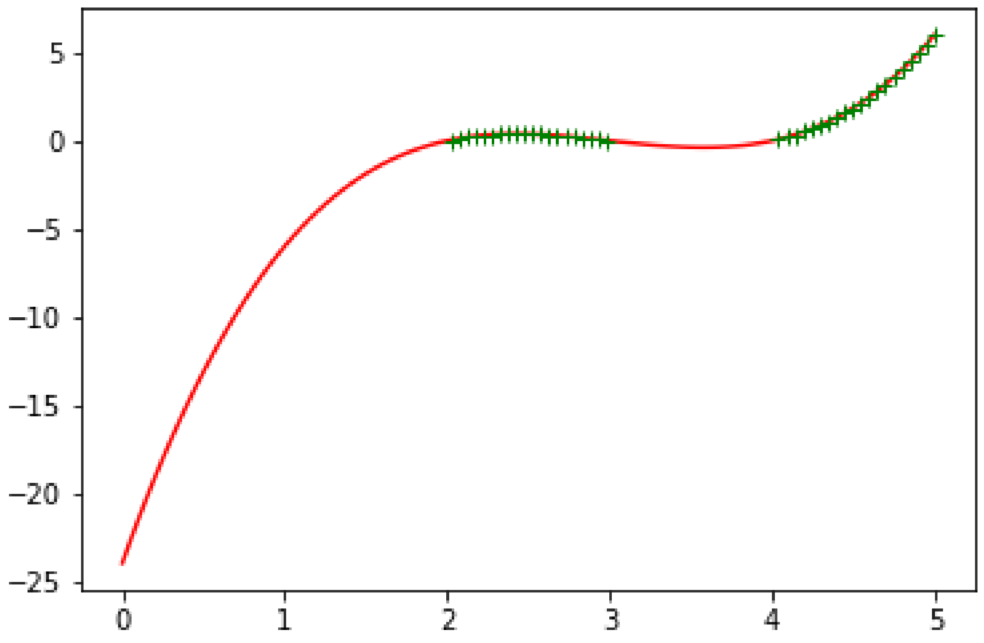
Putting together some ideas from above to look at some turning points in a function:

- generate a numpy array called x with 100 equally spaced numbers between 0 and 5
- generate a numpy array called y which contains $x^3 - 9x^2 + 26x - 24$
- plot y as a function of x with a red line
- plot **only positive** values of y (as a function of x) with a green line

Hint: to plot with red and green line `plot(x,y,'r')` and `plot(x,y,'g')`

```
# do exercise here
# ANSWER

x = np.linspace(0,5,100)
y = x**3 - 9 * x**2 + 26 * x - 24
# mask for +ve y
w = y > 0

'''
Its quite difficult to get the green line right
as it has a gap in the middle
so may be better to plot with symbols eg +
'''
plt.plot(x,y,'r-')
plt.plot(x[w],y[w],'g+')
```

```
[<matplotlib.lines.Line2D at 0x12296a048>]
```



2.4 Indexing and slicing arrays

2.4.1 Recap

Selecting different elements of the array to operate in them is a very common task. `numpy` has a very rich syntax for selecting different bits of the array. We have come across slicing before, but it is so important to array processing, we will go over some of it again.

Similar to lists, you can refer to elements in the array by their position. You can also use the `:` symbol to specify a range (a **slice**) of positions `first_element:(last_element+1`. If you want to start counting from the end of the array, use negative numbers: `-1` refers to the last element of the array, `-2` the one before last and so on. In a slice, you can also specify a step as the third element in `first_element:(last_element+1:step`. If the step is negative you count from the back.

All this probably appears mind bogging, but it's easier shown in practice. You'll get used to it quite quickly once you start using it

```
import numpy as np

a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print(a[2])       # 2
print(a[2:5])     # [2, 3, 4]
print(a[-1])      # 10
print(a[:8])      # [0, 1, 2, 3, 4, 5, 6, 7]
print(a[2:])      # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(a[5:2:-1])  # [5, 4, 3]
```

```
2
[2 3 4]
10
[0 1 2 3 4 5 6 7]
```

(continues on next page)

(continued from previous page)

```
[ 2  3  4  5  6  7  8  9 10]
[5 4 3]
```

The concept extends cleanly to multidimensional arrays…

```
b = np.array([[0, 1, 2, 3], [10, 11, 12, 13], [20, 21, 22, 23], [30, 31, 32, 33],
     [40, 41, 42, 43]])

print(b[2, 3])     # 23
print(b[0:5, 1])  # each row in the second column of b
print(b[:, 1])     # same thing as above
print(b[1:3, :])  # each column in the second and third row of b
```

```
23
[ 1 11 21 31 41]
[ 1 11 21 31 41]
[[10 11 12 13]
 [20 21 22 23]]
```

Exercise 2.4.1

- generate a 2-D numpy array of integer zeros called `x`, of shape (7,7)
- we can think of this as a square. Set the central 3 by 3 samples of the square to one
- print the result

Hint: Don't use looping, instead work out how to define the slice of the central 3 x 3 samples.

```
# do exercise here
# ANSWER

x = np.zeros((7,7)).astype(int)
r0,c0 = x.shape
cx,cy = int(r0/2),int(c0/2)

x[cx-1:cx+2,cy-1:cy+2] =1

print(x)
```

```
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 1 1 1 0 0]
 [0 0 1 1 1 0 0]
 [0 0 1 1 1 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]]
```

2.4.1 data mask

A useful way to select elements is by using what's called a mask as we saw above: an array of logical (boolean) elements that only selects the elements that are `True`:

```
a = np.arange(10)
select_me = a >= 7
print(a[select_me])
```

```
[7 8 9]
```

The previous point also shows something interesting: you can apply comparisons element by element. So in the previous example, `select_me` is a 10 element array where all the elements of `a` that are equal or higher than 7 are set to True.

If you want to build up element by element logical operations, it's best to use specialised functions like `` `np.logical_and `` <https://docs.scipy.org/doc/numpy/reference/generated/numpy.logical_and.html>`__ and friends

```
a = np.arange(100)
sel1 = a > 45
sel2 = a < 73
print(a[np.logical_and(sel1, sel2)])
```

```
[46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69
 70 71 72]
```

Exercise 2.4.2

- generate a numpy array called `x` with 100 equally spaced numbers between 0 and 5
- generate a numpy array called `y` which contains $x^3 - 9x^2 + 26x - 24$
- print the values of `x` for which `y` is greater than or equal to zero and `x` lies between 3.5 and 4.5

```
# do exercise here
# ANSWER

x = np.linspace(0,5,100)
y = x**3 - 9*x**2 + 26*x - 24

# conditions
w1 = np.logical_and(x>3.5,x<=4.5)
w2 = np.logical_and(y >= 0,w1)
print(x[w2])
```

```
[4.04040404 4.09090909 4.14141414 4.19191919 4.24242424 4.29292929
 4.34343434 4.39393939 4.44444444 4.49494949]
```

2.5 Reading data

2.5.1 `np.loadtxt`

It's a bit tedious just making up numbers to play with them, but it's easy to load up data from external files. The most common data interchange format is CSV (comma-seperated values), a plain text format. Think of CSV as a plain text table. Each element in each row is separated by a comma (although other symbols, such as white space, semicolons `;`, tabs `\t` or pipe `|` symbols are often found as delimiters). The first few lines might contain some metadata that describes the dataset, and the first line will also contain the names of the headers of the different columns. Lines starting with # tend to be ignored. An example file might look like this

```
# Monthly transatlantic airtravel, in thousands of passengers, for 1958-1960.
# There are 4 fields, "Month", "1958", "1959" and "1960" and 12 records, "JAN"␣
↪through "DEC".
# There is also an initial header line.
# And some lines with comments starting with #
```

(continues on next page)

(continued from previous page)

```
# Data obtained from https://people.sc.fsu.edu/~jburkardt/data/csv/csv.html
"Month", "1958", "1959", "1960"
"JAN",  340,  360,  417
"FEB",  318,  342,  391
"MAR",  362,  406,  419
"APR",  348,  396,  461
"MAY",  363,  420,  472
"JUN",  435,  472,  535
"JUL",  491,  548,  622
"AUG",  505,  559,  606
"SEP",  404,  463,  508
"OCT",  359,  407,  461
"NOV",  310,  362,  390
"DEC",  337,  405,  432
```

We can see the first few lines are comments or metadata, the first line without a `#` is the headers, and we note that text is entered between `"`s. In this case, the delimiter is a comma. We can read the data as an array with ``np.loadtxt` <https://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html>`__, telling it…

- to ignore lines starting by #
- to ignore the first column as it's text
- to note that the separator is a comma

```
air_travel = np.loadtxt("data/airtravel.csv", comments="#", skiprows=6, \
                        usecols=[1,2,3], delimiter=",")
print(air_travel)
print(air_travel.shape)
```

```
[[340. 360. 417.]
 [318. 342. 391.]
 [362. 406. 419.]
 [348. 396. 461.]
 [363. 420. 472.]
 [435. 472. 535.]
 [491. 548. 622.]
 [505. 559. 606.]
 [404. 463. 508.]
 [359. 407. 461.]
 [310. 362. 390.]
 [337. 405. 432.]]
(12, 3)
```

While `np.loadtxt` is quite flexible for dealing with text files, ``pandas` <https://pandas.pydata.org>`__ absolutely shines at working with tabular data. You can find a pandas quickstart tutorial here if you are curious about it!

Before we go into plotting, we can do some fun calculations (yay!) using our airtravel data

Exercise 2.5.1

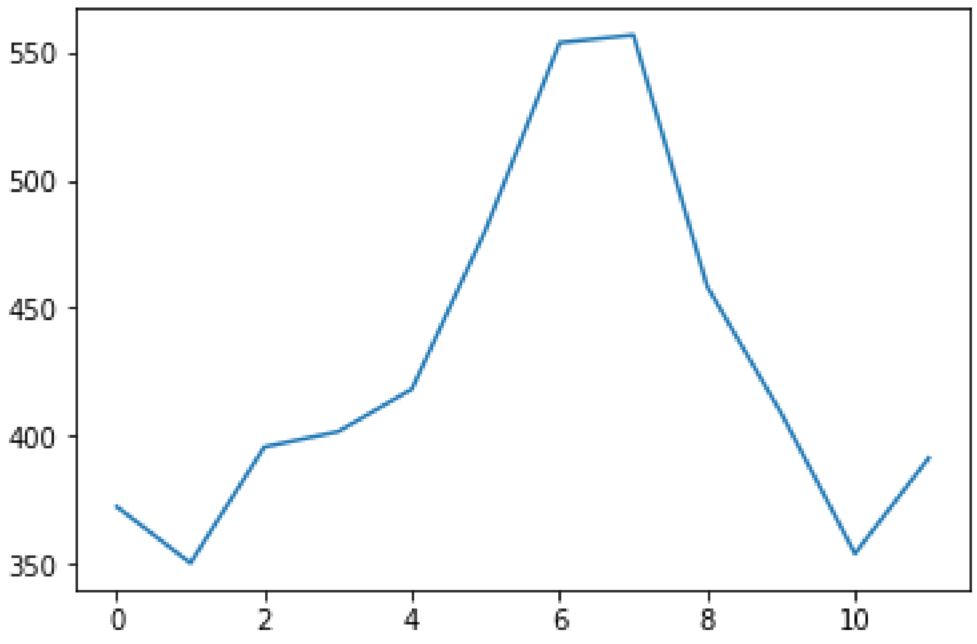
- Calculate the total number of passengers per year
- Calculate the average number of passengers per month
- Can you spot any trends in the data?

Hint: Remember the `.sum()`, `.mean()` methods for arrays?

```
# Space for your solution
# ANSWER

# shape is (12,3) so, month, year

'''
for examining data, do a plot:

we see a clear summer month peak!!

'''
print(f'total passengers per year: {air_travel.sum(axis=0)}')
print(f'mean passengers per month: {air_travel.mean(axis=1)}')

plt.plot(air_travel.mean(axis=1))
```

```
total passengers per year: [4572. 5140. 5714.]
mean passengers per month: [372.33333333 350.33333333 395.66666667 401.66666667 418.
↪33333333
 480.66666667 553.66666667 556.66666667 458.33333333 409.
 354.         391.33333333]
```

```
[<matplotlib.lines.Line2D at 0x122b060b8>]
```



Let's plot our previous air travel dataset... We'll plot it as annual lines, so the x axis will be month number (running from 1 to 12) and the y axis will be 1000s of passengers. Different line colours will be used for every year. We'll also add x and y axes labels, as well as a legend:

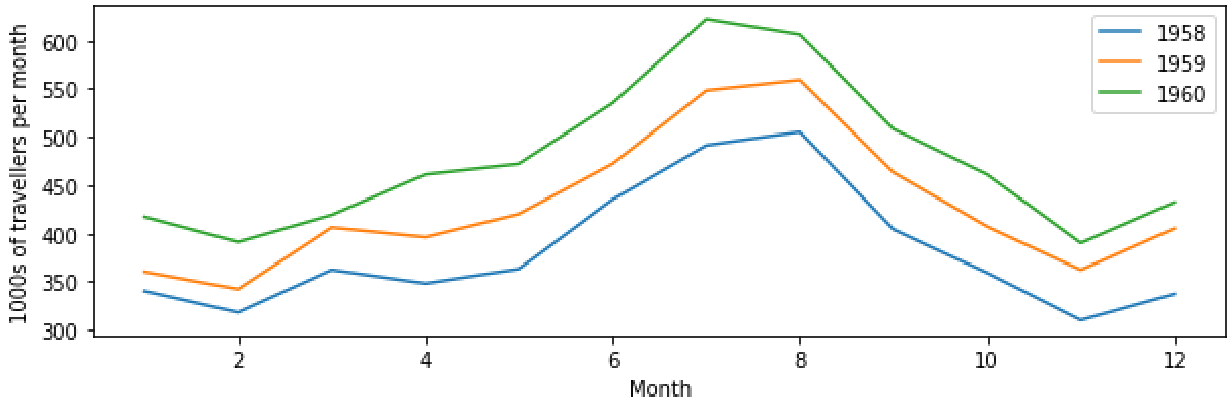
```
# You can probably just put this at the top of every notebook you write
# Adding it here for completeness
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
# Load airtravel data
air_travel = np.loadtxt("data/airtravel.csv", skiprows=6, \
                        unpack=True, usecols=[1,2,3], delimiter=",")

mths = np.arange(1, 13)
plt.figure(figsize=(10,3))
plt.plot(mths, air_travel[0], '-', label="1958")
plt.plot(mths, air_travel[1], '-', label="1959")
plt.plot(mths, air_travel[2], '-', label="1960")
plt.xlabel("Month")
plt.ylabel("1000s of travellers per month")
plt.legend(loc="best")
```

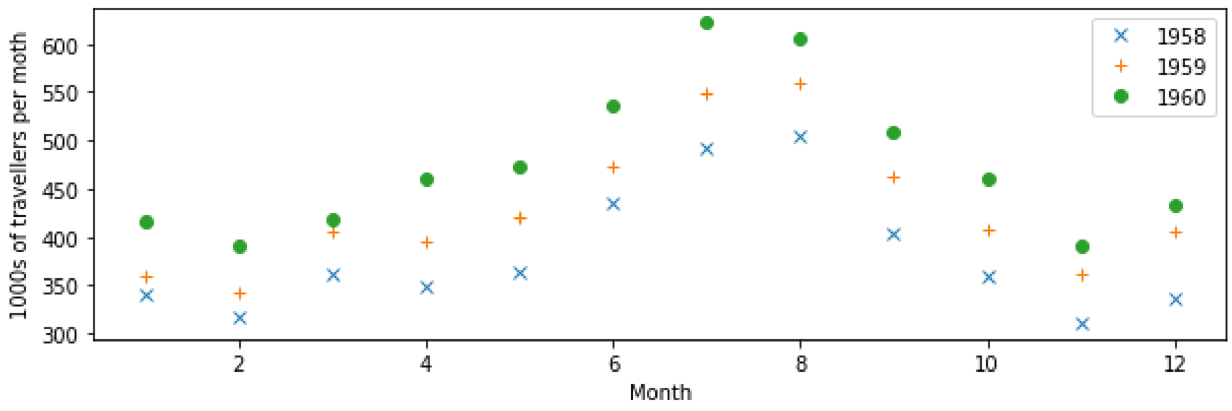
```
<matplotlib.legend.Legend at 0x11d5e4898>
```



You may not want to use lines to join the data points, but symbols like dots, crosses, etc.

```
plt.figure(figsize=(10,3))
plt.plot(mths, air_travel[0], 'x', label="1958")
plt.plot(mths, air_travel[1], '+', label="1959")
plt.plot(mths, air_travel[2], 'o', label="1960")
plt.xlabel("Month")
plt.ylabel("1000s of travellers per moth")
plt.legend(loc="best")
```

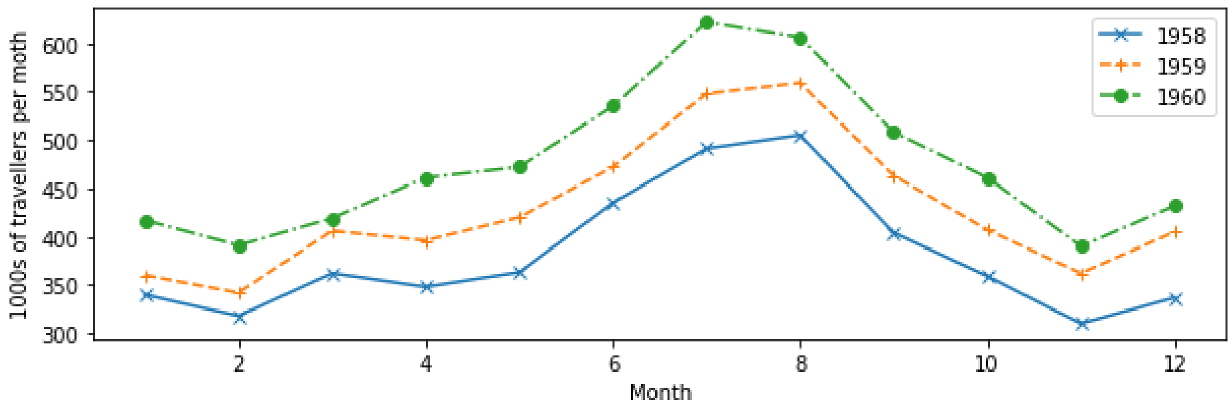
```
<matplotlib.legend.Legend at 0x122220d30>
```



We can also use dots **and** lines. Moreover, we can change the type of line: from full lines to dashed to dash-dot…

```
plt.figure(figsize=(10,3))
plt.plot(mths, air_travel[0], 'x-', label="1958")
plt.plot(mths, air_travel[1], '+--', label="1959")
plt.plot(mths, air_travel[2], 'o-.', label="1960")
plt.xlabel("Month")
plt.ylabel("1000s of travellers per moth")
plt.legend(loc="best")
```

```
<matplotlib.legend.Legend at 0x122314208>
```



Exercise 2.5.2

The file ``NOAA.csv` <data/NOAA.csv>`__ contains data from NOAA on the number of storms and hurricanes in the Atlantic basin from 1851 to 2015. The data columns are described in the first row of the file. The year is in column 1 and the number of hurricanes in column 3.

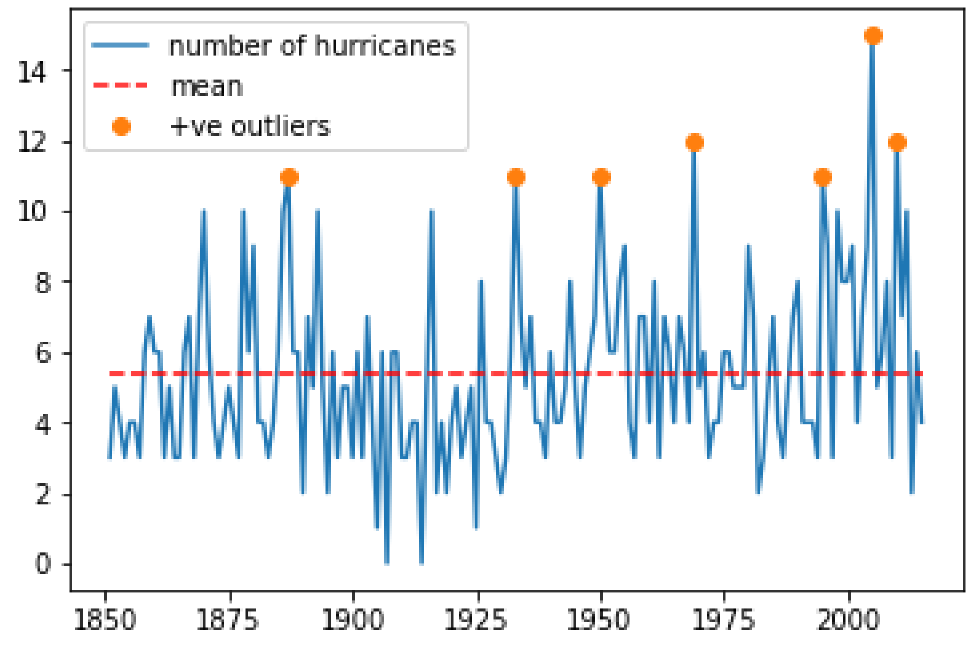
For those interested, the data is pulled from the website with getNOAA.py.

- load the year and hurricane data from the file ``NOAA.csv` <data/NOAA.csv>`__ into a numpy array
- produce a plot showing the number of hurricanes as a function of year, with the data plotted in a blue line
- put a dashed red line on the graph showing the mean number of hurricanes
- plot circle symbols for all years where the number of hurricanes is greater than the mean + 1.96 standard deviations.

Hint: the options on `np.loadtxt` you probably want to use are: `skiprows`, `delimiter`, `usecol` and `unpack`. You will need to select the data that meet the required conditions, combining the conditions with `np.logical_and()`.

```
# do exercise here

datafile = 'data/NOAA.csv'
data = np.loadtxt(datafile,skiprows=1,delimiter=',',\
                  usecols = (1,3),unpack=True)

mean = data[1].mean()
std  = data[1].std()

w = data[1] > mean + 1.96 * std

plt.plot(data[0],data[1],label='number of hurricanes')
plt.plot([data[0][0],data[0][-1]],[mean,mean],'r--',label='mean')
plt.plot(data[0][w],data[1][w],'o',label='+ve outliers')
plt.legend(loc='best')
```

```
<matplotlib.legend.Legend at 0x122eb3240>
```



2.5.2 `requests`

We can use `np.loadtxt` or similar functions to load tabular data that we have stored locally in e.g. csv format.

Sometimes we will need pull a data file from a URL. We have used this idea previously to 'scrape' data from a web page, but often the task is more straightforward, and we effectively need only to 'download' the data in the file.

We will use the `requests` package to do this and pull the data as a string. We then use `StringIO` to allow `np.loadtxt` to think the string comes from a data file.

```
import requests
from io import StringIO

# Define the URL with the parameters of interest
url = "https://daymet.ornl.gov/single-pixel/api/" + \
        "data?lat=45.4&lon=-115.0534&vars=tmax&start=2000-01-01&end=2009-12-31"

data = requests.get(url).text

# You can check the text file to see its contents, but we now
# (i) it's separated by commas
# (ii) the first 8 lines are metadata that we're not interested in.
temperature = np.loadtxt(StringIO(data), skiprows=8, delimiter=",", unpack=True)

# We expect to get 10 years of data here, so 3650 daily records
# the data are given for 365 days per year ...
print(temperature.shape)
```

```
(3, 3650)
```

If we want to store the data file, we can do so by opening a file:

```
# We open the output file, `daymet.csv`
with open("data/daymet_tmax.csv", 'w') as fp:
    # make the HTTPS connection and pull text
    # then write to file
    r = fp.write(data)

# You can check the text file to see its contents, but we now
# (i) it's separated by commas
# (ii) the first 9 lines are metadata that we're not interested in.
temperature = np.loadtxt("data/daymet_tmax.csv", skiprows=8, delimiter=",",␣
↪unpack=True)

# We expect to get ~10 years of data here, so 3650 daily records
print(temperature.shape)
```

```
(3, 3650)
```

The data columns are: `Year`, `day of year` (1 to 365) and `Tmax` (C).

How can we plot such data? the technical issue we face is needing to use the first *two* columns of data (day of year and year) to describe the x-axis location.

```
print('Year ',temperature[0])
print('DOY  ',temperature[1])
print('T_max',temperature[2])
```

```
Year  [2000. 2000. 2000. ... 2009. 2009. 2009.]
DOY   [  1.   2.   3. ... 363. 364. 365.]
T_max [-1.5 -2.5 -1.5 ... -1.5 -3.  -2. ]
```

A simple way of doing this, that would suffice here, would be to convert day of year to year fraction, then we could write:

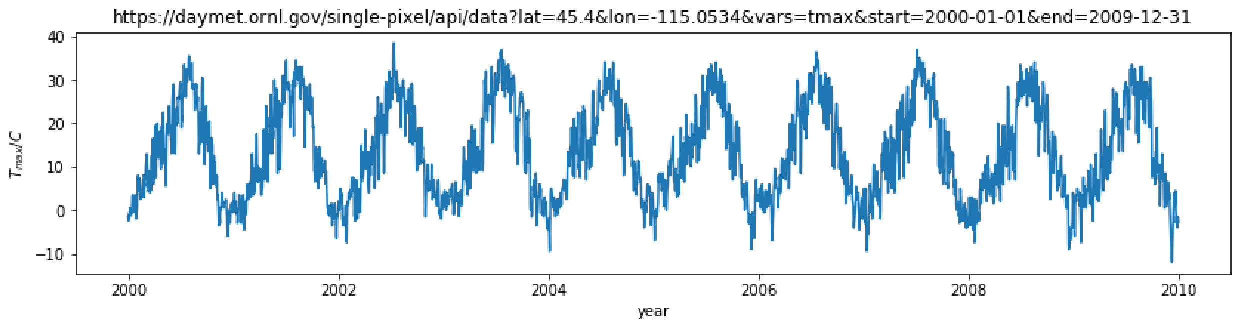
```
year,doy,tmax = temperature
dates = year + (doy-1)/365.
```

A more elegant way might be to use `` `datetime `` <https://docs.python.org/3/library/datetime.html>'__. This contains a set of methods that allow you to manipulate date formats. `matplotlib` understands the format used, and so it is generally appropriate to use `datetime` for date information when plotting.

```
year,doy,tmax = temperature
dates = [datetime.datetime(int(y), 1, 1) + \
     datetime.timedelta(d - 1) for y,d in zip(year,doy)]
```

```
import datetime

year,doy,tmax = temperature
dates = year + (doy-1)/365.
# using the simple way here

plt.figure(figsize=(14,3))
plt.plot(dates,tmax)
plt.title(url)
plt.ylabel('$T_{max} / C$')
plt.xlabel('year')
```

```
Text(0.5, 0, 'year')
```

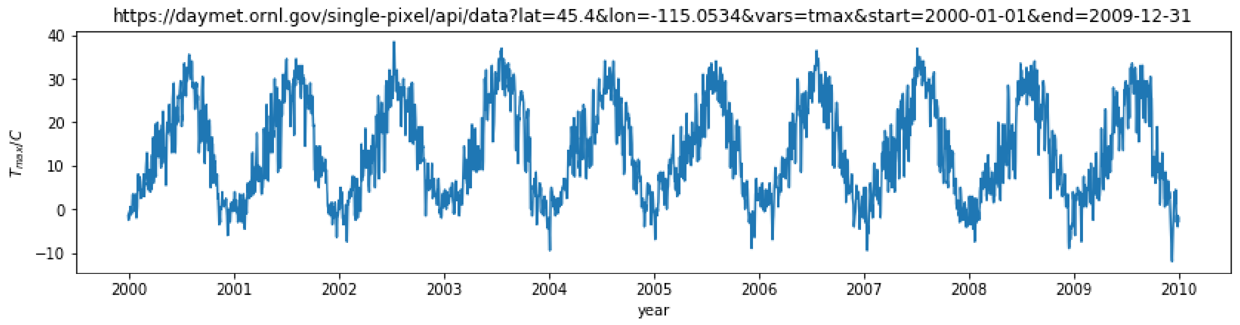


Exercise E2.5.3

- use the datetime approach to plot the dataset
- print out the value of `dates` for the first 10 entries to see what the format looks like

```
# do exercise here
# ANSWER

import datetime

year,doy,tmax = temperature

# see above!
dates = [datetime.datetime(int(y), 1, 1) + \
     datetime.timedelta(d - 1) for y,d in zip(year,doy)]

# using the simple way here

plt.figure(figsize=(14,3))
plt.plot(dates,tmax)
plt.title(url)
plt.ylabel('$T_{max} / C$')
plt.xlabel('year')
```

```
Text(0.5, 0, 'year')
```

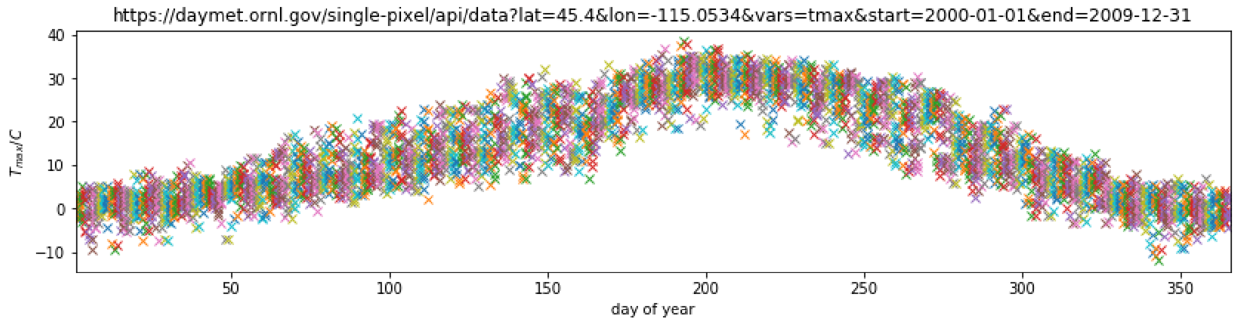


Although we have used this as a one-dimensional dataset (temperature as a function of time) we could also think of it as two-dimensional (temperature as a function of (year,doy)). Recall that the `shape` of the `temperature` dataset was `(3,3650)`. We could put the temperature column into a gridded dataset of shape `(10,365)` which would then emphasise the 2-D nature.

We can do this with the `numpy` method `reshape()`.

```
year = temperature[0].reshape(10,365)
doy  = temperature[1].reshape(10,365)
tmax = temperature[2].reshape(10,365)

plt.figure(figsize=(14,3))
plt.plot(doy,tmax,'x')
plt.title(url)
plt.ylabel('$T_{max} / C$')
plt.xlabel('day of year')
plt.xlim([1,366])
```

```
(1, 366)
```



Plotting this, we can visualise the year-on-year variations in temperature for any particular day.

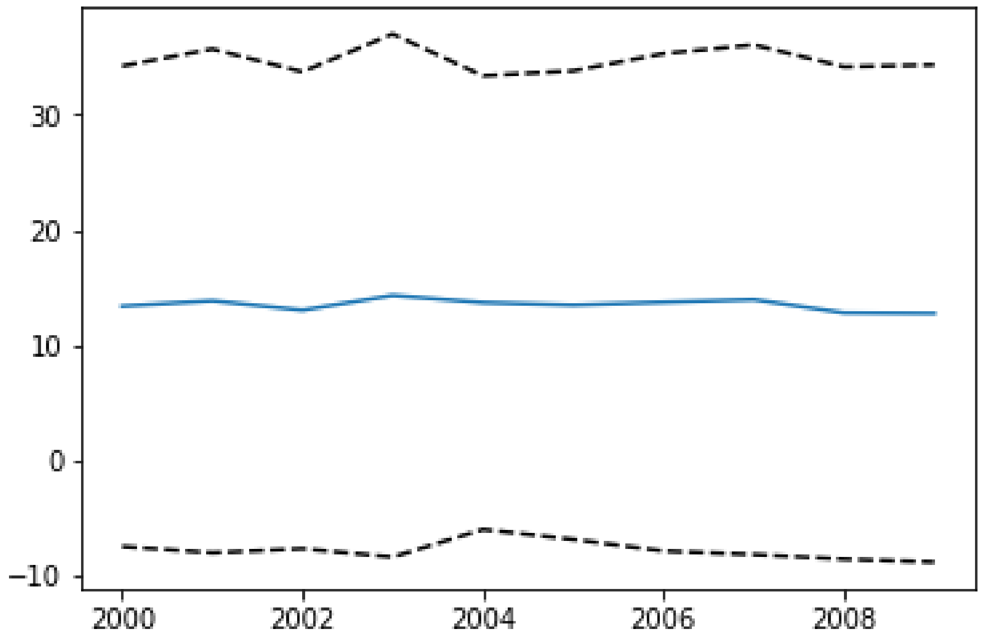
Exercise E2.5.3

- using the reshaped datasets above, calculate and plot the mean value of `tmax` as a function of day of year
- calculate standard deviation of `tmax` as a function of day of year, and plot dashed lines at mean +/- 1.96 standard deviations
- in another plot, show the mean, +/- 1.96 standard deviations of `tmax` as a function of year (i.e. the annual average and standard deviation)

Hint: use `axis=0` when calculating the mean/std over `doy` of `tmax` and `axis=1` for processing over `year`.

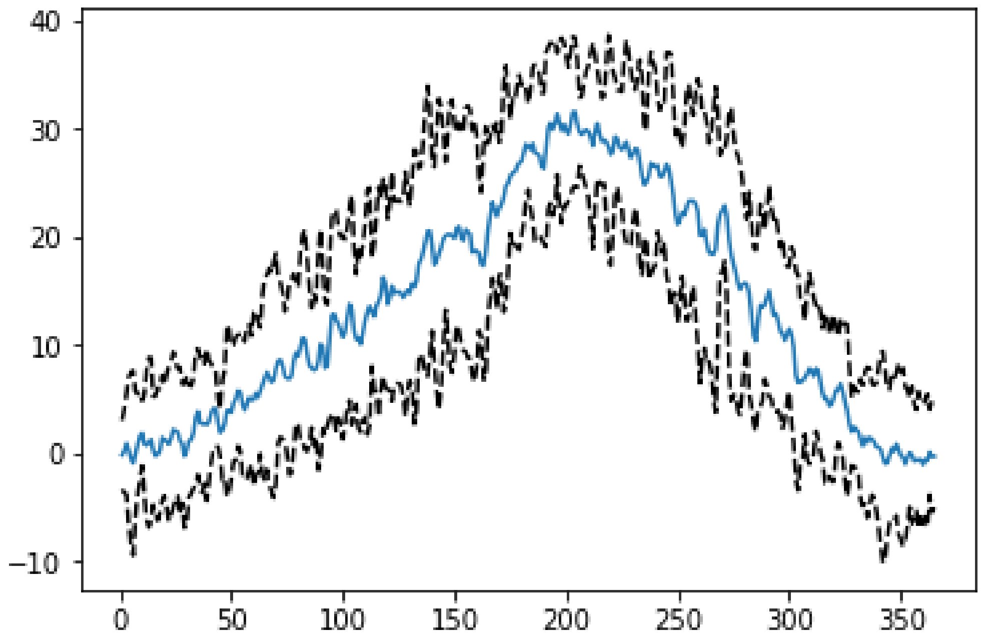
```
# do exercise here
# ANSWER

meanval = tmax.mean(axis=1)
stdval = tmax.std(axis=1)

plt.plot(year[:,0],meanval)
plt.plot(year[:,0],meanval+1.96*stdval,'k--')
plt.plot(year[:,0],meanval-1.96*stdval,'k--')
```

```
[<matplotlib.lines.Line2D at 0x1230c6550>]
```



```
# do exercise here
# ANSWER

meanval = tmax.mean(axis=0)
stdval = tmax.std(axis=0)

plt.plot(doy[0],meanval)
plt.plot(doy[0],meanval+1.96*stdval,'k--')
plt.plot(doy[0],meanval-1.96*stdval,'k--')
```

```
[<matplotlib.lines.Line2D at 0x12398a588>]
```



2.5.4 Homework

Exercise E2.5.4

Select 4 locations in different regions of North America (e.g. Anchorage, Albuquerque, Seattle, Chicago). Request data on maximum temperature, precipitation and incident solar radiation for the years between 1981 to 2010, and plot in 3 different figures:

- Figure 1: The *mean daily temperature* and the variation (a shaded area around the mean going from mean value **minus** 1.96 times the standard deviation to mean value **plus** 1.96 times the standard deviation). Use a subplot or panel for each site
- Figure 2: The *mean daily precipitation* and the variation (a shaded area around the mean going from mean value **minus** 1.96 times the standard deviation to mean value **plus** 1.96 times the standard deviation). Use a subplot or panel for each site
- Figure 3: The *mean daily incident solar radiation* and the variation (a shaded area around the mean going from mean value **minus** 1.96 times the standard deviation to mean value **plus** 1.96 times the standard deviation). Use a subplot or panel for each site

In each plot, the mean value should be a full line, and the variation should be an envelope, visually similar to the plot shown below (clearly not identical!!!!)

Label each plot with a title, units and so on. Some useful functions to consider

- `` `plt.subplots `` <https://matplotlib.org/api/_as_gen/matplotlib.pyplot.subplots.html>'__ Allows you to split a figure into several panels or subplots. In particular, pay attention to the `sharex` and `sharey` options that allow you to have the same scales for all plots so they can be directly compared.
- `` `plt.fill_between `` <https://matplotlib.org/api/_as_gen/matplotlib.pyplot.fill_between.html>'__ Allows you to fill the space between two curves. You may want to give the option `color=0.8` for a nice grey effect.

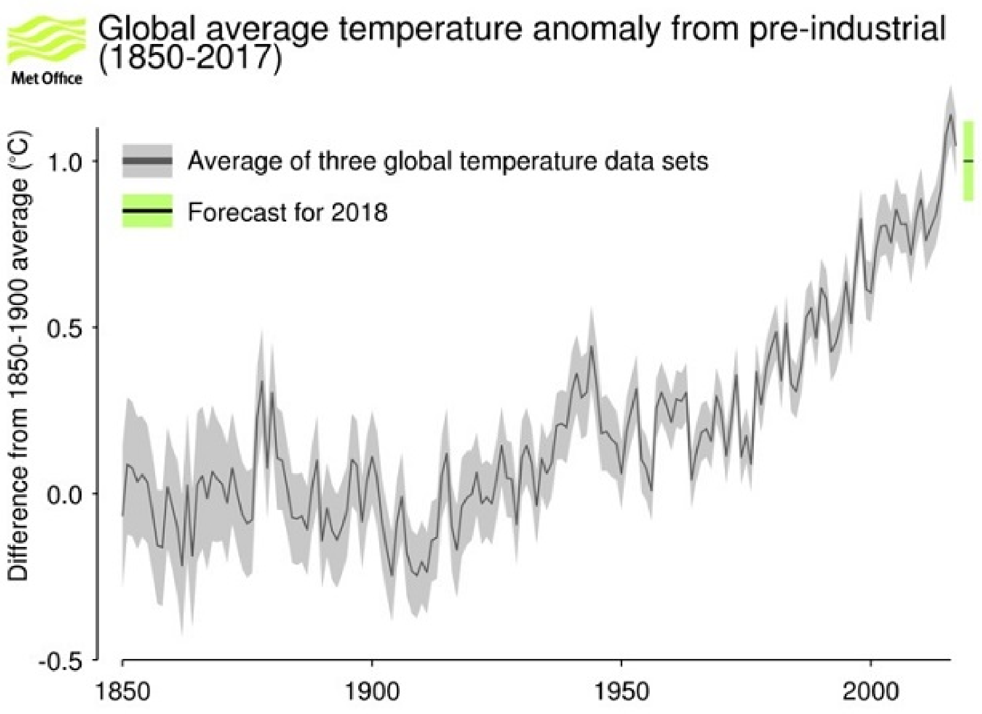


Fig. 3: la niña plot

```
# do exercise here
# ANSWER
import requests
import matplotlib.pylab as plt
import numpy as np
from io import StringIO
%matplotlib inline

locs = ['Anchorage', 'Albuquerque', 'Seattle', 'Chicago']
lats = [61.2181,      35.0844,       47.6062,    41.8781]
lons = [-149.9003,    -106.6504,     -122.3321, -87.6298]

axs = []
figs = []
for i in range(3):
    fig, axs1 = plt.subplots(nrows=2, ncols=2, \
                        sharex=True, sharey=False,\
                        figsize=(10, 10))
    axs.append(axs1)
    figs.append(fig)

axs = np.array(axs).flatten()
figs = np.array(figs).flatten()

txt = ['prcp (mm/day)', 'srad (W/m^2)', 'temperature / C']

for i,l in enumerate(locs):

    url = "https://daymet.ornl.gov/single-pixel/api/" + \
```

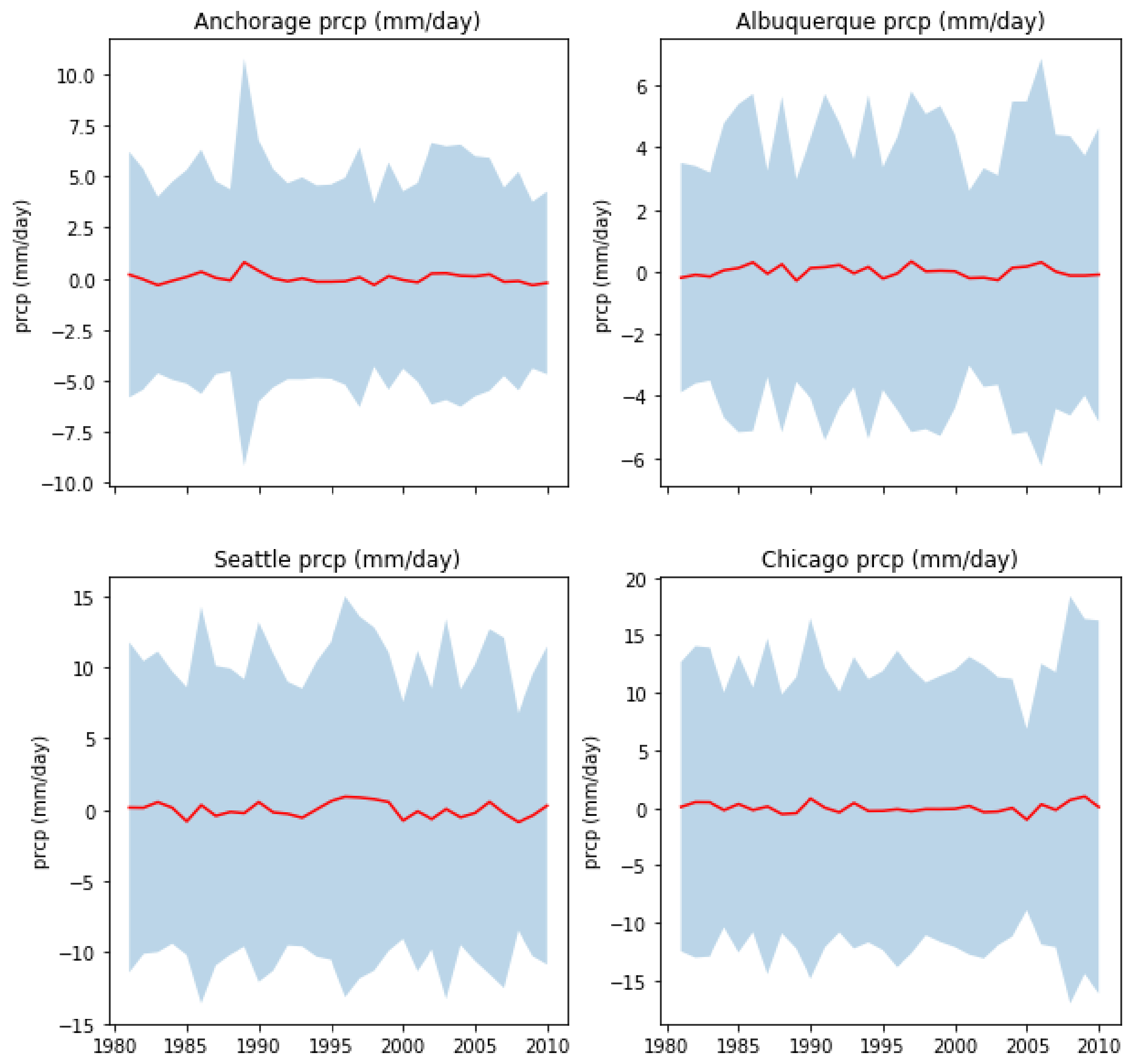
(continues on next page)

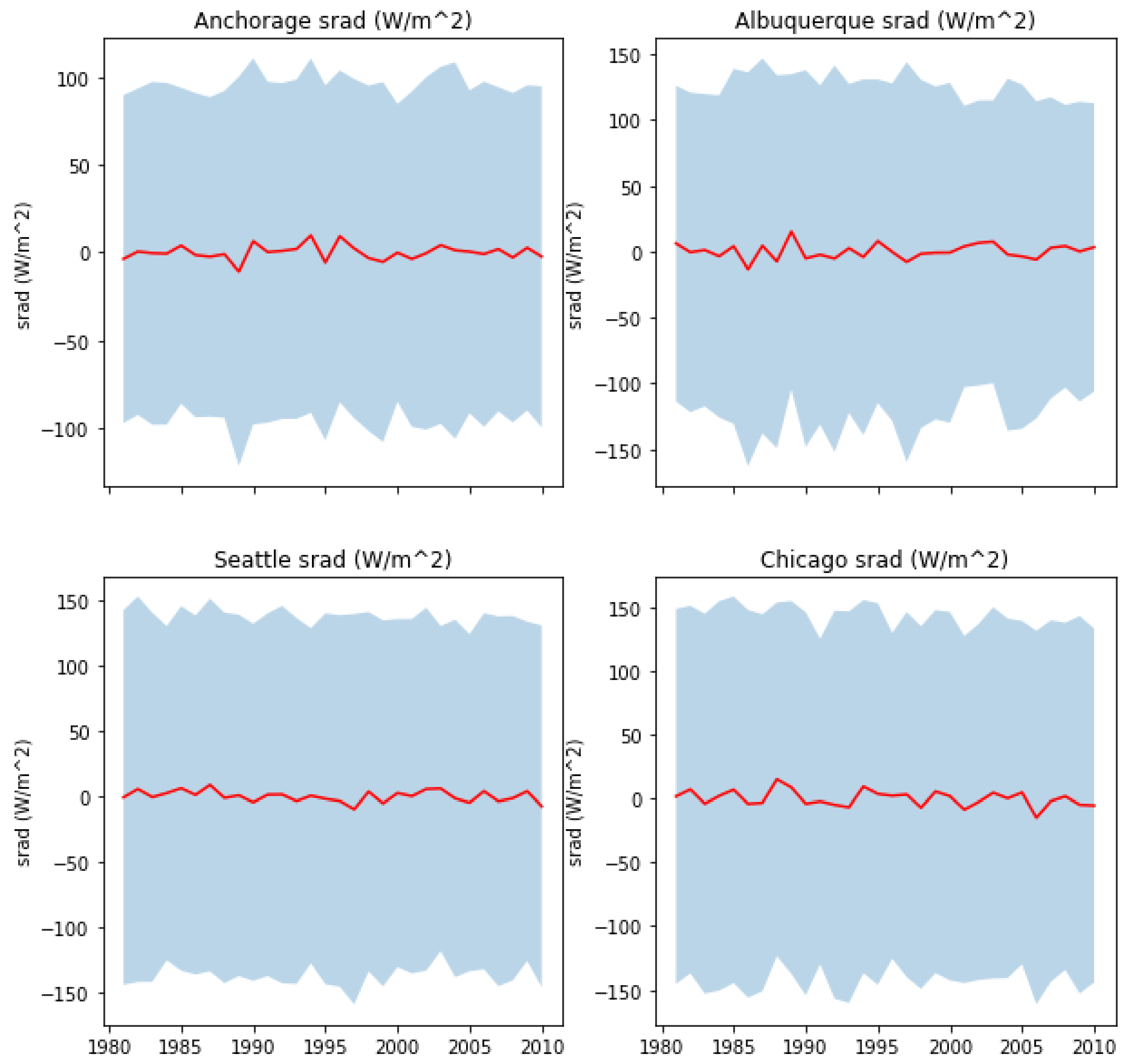
(continued from previous page)

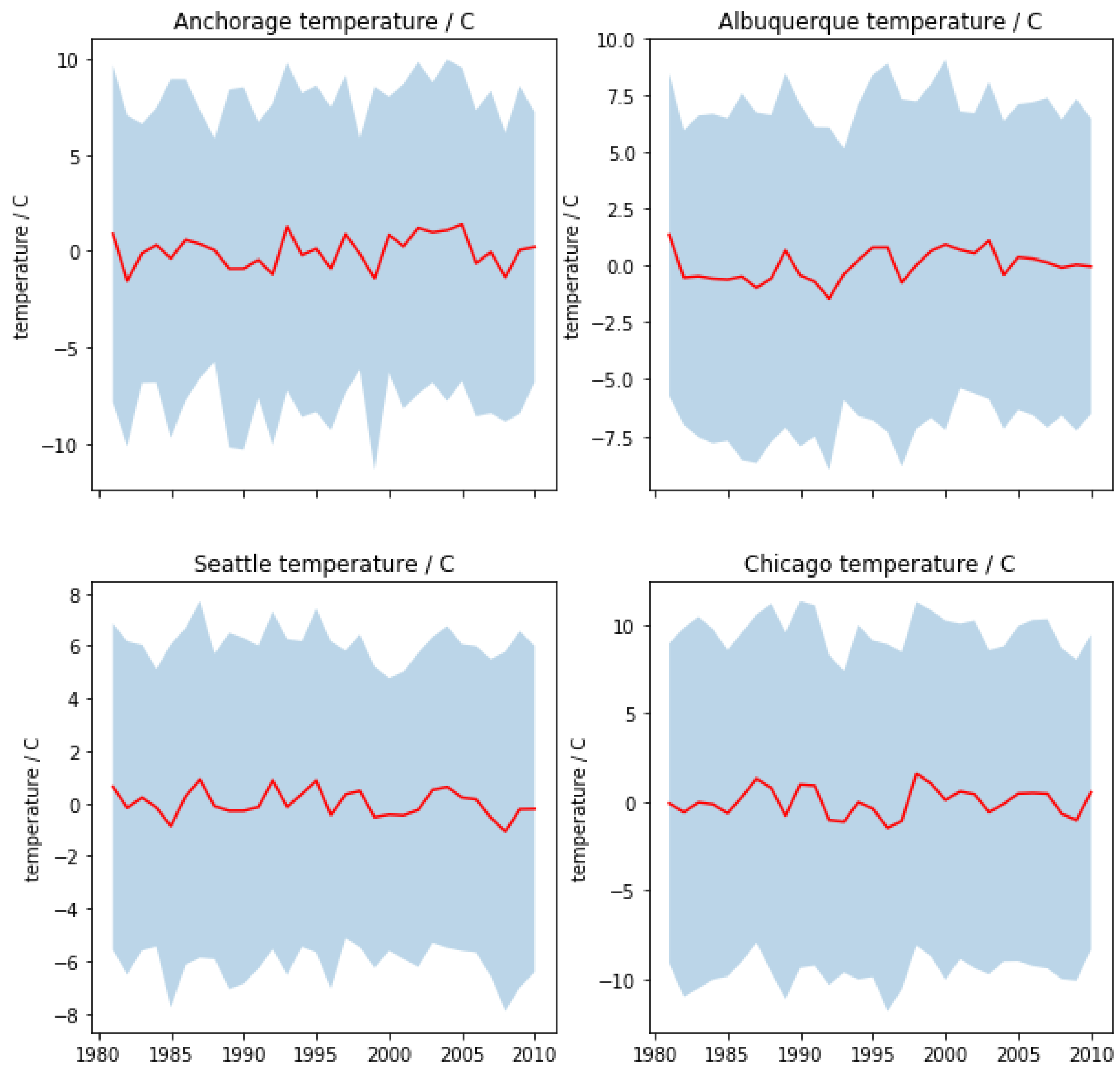
```
        f"data?lat={lats[i]}&lon={lons[i]}&vars=tmax,prcp,srad&start=1981-01-01&
↪end=2010-12-31"
    data = requests.get(url).text

    # little data size check !!!
    print (l,i,len(data))

    temperature = np.loadtxt(StringIO(data), skiprows=8, delimiter=",", unpack=True)

    nyears = int(temperature.shape[1]/365)
    year = temperature[0].reshape(nyears,365)
    doy  = temperature[1].reshape(nyears,365)


    years = (year[:,0]).astype(int)

    for j in range(3):
        vari = temperature[2+j].reshape(nyears,365)
        mean_year = vari.mean(axis=0)
        # detrended mean .... depends what you want to show
        mean = (vari - mean_year).mean(axis=1)
        var = ((vari - mean_year)**2).mean(axis=1)
        std = np.sqrt(var)
        axs[j*4+i].set_title(l+' ' + txt[j])
        axs[j*4+i].fill_between(years,mean-std*1.96,mean+std*1.96,alpha=0.3)
        axs[j*4+i].plot(years,mean,'r')
        axs[j*4+i].set_ylabel(txt[j])

'''
Need to complete this!

Lewis
'''
```

```
Anchorage 0 410793
Albuquerque 1 411666
Seattle 2 410847
Chicago 3 411264
```

```
'nNeed to complete this!nnLewisn'
```


Anchorage prcp (mm/day)
Albuquerque prcp (mm/day)
Seattle prcp (mm/day)
Chicago prcp (mm/day)
prcp (mm/day)
1980
1985
1990
1995
2000
2005
2010


Anchorage srad (W/m^2)
Albuquerque srad (W/m^2)
Seattle srad (W/m^2)
Chicago srad (W/m^2)
srad (W/m^2)
1980
1985
1990
1995
2000
2005
2010



2.5.5 Summary

In this section, we have learned about reading data from csv files from the local disc or that we have pulled from the web (given a URL). We have gone into a little more detail and sophistication on plotting graphs, and you now should be able to produce sensible plots of datasets or summaries of datasets (e.g. mean standard deviation).

2.8.6 2. Manipulating and plotting data in Python: `numpy`, and `matplotlib` libraries

Table of Contents

While Python has a rich set of modules and data types by default, for numerical computing you'll be using two main libraries that conform the backbone of the Python scientific stack. These libraries implement a great deal of functionality related to mathematical operations and efficient computations on large data volumes. These libraries are `` `numpy `` <http://numpy.org>`__ and `` `scipy `` <http://scipy.org>`__. `numpy`, which we will concentrate on in this section, deals with efficient arrays, similar to lists, that simplify many common processing operations. Of course, just doing calculations isn't much fun if you can't plot some results. To do this, we use the `` `matplotlib `` <http://matplotlib.org>'__ library.

But first, we'll see the concept of *functions*. . . .

2.1 Functions

A function is a collection of Python statements that do something (usually on some data). For example, you may want to convert from Fahrenheit to Centigrade. The conversion is

$$^\circ C = (^\circ F - 32) \cdot \frac{5}{9}$$

A Python function will have a name (and we hope that the name is self-explanatory as to what the function does), and a set of input parameters. In the case above, the function would look like this:

```
def fahrenheit_to_centigrade(deg_fahrenheit):
    """A function to convert from degrees Fahrenheit to degrees Centigrade

    Parameters
    ----------
    deg_fahrenheit: float
        Temperature in degrees F

    Returns
    -------
    Temperature converted to degrees C
    """
    deg_c = (deg_fahrenheit - 32.)*5./9.
    return deg_c
```

We see that the function has a name (`fahrenheit_to_centigrade`), and takes one parameter (`deg_fahrenheit`).

The main body of the function is indented (like `if` and `for` statements). There is first a comment string, that describes what the function does, as well as what the inputs are, and what the output is. This is just useful documentation of the code.

The main body of the function calculates `deg_C` from the given input, and **returns** it back to the user.

Notice that the document string `"""A function to convert from temperature... """` is what is printed when you request `help` on the function:

```
help(fahrenheit_to_centigrade)
```

```
Help on function fahrenheit_to_centigrade in module __main__:

fahrenheit_to_centigrade(deg_fahrenheit)
    A function to convert from degrees Fahrenheit to degrees Centigrade

    Parameters
    ----------
    deg_fahrenheit: float
        Temperature in degrees F

    Returns
    -------
    Temperature converted to degrees C
```

E2.1.1 Exercise

- In the vein of converting units, write functions that convert from
  - inches to m (and back)
  - kg to stones (and back)

Hint: A stone is equal to 14 pounds, and a pound is equal to 0.45359237 kg.

Ensure that your functions are clearly named, have sensible variable names, a brief docmentation string, and remember to test the functions work: just demonstrate running the function with some input pairs where you know the output and checking it makese sense.

```
# Space for your solution
```

2.2 numpy

2.2.1 arrays

You import the `numpy` library using

```
import numpy as np
```

This means that all the functionality of `numpy` is accessed by the prefix `np.`: e.g. `np.array`. The main element of `numpy` is the numpy array. An array is like a list, but unlike a list, all the elements are of the same type, floating point numbers for example.

Let's see some arrays in action...

```
import numpy as np  # Import the numpy library

# An array with 5 ones
arr = np.ones(5)
print(arr)
print(type(arr))

# An array started from a list of **integers**
arr = np.array([1, 2, 3, 4])
print(arr)

# An array started from a list of numbers, what's the difference??
arr = np.array([1., 2, 3, 4])
print(arr)
```

```
[1. 1. 1. 1. 1.]
<class 'numpy.ndarray'>
[1 2 3 4]
[1. 2. 3. 4.]
```

In the example above we have generated an array where all the elements are `1.0`, using `` `np.ones `` <https://docs.scipy.org/doc/numpy/reference/generated/numpy.ones.html>'__, and then we have been able to generate arrays from lists using the `` `np.array `` <https://docs.scipy.org/doc/numpy/reference/generated/numpy.array.html>'__ function. The difference between the 2nd and 3rd examples is that in the 2nd example, all the elements of the list are integers, and in the 3rd example, one is a floating point number. `numpy` automatically makes the array floating point by converting the integers to floating point numbers.

What can we do with arrays? We can efficiently operate on individual elements without loops:

```
arr = np.ones(10)
print(2 * arr)
```

```
[2. 2. 2. 2. 2. 2. 2. 2. 2. 2.]
```

numpy is clever enough to figure out that the 2 multiplying the array is applied to all elements of the array, and returns an array of the same size as arr with the elements of arr multiplied by 2. We can also multiply two arrays of the same size. So let's create an array with the numbers 0 to 9 and one with the numbers 9 to 0 and do a times table:

```
arr1 = 9 * np.ones(10)
arr2 = np.arange(1, 11)  # arange gives an array from 1 to 11, 11 not included

print(arr1)
print(arr2)

print(arr1 * arr2)
```

```
[9. 9. 9. 9. 9. 9. 9. 9. 9. 9.]
[ 1  2  3  4  5  6  7  8  9 10]
[ 9. 18. 27. 36. 45. 54. 63. 72. 81. 90.]
```

E2.2.1 Exercise

- Using code similar to the above and a for loop, write the times tables for 2 to 10. The solution you're looking for should look a bit like this:

```
[ 2  4  6  8 10 12 14 16 18 20]
[ 3  6  9 12 15 18 21 24 27 30]
[ 4  8 12 16 20 24 28 32 36 40]
[ 5 10 15 20 25 30 35 40 45 50]
[ 6 12 18 24 30 36 42 48 54 60]
[ 7 14 21 28 35 42 49 56 63 70]
[ 8 16 24 32 40 48 56 64 72 80]
[ 9 18 27 36 45 54 63 72 81 90]
[ 10  20  30  40  50  60  70  80  90 100]
```

```
# Your solution here
```

If the arrays are of the same *shape*, you can do standard operations between them **element-wise**:

```
arr1 = np.array([3, 4, 5, 6.])
arr2 = np.array([30, 40, 50, 60.])

print(arr2 - arr1)
print(arr1 * arr2)

print("Array shapes:")
print("arr1: ", arr1.shape)
print("arr2: ", arr2.shape)
```

```
[27. 36. 45. 54.]
[ 90. 160. 250. 360.]
Array shapes:
arr1:  (4,)
arr2:  (4,)
```

The numpy documenation is huge. There's an user's guide, as well as a reference to all the contents of the library. There's even a tutorial availabe if you get bored with this one.

2.2.2 More detail about numpy.arrays

So far, we have seen a 1D array, which is the equivalent to a vector. But arrays can have more dimensions: a 2D array would be equivalent to a matrix (or an image, with rows and columns), and a 3D array would be a volume split into voxels, as seen below

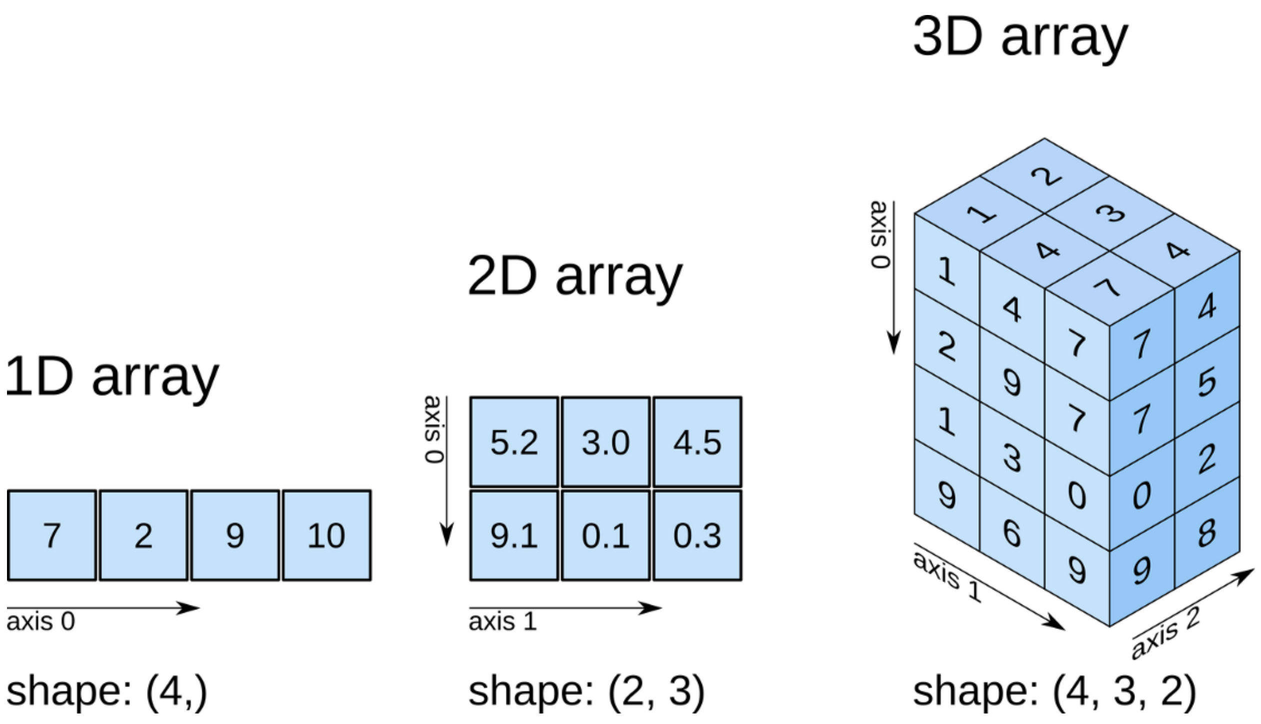


Fig. 4: numpy arrays

So a 1D array has one axis, a 2D array has 2 axes, a 3D array 3, and so on. The shape of the array provides a tuple with the number of elements along each axis. Let's see this with some generally useful array creation options:

```
# Create a 2D array from a list of rows. Note that the 3 rows have the same number of␣
↪elements!
arr1 = np.array([[0, 1, 2, 3, 4], [5, 6, 7, 8, 9], [10, 11, 12, 13, 14]])
# A 2D array from a list of tuples.
# We're specifically asking for floating point numbers
arr2 = np.array([(1.5, 2, 3), (4, 5, 6)], dtype=np.float)
print("3*5 array:")
print(arr1)
print("2*3 array:")
print(arr2)
```

```
3*5 array:
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
2*3 array:
```

(continues on next page)

(continued from previous page)

```
[[1.5 2.  3. ]
 [4.  5.  6. ]]
```

2.2.3 Array creators

Quite often, we will want to initialise an array to be all the same number. The methods for doing this as 0,1 and unspecified in `numpy` are `np.zeros()`, `np.ones()`, `np.empty()` respectively.

```
# Creates a 3*4 array of 0s
arr = np.zeros((3, 4))
print("3*4 array of 0s")
print(arr)

# Creates a 2x3x4 array of int 1's
print("2*3*4 array of 1s (integers)")
arr = np.ones((2, 3, 4), dtype=np.int)
print(arr)

# Creates an empty (e.g. uninitialised) 2x3 array. Elements are random
print("2*3 empty array (contents could be anything)")
arr = np.empty((2, 3))
print(arr)
```

```
3*4 array of 0s
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
2*3*4 array of 1s (integers)
[[[1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]]

 [[1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]]]
2*3 empty array (contents could be anything)
[[1.5 2.  3. ]
 [4.  5.  6. ]]
```

Exercise E2.2.2

- write a function that does the following:
  - create a 2-D tuple called `indices` containing the integers `((0, 1, 2, 3, 4),(5, 6, 7, 8, 9))`
  - create a 2-D numpy array called `data` of shape `(5,10)`, data type `int`, initialised with zero
  - set the value of `data[r,c]` to be `1` for each of the 5 row,column pairs specified in `indices`.
  - return the data array
- print out the result returned

The result should look like:

```
[[0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 0 1]]
```

Hint: You could use a `for` loop, but what does `data[indices]` give?

```
# do exercise here
```

Exercise 2.2.3

- write a more flexible version of you function above where `indices`, the value you want to set (`1` above) and the desired shape of `data` are specified through function keyword arguments (e.g. `indices=((0, 1, 2, 3, 4),(5, 6, 7, 8, 9)),value=1,shape=(5,10))`

```
# do exercise here
```

As well as initialising arrays with the same number as above, we often also want to initialise with common data patterns. This includes simple integer ranges `(start, stop, skip)` in a similar fashion to slicing in the last session, or variations on this theme:

```
### array creators

print("1D array of numbers from 0 to 2 in increments of 0.3")
start = 0
stop  = 2.0
skip  = 0.3

arr = np.arange(start,stop,skip)
print(f'arr of shape {arr.shape}:\n\t{arr}')

start = 0
stop  = 34
nsamp = 9
arr = np.linspace(start,stop,nsamp)
print(f"array of shape {arr.shape} numbers equally spaced from {start} to {stop}:\n\t
↪{arr}")

np.linspace(stop,start,9)
```

```
1D array of numbers from 0 to 2 in increments of 0.3
arr of shape (7,):
    [0.  0.3 0.6 0.9 1.2 1.5 1.8]
array of shape (9,) numbers equally spaced from 0 to 34:
    [ 0.    4.25  8.5  12.75 17.   21.25 25.5  29.75 34.  ]
```

```
array([34.  , 29.75, 25.5 , 21.25, 17.  , 12.75,  8.5 ,  4.25,  0.  ])
```

Exercise E2.2.4

- print an array of integer numbers from 100 to 1
- print an array with 9 numbers equally spaced between 100 and 1

Hint: what value of skip would be appropriate here? what about `start` and `stop`?

```
# do exercise here
```

2.2.4 Summary statistics

Below are some typical arithmetic operations that you can use on arrays. Remember that they happen **elementwise** (i.e. to the whole array).

```
b = np.arange(4)
print(f'{b}^2 = {b**2}\n')

a = np.array([20, 30, 40, 50])
print(f"assuming in radians,\n10*sin({a}) = {10 * np.sin(a)}")

print("\nSome useful numpy array methods for summary statistics...\n")
print("Find the maximum of an array: a.max(): ", a.max())
print("Find the minimum of an array: a.min(): ", a.min())
print("Find the sum of an array: a.sum(): ", a.sum())
print("Find the mean of an array: a.mean(): ", a.mean())
print("Find the standard deviation of an array: a.std(): ", a.std())
```

```
[0 1 2 3]^2 = [0 1 4 9]

assuming in radians,
10*sin([20 30 40 50]) = [ 9.12945251 -9.88031624  7.4511316  -2.62374854]

Some useful numpy array methods for summary statistics...

Find the maximum of an array: a.max():  50
Find the minimum of an array: a.min():  20
Find the sum of an array: a.sum():  140
Find the mean of an array: a.mean():  35.0
Find the standard deviation of an array: a.std():  11.180339887498949
```

Let's access an interesting dataset on the frequency of satellite launches to illustrate this.

Fig. 5: SpaceX landing

```
from geog0111.nsat import nsat

'''
This dataset gives the number of
satellites launched per month and year
data from https://www.n2yo.com
'''
# We use the code supplied in nsat.py
# to generate the dataset (takes time)
# or to load it if it exists
data,years = nsat().data,nsat().years

print(f'data shape {data.shape}')

print(f'some summary statistics over the period {years[0]} to {years[1]}:')
print(f'The total number of launches is {data.sum():d}')
print(f'The mean number of launches is {data.mean():.3f} per month')
```

```
data shape (12, 62)
some summary statistics over the period 1957 to 2019:
The total number of launches is 43611
The mean number of launches is 58.617 per month
```

Exercise E2.2.5

- copy the code above but generate a fuller set of summary statistics including the standard deviation, minimum and maximum.

```
# do exercise here
```

Whilst we have generated some interesting summary statistics on the dataset, it's not really enough to give us a good idea of the data characteristics.

To do that, we want to be able to ask somewhat more complex questions of the data, such as, which *year* has the most/least launches? which month do most launches happen in? which month in which year had the most launches? which years had more than 100 launches?

To be able to address these, we need some new concepts:

- methods `argmin()` and `argmax()` that provide the *index* where the min/max occurs
- filtering and the related method `where()`
- `axis` methods: the dataset is two-dimensional, and for some questions we need to operate only over one of these

To illustrate:

```
from geog0111.nsat import nsat
import numpy as np

data,years = nsat().data,nsat().years

year = np.arange(years[0],years[1],dtype=np.int)

# sum the data over all months (axis 0)
sum_per_year = data.sum(axis=0)

imax = np.argmax(sum_per_year)
imin = np.argmin(sum_per_year)

# filtering
# high(low) is an array set to True where the condition
# is True, and False otherwise
high = sum_per_year>=1000
low  = sum_per_year<=300

print(f'the year with most launches was {year[imax]} with {sum_per_year[imax]}')
print(f'the year with fewest launches was {year[imin]} with {sum_per_year[imin]}')

print('\nThe years with >= 1000 launches are:')
print(year[high],'\nvalues:\n',sum_per_year[high])
print('The years with <= 300 launches are:')
print(year[low],'\nvalues:\n',sum_per_year[low])
```

```
the year with most launches was 1999 with 4195
the year with fewest launches was 1957 with 3
```

(continues on next page)

(continued from previous page)

```

The years with >= 1000 launches are:
[1965 1975 1976 1981 1986 1987 1993 1994 1999 2006]
values:
 [1527 1195 1264 1190 1375 1130 2131 1166 4195 1158]
The years with <= 300 launches are:
[1957 1958 1959 1960 1962 1996 2002 2003 2004 2005]
values:
 [  3  11  22  52 207 246 277 243 209 192]
```

Exercise E2.2.6

- copy the code above, and modify it to find the total launches *per month* (over all years)
- show these data in a table
- which month do launches mostly take place in? which month do launches most seldom take place in?

```
# do exercise here
```

The form of filtering above (`high = sum_per_year>=1000`) produces a numpy array of the same shape as that operated on (`sum_per_year` here) of `bool` data type. It has entries of `True` where the condition is met, and `False` where it is not met.

```
from geog0111.nsat import nsat
# sum the data over all months (axis 0)
sum_per_year = nsat().data.sum(axis=0)

high = sum_per_year>=1000
low  = sum_per_year<=300

print(f'type(sum_per_year): {type(sum_per_year)}, sum_per_year.shape: {sum_per_year.
↪shape}, ' \
        + f'sum_per_year.dtype: {sum_per_year.dtype}')
print(f'type(high): {type(high)}, high.shape: {high.shape}, high.dtype: {high.dtype}\n
↪')

print(f'sum_per_year: {sum_per_year}')
print(f'high: {high}')
print(f'low: {low}')
```

```
type(sum_per_year): <class 'numpy.ndarray'>, sum_per_year.shape: (62,), sum_per_year.
↪dtype: int64
type(high): <class 'numpy.ndarray'>, high.shape: (62,), high.dtype: bool

sum_per_year: [   3   11   22   52  396  207  346  401 1527  786  466  690  641  906
  636  654  875  694 1195 1264  891  783  857  637 1190  946  884  760
  788 1375 1130  814  950  691  691  740 2131 1166  534  246  960  651
 4195  730  582  277  243  209  192 1158  349  406  378  373  315  435
  352  355  335  308  512  320]
high: [False False False False False False False False  True False False False
 False False False False False False  True  True False False False False
  True False False False False  True  True False False False False False
  True  True False False False False  True False False False False False
 False  True False False False False False False False False False False
 False False]
low: [ True  True  True  True False  True False False False False False False
```

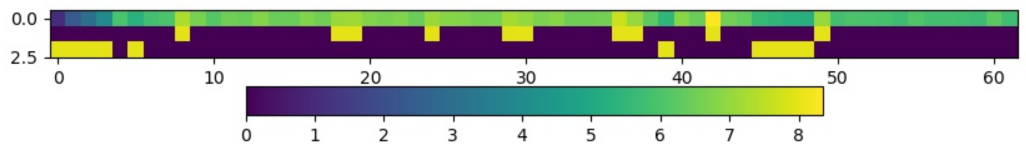
(continues on next page)

(continued from previous page)

```
False False False False False False False False False False False False
False False False False False False False False False False False False
False False False  True False False False False False  True  True  True
 True False False False False False False False False False False False
False False]
```

We can think of this logical array as a 'data mask' that we use to select (filter) entries.

The figure shows `log(sum_per_year)` in the top line of the image (numbers represented by colour shown in colourbar), then a representation of the `bool` arrays `high` and `low`. Where the `bool` value is shown yellow, the 'data mask' is true.



```
print(f'{sum_per_year[high]}')
print(f'{sum_per_year[low]}')
```

```
[1527 1195 1264 1190 1375 1130 2131 1166 4195 1158]
[   3   11   22   52  207  246  277  243  209  192]
```

Sometimes, instead of just applying the filter as above, we want to know the indices of the filtered values.

To do this, we can use the `np.where()` method. This takes a `bool` array as its argument (such as our data masks or other conditions) and returns a tuple of the indices where this is set `True`.

```
from geog0111.nsat import nsat
data,years = nsat().data,nsat().years
# where :
# which months in the dataset were particularly busy ..
# we select data > 400 as a condition

indices = np.where(data > 400)
print(f'indices:\n{indices[0]}\n{indices[1]}')
print(f'\ntype(indices): {type(indices)}')
print(f'len(indices): {len(indices)}, len(indices[0]): {len(indices[0])}')
print(f'type(indices[0][0]): {type(indices[0][0])}')

year = np.arange(years[0],years[1],dtype=np.int)
month = np.arange(12)

nsamp = len(indices[0])

# loop over the entries in the tuple
print('*'*23)
print('busy months')
print('*'*23)
for i in range(nsamp):
    print(f'{i:04d} month {month[indices[0][i]]:02d}'+\
                  f' year {year[indices[1][i]]:04d}')
print('*'*23)
```

```
indices:
[1 3 4 4 5 5 5 6 8 8 9 9]
[29 13 37 42 24 36 49 19 40 43  8 42]

type(indices): <class 'tuple'>
len(indices): 2, len(indices[0]): 12
type(indices[0][0]): <class 'numpy.int64'>
*******************
busy months
*******************
0000 month 01 year 1986
0001 month 03 year 1970
0002 month 04 year 1994
0003 month 04 year 1999
0004 month 05 year 1981
0005 month 05 year 1993
0006 month 05 year 2006
0007 month 06 year 1976
0008 month 08 year 1997
0009 month 08 year 2000
0010 month 09 year 1965
0011 month 09 year 1999
*******************
```

Exercise E2.2.7

- Using code from the sections above, print out a table with the busiest launch months with an additional column stating the number of launches

Hint: this is just adding another column to the print statement in the for loop

```
# do exercise here
```

You might notice the indices in the tuple derived above using `where` are *ordered*, but the effect of this is that the months are in sequential order, rather than the years. We have

```
month[indices[0][i]]
year[indices[1][i]]
```

If we want to put the data in year order, there are several ways we could go about this. An insteresting one, following the ideas in `argmax()` and `argmin()` above is to use `argsort()`. This gives the *indices* of the sorted array, rather than the values.

So here, we can find the *indices* of the year-sorted array, and apply them to both `month` and `year` datasets:

```
# prepare data as above
from geog0111.nsat import nsat
data,years = nsat().data,nsat().years
indices = np.where(data > 400)
year = np.arange(years[0],years[1],dtype=np.int)
month = np.arange(12,dtype=np.int)

# store the months and years
# in their unsorted (original) form
unsorted_months = month[indices[0]]
unsorted_years = year[indices[1]]
print(f'years not in order: {unsorted_years}')
```

(continues on next page)

(continued from previous page)

```
print(f'but months are:     {unsorted_months}\n')


# get the indices to put years in order
year_order = np.argsort(indices[1])

# apply this to months and years
print(f'year order: {year_order}\n')
print(f'years in order: {unsorted_years[year_order]}')
print(f'months in year order: {unsorted_months[year_order]}')
```

```
years not in order: [1986 1970 1994 1999 1981 1993 2006 1976 1997 2000 1965 1999]
but months are:     [1 3 4 4 5 5 5 6 8 8 9 9]

year order: [10  1  7  4  0  5  2  8  3 11  9  6]

years in order: [1965 1970 1976 1981 1986 1993 1994 1997 1999 1999 2000 2006]
months in year order: [9 3 6 5 1 5 4 8 4 9 8 5]
```

Exercise E2.2.8

- Use this example of `argsort()` to redo Exercise E2.2.7, putting the data in correct year order

```
# do exercise here
```

2.2.5 Summary

In this section, you have been introduced to more detail on arrays in `numpy`. The big advantages of `numpy` are that you can easily perform array operators (such as adding two arrays together), and that `numpy` has a large number of useful functions for manipulating N-dimensional data in array form. This makes it particularly appropriate for raster geospatial data processing.

We have seen how to create various forms of array (e.g. `np.ones()`, `np.arange()`), how to calculate some basic statistics (`min()`, `max()` etc), and finding the array index where some pattern occurs (e.g. `argmin()`, `argsort()` or `where()`).

2.3 Plotting with Matplotlib

There are quite a few graphical libraries for Python, but matplotlib is probably the most famous one. It does pretty much all you need in terms of 2D plots, and simple 3D plots, and is fairly straightforward to use. Have a look at the matplotlib gallery for a fairly comprehensive list of examples of what the library can do as well as the code that was used in the examples.

Importing matplotlib

You can import matplotlib with

```
import matplotlib.pyplot as plt
```

As with `numpy`, it's custom to use the `plt` prefix to call matplotlib commands. In the notebook, you should also issue the following command just after the import

```
%matplotlib notebook
```

or

```
%matplotlib inline
```

The former command will make the plots in the notebook interactive (i.e. point-and-click-ey), and the second will just stick the plots into the notebook as PNG files.

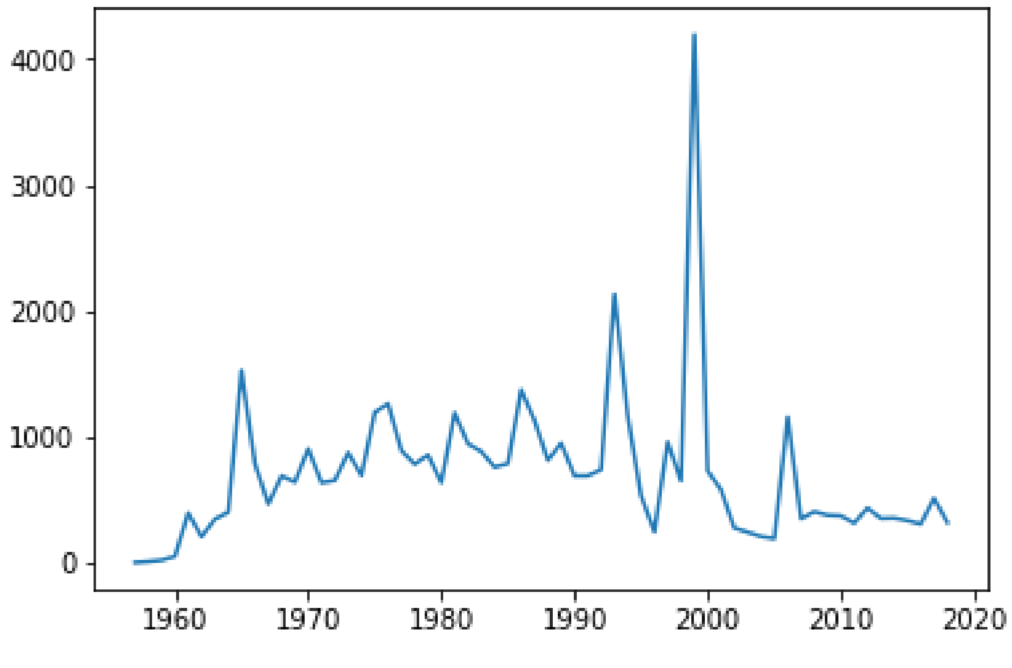
Simple 2D plots

The most basic plots are 2D plots (e.g. x and y).

```
import matplotlib.pyplot as plt
%matplotlib inline

from geog0111.nsat import nsat
import numpy as np
'''
This dataset gives the number of
satellites launched per month and year

data from https://www.n2yo.com
'''
data,years = nsat().data,nsat().years
year = np.arange(years[0],years[1],dtype=np.int)
# sum the data over all months (axis 0)
sum_per_year = data.sum(axis=0)

print(f'data shape {data.shape}')

# plot x as year
# plot y as the number of satellites per year
plt.plot(year,sum_per_year,label='launches per year')
```

```
data shape (12, 62)
```

```
[<matplotlib.lines.Line2D at 0x1168270b8>]
```



Whilst this plot is fine, there are a few simple things we could do improve it.

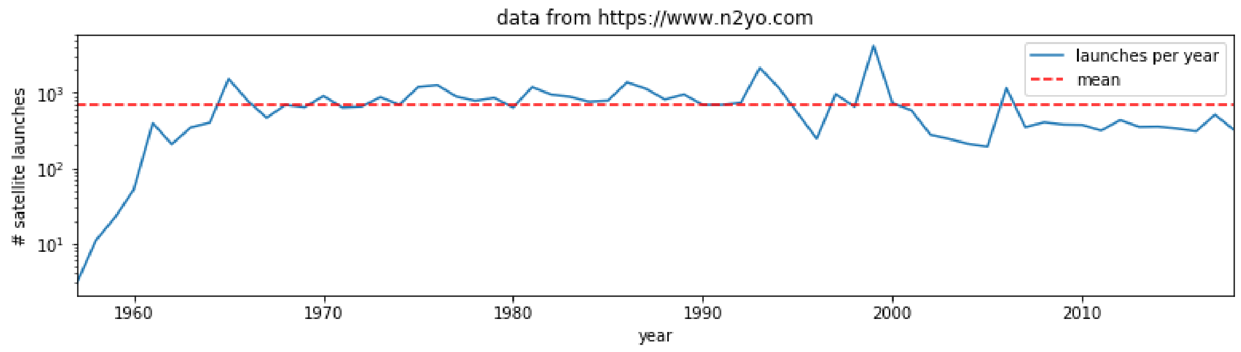
We will go through some of the options below, but to get a taste of improved ploitting, lets use e.g.:

- reset the image shape/size
  - `plt.figure(figsize=(13,3))`
- plot the mean value (as a red dashed line) for comparison
  - `plt.plot([year[0],year[-1]],[mean,mean],'r--',label='mean')`
- limit the dataset to range of variable `year`
  - `plt.xlim(year[0],year[-1])`
- put labels on the x and y axes
  - `plt.xlabel('year')`
  - `plt.ylabel('# satellite launches')`
- set a title
  - `plt.title('data from https://www.n2yo.com')`
- use a legend (in conjunction with `label=` using `plot`)
  - `plt.legend(loc='best')`
- use a log scale in the y-axis
  - `plt.semilogy()`

What you choose to do will depend on what you want to show on the graph, but the examples above are quite common.

```
import matplotlib.pyplot as plt
%matplotlib inline
from geog0111.nsat import nsat
import numpy as np
'''data as above'''
data,years = nsat().data,nsat().years
year = np.arange(years[0],years[1],dtype=np.int)
sum_per_year = data.sum(axis=0)

# calculate mean of sum_per_year
mean = sum_per_year.mean()

plt.figure(figsize=(13,3))
plt.plot(year,sum_per_year,label='launches per year')
plt.plot([year[0],year[-1]],[mean,mean],'r--',label='mean')
plt.xlim(year[0],year[-1])
plt.xlabel('year')
plt.ylabel('# satellite launches')
plt.title('data from https://www.n2yo.com')
plt.legend(loc='best')
plt.semilogy()
```

```
[]
```



Exercise E2.3.1

- produce a plot showing launches per year as a function of year, showing data for selected months individually.

Hint: do a simple plot first, then add some improvements gradually. You might set up a list of months to process and use a loop to go over each month.

```
# do exercise here
```

Exercise 2.3.2

Putting together some ideas from above to look at some turning points in a function:

- generate a numpy array called `x` with 100 equally spaced numbers between 0 and 5
- generate a numpy array called `y` which contains $x^3 - 9x^2 + 26x - 24$
- plot `y` as a function of `x` with a red line
- plot **only positive** values of `y` (as a function of `x`) with a green line

Hint: to plot with red and green line `plot(x,y,'r')` and `plot(x,y,'g')`

```
# do exercise here
```

2.4 Indexing and slicing arrays

2.4.1 Recap

Selecting different elements of the array to operate in them is a very common task. `numpy` has a very rich syntax for selecting different bits of the array. We have come across slicing before, but it is so important to array processing, we will go over some of it again.

Similar to lists, you can refer to elements in the array by their position. You can also use the `:` symbol to specify a range (a **slice**) of positions `first_element:(last_element+1`. If you want to start counting from the end of the array, use negative numbers: `-1` refers to the last element of the array, `-2` the one before last and so on. In a slice, you can also specify a step as the third element in `first_element:(last_element+1:step`. If the step is negative you count from the back.

All this probably appears mind bogging, but it's easier shown in practice. You'll get used to it quite quickly once you start using it

```
import numpy as np

a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print(a[2])       # 2
```

(continues on next page)

(continued from previous page)

```
print(a[2:5])     # [2, 3, 4]
print(a[-1])      # 10
print(a[:8])      # [0, 1, 2, 3, 4, 5, 6, 7]
print(a[2:])      # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(a[5:2:-1]) # [5, 4, 3]
```

```
2
[2 3 4]
10
[0 1 2 3 4 5 6 7]
[ 2  3  4  5  6  7  8  9 10]
[5 4 3]
```

The concept extends cleanly to multidimensional arrays. . .

```
b = np.array([[0, 1, 2, 3], [10, 11, 12, 13], [20, 21, 22, 23], [30, 31, 32, 33],
     [40, 41, 42, 43]])

print(b[2, 3])     # 23
print(b[0:5, 1])  # each row in the second column of b
print(b[:, 1])     # same thing as above
print(b[1:3, :])  # each column in the second and third row of b
```

```
23
[ 1 11 21 31 41]
[ 1 11 21 31 41]
[[10 11 12 13]
 [20 21 22 23]]
```

Exercise 2.4.1

- generate a 2-D numpy array of integer zeros called `x`, of shape (7,7)
- we can think of this as a square. Set the central 3 by 3 samples of the square to one
- print the result

Hint: Don't use looping, instead work out how to define the slice of the central 3 x 3 samples.

```
# do exercise here
```

2.4.1 data mask

A useful way to select elements is by using what's called a mask as we saw above: an array of logical (boolean) elements that only selects the elements that are `True`:

```
a = np.arange(10)
select_me = a >= 7
print(a[select_me])
```

```
[7 8 9]
```

The previous point also shows something interesting: you can apply comparisons element by element. So in the previous example, `select_me` is a 10 element array where all the elements of `a` that are equal or higher than 7 are set to True.

If you want to build up element by element logical operations, it's best to use specialised functions like `` `np.logical_and `` <https://docs.scipy.org/doc/numpy/reference/generated/numpy.logical_and.html>'__ and friends

```
a = np.arange(100)
sel1 = a > 45
sel2 = a < 73
print(a[np.logical_and(sel1, sel2)])
```

```
[46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69
 70 71 72]
```

Exercise 2.4.2

- generate a numpy array called `x` with 100 equally spaced numbers between 0 and 5
- generate a numpy array called `y` which contains $x^3 - 9x^2 + 26x - 24$
- print the values of `x` for which `y` is greater than or equal to zero and `x` lies between 3.5 and 4.5

```
# do exercise here
```

2.5 Reading data

2.5.1 `np.loadtxt`

It's a bit tedious just making up numbers to play with them, but it's easy to load up data from external files. The most common data interchange format is CSV (comma-seperated values), a plain text format. Think of CSV as a plain text table. Each element in each row is separated by a comma (although other symbols, such as white space, semicolons `;`, tabs `\t` or pipe `|` symbols are often found as delimiters). The first few lines might contain some metadata that describes the dataset, and the first line will also contain the names of the headers of the different columns. Lines starting with `#` tend to be ignored. An example file might look like this

```
# Monthly transatlantic airtravel, in thousands of passengers, for 1958-1960.
# There are 4 fields, "Month", "1958", "1959" and "1960" and 12 records, "JAN"␣
↪through "DEC".
# There is also an initial header line.
# And some lines with comments starting with #
# Data obtained from https://people.sc.fsu.edu/~jburkardt/data/csv/csv.html
"Month", "1958", "1959", "1960"
"JAN",  340,  360,  417
"FEB",  318,  342,  391
"MAR",  362,  406,  419
"APR",  348,  396,  461
"MAY",  363,  420,  472
"JUN",  435,  472,  535
"JUL",  491,  548,  622
"AUG",  505,  559,  606
"SEP",  404,  463,  508
"OCT",  359,  407,  461
"NOV",  310,  362,  390
"DEC",  337,  405,  432
```

We can see the first few lines are comments or metadata, the first line without a `#` is the headers, and we note that text is entered between `"`s. In this case, the delimiter is a comma. We can read the data as an array with `` `np.loadtxt `` <https://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html>'__, telling it…

- to ignore lines starting by #

- to ignore the first column as it's text
- to note that the separator is a comma

```
air_travel = np.loadtxt("data/airtravel.csv", comments="#", skiprows=6, \
                        usecols=[1,2,3], delimiter=",")
print(air_travel)
print(air_travel.shape)
```

```
[[340. 360. 417.]
 [318. 342. 391.]
 [362. 406. 419.]
 [348. 396. 461.]
 [363. 420. 472.]
 [435. 472. 535.]
 [491. 548. 622.]
 [505. 559. 606.]
 [404. 463. 508.]
 [359. 407. 461.]
 [310. 362. 390.]
 [337. 405. 432.]]
(12, 3)
```

While `np.loadtxt` is quite flexible for dealing with text files, ``` `pandas ```<https://pandas.pydata.org>'__ absolutely shines at working with tabular data. You can find a pandas quickstart tutorial here if you are curious about it!

Before we go into plotting, we can do some fun calculations (yay!) using our airtravel data

Exercise 2.5.1

- Calculate the total number of passengers per year
- Calculate the average number of passengers per month
- Can you spot any trends in the data?

Hint: Remember the `.sum()`, `.mean()` methods for arrays?

```
# Space for your solution
```

Let's plot our previous air travel dataset… We'll plot it as annual lines, so the x axis will be month number (running from 1 to 12) and the y axis will be 1000s of passengers. Different line colours will be used for every year. We'll also add x and y axes labels, as well as a legend:

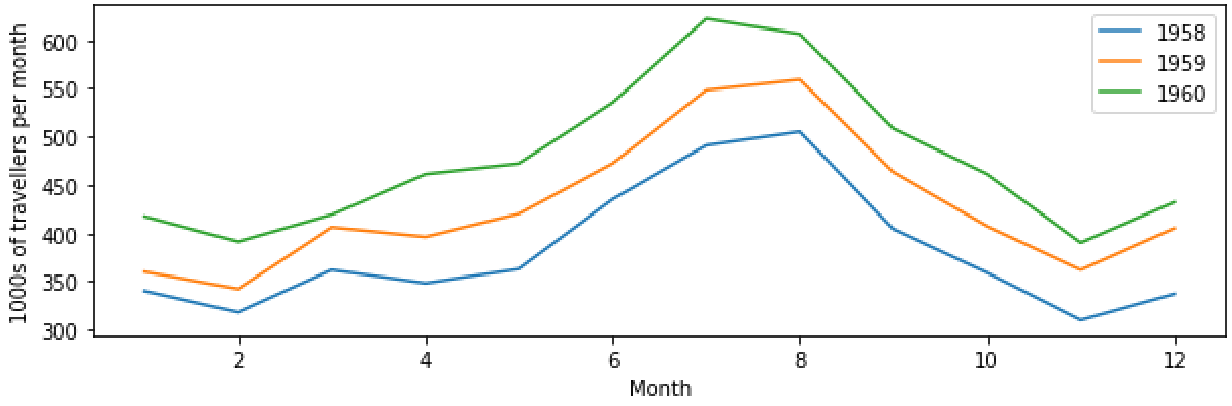
```
# You can probably just put this at the top of every notebook you write
# Adding it here for completeness
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt


# Load airtravel data
air_travel = np.loadtxt("data/airtravel.csv", skiprows=6, \
                        unpack=True, usecols=[1,2,3], delimiter=",")

mths = np.arange(1, 13)
plt.figure(figsize=(10,3))
plt.plot(mths, air_travel[0], '-', label="1958")
plt.plot(mths, air_travel[1], '-', label="1959")
plt.plot(mths, air_travel[2], '-', label="1960")
```

(continues on next page)

(continued from previous page)

```
plt.xlabel("Month")
plt.ylabel("1000s of travellers per month")
plt.legend(loc="best")
```

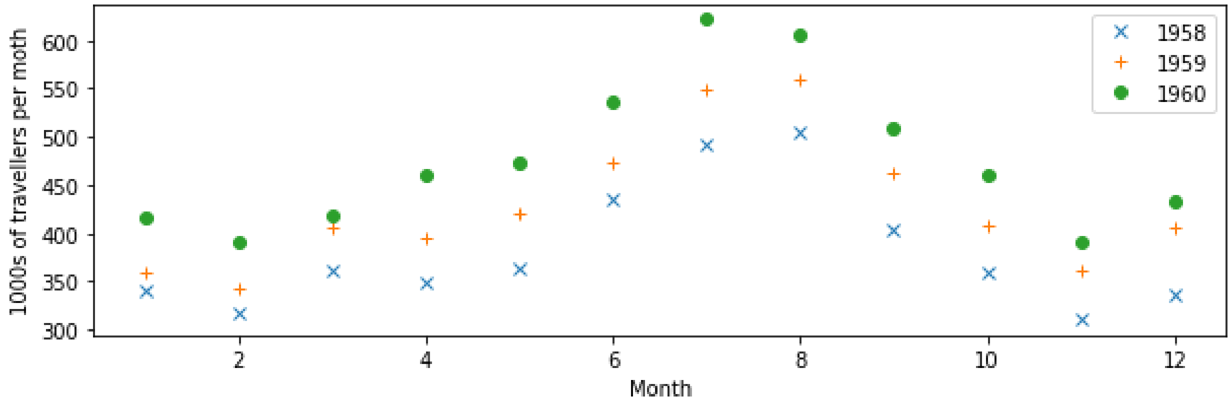
```
<matplotlib.legend.Legend at 0x1169609b0>
```



You may not want to use lines to join the data points, but symbols like dots, crosses, etc.

```
plt.figure(figsize=(10,3))
plt.plot(mths, air_travel[0], 'x', label="1958")
plt.plot(mths, air_travel[1], '+', label="1959")
plt.plot(mths, air_travel[2], 'o', label="1960")
plt.xlabel("Month")
plt.ylabel("1000s of travellers per moth")
plt.legend(loc="best")
```

```
<matplotlib.legend.Legend at 0x116a925f8>
```



We can also use dots **and** lines. Moreover, we can change the type of line: from full lines to dashed to dash-dot. . .

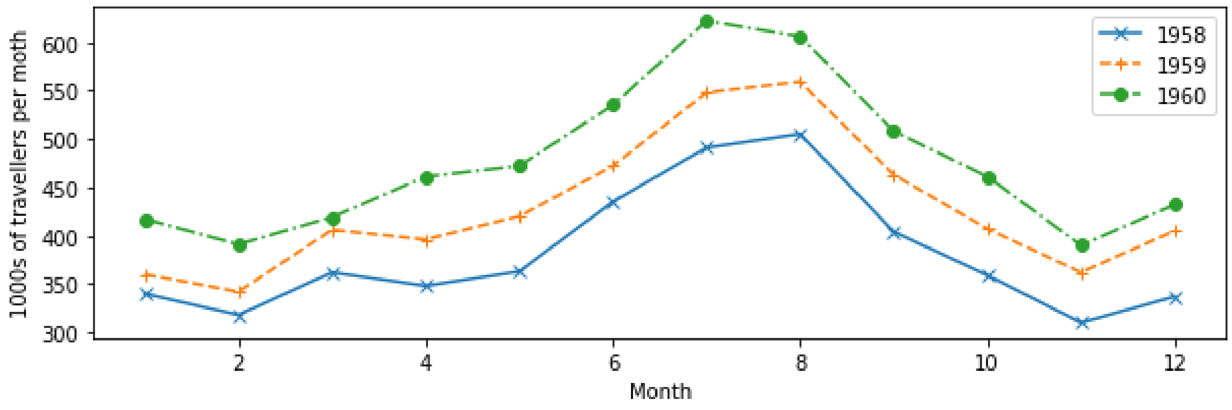
```
plt.figure(figsize=(10,3))
plt.plot(mths, air_travel[0], 'x-', label="1958")
plt.plot(mths, air_travel[1], '+--', label="1959")
plt.plot(mths, air_travel[2], 'o-.', label="1960")
plt.xlabel("Month")
```

(continues on next page)

(continued from previous page)

```
plt.ylabel("1000s of travellers per moth")
plt.legend(loc="best")
```

```
<matplotlib.legend.Legend at 0x11691b7b8>
```



Exercise 2.5.2

The file `` ` ``NOAA.csv <data/NOAA.csv>`__ contains data from NOAA on the number of storms and hurricanes in the Atlantic basin from 1851 to 2015. The data columns are described in the first row of the file. The year is in column 1 and the number of hurricanes in column 3.

For those interested, the data is pulled from the website with getNOAA.py.

- load the year and hurricane data from the file `` ` ``NOAA.csv <data/NOAA.csv>`__ into a numpy array
- produce a plot showing the number of hurricanes as a function of year, with the data plotted in a blue line
- put a dashed red line on the graph showing the mean number of hurricanes
- plot circle symbols for all years where the number of hurricanes is greater than the mean + 1.96 standard deviations.

Hint: the options on `np.loadtxt` you probably want to use are: `skiprows`, `delimiter`, `usecol` and `unpack`. You will need to select the data that meet the required conditions, combining the conditions with `np.logical_and()`.

```
# do exercise here
```

2.5.2 requests

We can use `np.loadtxt` or similar functions to load tabular data that we have stored locally in e.g. csv format.

Sometimes we will need pull a data file from a URL. We have used this idea previously to 'scrape' data from a web page, but often the task is more straightforward, and we effectively need only to 'download' the data in the file.

We will use the `requests` package to do this and pull the data as a string. We then use `StringIO` to allow `np.loadtxt` to think the string comes from a data file.

```
import requests
from io import StringIO

# Define the URL with the parameters of interest
```

(continues on next page)

(continued from previous page)

```
url = "https://daymet.ornl.gov/single-pixel/api/" + \
        "data?lat=45.4&lon=-115.0534&vars=tmax&start=2000-01-01&end=2009-12-31"

data = requests.get(url).text

# You can check the text file to see its contents, but we now
# (i) it's separated by commas
# (ii) the first 8 lines are metadata that we're not interested in.
temperature = np.loadtxt(StringIO(data), skiprows=8, delimiter=",", unpack=True)

# We expect to get 10 years of data here, so 3650 daily records
# the data are given for 365 days per year ...
print(temperature.shape)
```

```
(3, 3650)
```

If we want to store the data file, we can do so by opening a file:

```
# We open the output file, `daymet.csv`
with open("data/daymet_tmax.csv", 'w') as fp:
    # make the HTTPS connection and pull text
    # then write to file
    r = fp.write(data)

# You can check the text file to see its contents, but we now
# (i) it's separated by commas
# (ii) the first 9 lines are metadata that we're not interested in.
temperature = np.loadtxt("data/daymet_tmax.csv", skiprows=8, delimiter=",",␣
↪unpack=True)

# We expect to get ~10 years of data here, so 3650 daily records
print(temperature.shape)
```

```
(3, 3650)
```

The data columns are: `Year`, `day of year` (1 to 365) and `Tmax` (C).

How can we plot such data? the technical issue we face is needing to use the first *two* columns of data (day of year and year) to describe the x-axis location.

```
print('Year ',temperature[0])
print('DOY  ',temperature[1])
print('T_max',temperature[2])
```

```
Year  [2000. 2000. 2000. ... 2009. 2009. 2009.]
DOY   [  1.   2.   3. ... 363. 364. 365.]
T_max [-1.5 -2.5 -1.5 ... -1.5 -3.  -2. ]
```

A simple way of doing this, that would suffice here, would be to convert day of year to year fraction, then we could write:

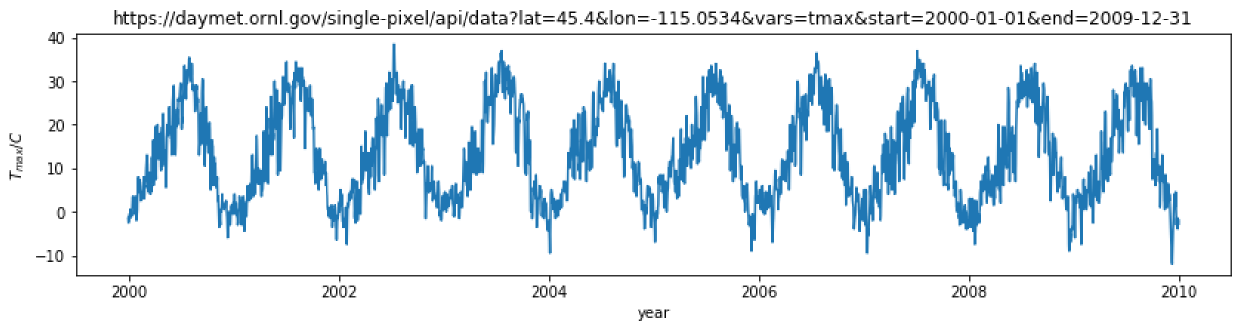
```
year,doy,tmax = temperature
dates = year + (doy-1)/365.
```

A more elegant way might be to use ``datetime` <https://docs.python.org/3/library/datetime.html>`__. This contains a set of methods that allow you to manipulate date formats. `matplotlib` understands the format used, and so it is generally appropriate to use `datetime` for date information when plotting.

```
year,doy,tmax = temperature
dates = [datetime.datetime(int(y), 1, 1) + \
         datetime.timedelta(d - 1) for y,d in zip(year,doy)]
```

```
import datetime

year,doy,tmax = temperature
dates = year + (doy-1)/365.
# using the simple way here

plt.figure(figsize=(14,3))
plt.plot(dates,tmax)
plt.title(url)
plt.ylabel('$T_{max} / C$')
plt.xlabel('year')
```

```
Text(0.5, 0, 'year')
```



Exercise E2.5.3

- use the datetime approach to plot the dataset
- print out the value of `dates` for the first 10 entries to see what the format looks like

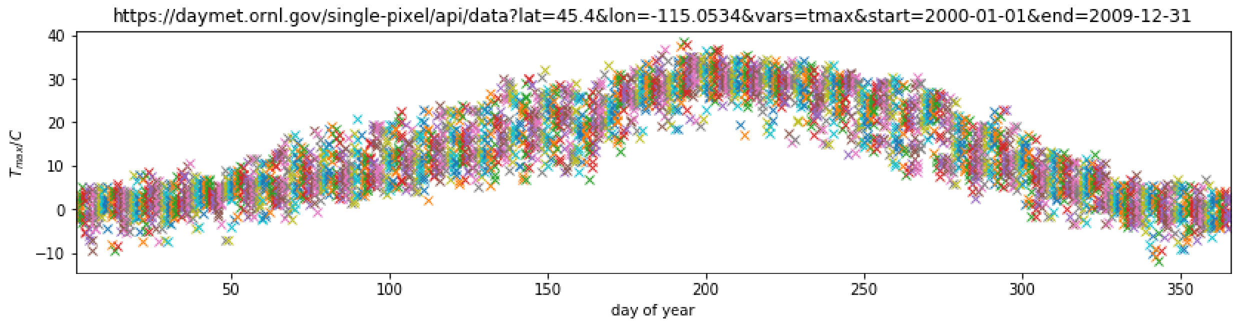
```
# do exercise here
```

Although we have used this as a one-dimensional dataset (temperature as a function of time) we could also think of it as two-dimensional (temperature as a function of (year,doy)). Recall that the `shape` of the `temperature` dataset was `(3,3650)`. We could put the temperature column into a gridded dataset of shape `(10,365)` which would then emphasise the 2-D nature.

We can do this with the `numpy` method `reshape()`.

```
year = temperature[0].reshape(10,365)
doy  = temperature[1].reshape(10,365)
tmax = temperature[2].reshape(10,365)

plt.figure(figsize=(14,3))
plt.plot(doy,tmax,'x')
plt.title(url)
plt.ylabel('$T_{max} / C$')
plt.xlabel('day of year')
plt.xlim([1,366])
```

```
(1, 366)
```



Plotting this, we can visualise the year-on-year variations in temperature for any particular day.

Exercise E2.5.3

- using the reshaped datasets above, calculate and plot the mean value of `tmax` as a function of day of year
- calculate standard deviation of `tmax` as a function of day of year, and plot dashed lines at mean +/- 1.96 standard deviations
- in another plot, show the mean, +/- 1.96 standard deviations of `tmax` as a function of year (i.e. the annual average and standard deviation)

Hint: use `axis=0` when calculating the mean/std over `doy` of `tmax` and `axis=1` for processing over `year`.

2.5.4 Homework

Exercise E2.5.4

Select 4 locations in different regions of North America (e.g. Anchorage, Albuquerque, Seattle, Chicago). Request data on maximum temperature, precipitation and incident solar radiation for the years between 1981 to 2010, and plot in 3 different figures:

- Figure 1: The *mean daily temperature* and the variation (a shaded area around the mean going from mean value **minus** 1.96 times the standard deviation to mean value **plus** 1.96 times the standard deviation). Use a subplot or panel for each site
- Figure 2: The *mean daily precipitation* and the variation (a shaded area around the mean going from mean value **minus** 1.96 times the standard deviation to mean value **plus** 1.96 times the standard deviation). Use a subplot or panel for each site
- Figure 3: The *mean daily incident solar radiation* and the variation (a shaded area around the mean going from mean value **minus** 1.96 times the standard deviation to mean value **plus** 1.96 times the standard deviation). Use a subplot or panel for each site

In each plot, the mean value should be a full line, and the variation should be an envelope, visually similar to the plot shown below (clearly not identical!!!!)

Label each plot with a title, units and so on. Some useful functions to consider

- \``plt.subplots` <https://matplotlib.org/api/_as_gen/matplotlib.pyplot.subplots.html>\`__ Allows you to split a figure into several panels or subplots. In particular, pay attention to the `sharex` and `sharey` options that allow you to have the same scales for all plots so they can be directly compared.

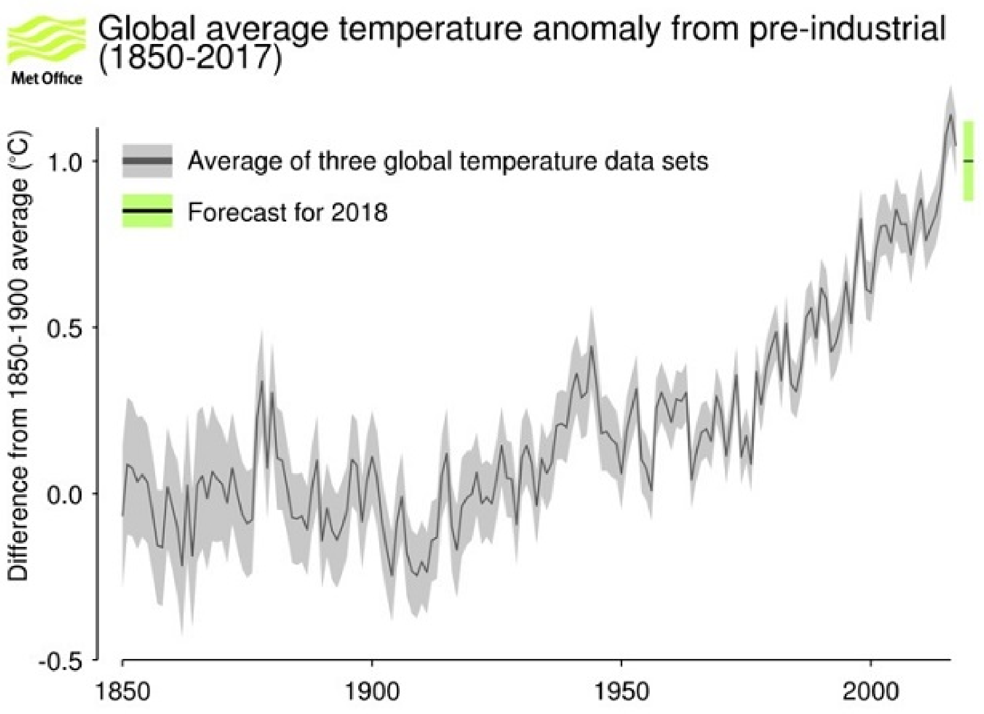


Fig. 6: la niña plot

- `` `plt.fill_between `` <https://matplotlib.org/api/_as_gen/matplotlib.pyplot.fill_between.html>`__ Allows you to fill the space between two curves. You may want to give the option `color=0.8` for a nice grey effect.

```
# do exercise here
```

2.5.5 Summary

In this section, we have learned about reading data from csv files from the local disc or that we have pulled from the web (given a URL). We have gone into a little more detail and sophistication on plotting graphs, and you now should be able to produce sensible plots of datasets or summaries of datasets (e.g. mean standard deviation).

2.8.7 3 Geospatial processing with `gdal`

Table of Contents

GDAL is the workhorse of geospatial processing. Basically, GDAL offers a common library to access a vast number of formats (if you want to see how vast, check this). In addition to letting you open and convert obscure formats to something more useful, a lot of functionality in terms of processing raster data is available (for example, working with projections, combining datasets, accessing remote datasets, etc).

For vector data, the counterpart to GDAL is OGR (which is now a part of the GDAL library anyway), which also supports many vector formats. The combination of both libraries is a very powerful tool to work with geospatial data, not only from Python, but from many other popular computer languages.

In this session, we will introduce the `gdal` geospatial module which can read a wide range of raster scientific data formats. We will also introduce the related `ogr` vector package.

In pacticular, we will learn how to:

- access and download NASA geophysical datasets (specifically, the MODIS LAI/FPAR product)
- apply a vector mask to the dataset
- apply quality control flags to the data
- stack datasets into a 3D numpy dataset for further analysis, including interpolation of missing values
- visualise the data
- store the stacked dataset

These are all tasks that you will be required to do for thepart 1 formal assessment**of this course. You will however be using a different NASA dataset.**

3.1 MODIS LAI product

To introduce geospatial processing, we will use a dataset from the MODIS LAI product over the UK.

You should note that the dataset you need to use for your assessed practical is a MODIS dataset with similar characteristics to the one in this example.

The data product MOD15 LAI/FPAR has been generated from NASA MODIS sensors Terra and Aqua data since 2002. We are now in dataset collection 6 (the data version to use).

```
LAI is defined as the one-sided green leaf area per unit ground area in broadleaf␣
↪canopies and as half the total needle surface area per unit ground area in␣
↪coniferous canopies. FPAR is the fraction of photosynthetically active radiation␣
↪(400-700 nm) absorbed by green vegetation. Both variables are used for calculating␣
↪surface photosynthesis, evapotranspiration, and net primary production, which in␣
↪turn are used to calculate terrestrial energy, carbon, water cycle processes, and␣
↪biogeochemistry of vegetation. Algorithm refinements have improved quality of␣
↪retrievals and consistency with field measurements over all biomes, with a focus on␣
↪woody vegetation.
```

We use such data to map and understand about the dynamics of terrestrial vegetation / carbon, for example, for climate studies.

The raster data are arranged in tiles, indexed by row and column, to cover the globe:

Exercise 3.1.1

The pattern on the tile names is `hXXvYY` where `XX` is the horizontal coordinate and `YY` the vertical.

- use the map above to work out the names of the two tiles that we will need to access data over the UK
- set the variable `tiles` to contain these two names in a list

For example, for the two tiles covering Madegascar, we would set:

```
tiles = ['h22v10','h22v11']
```

```
# do exercise here
```

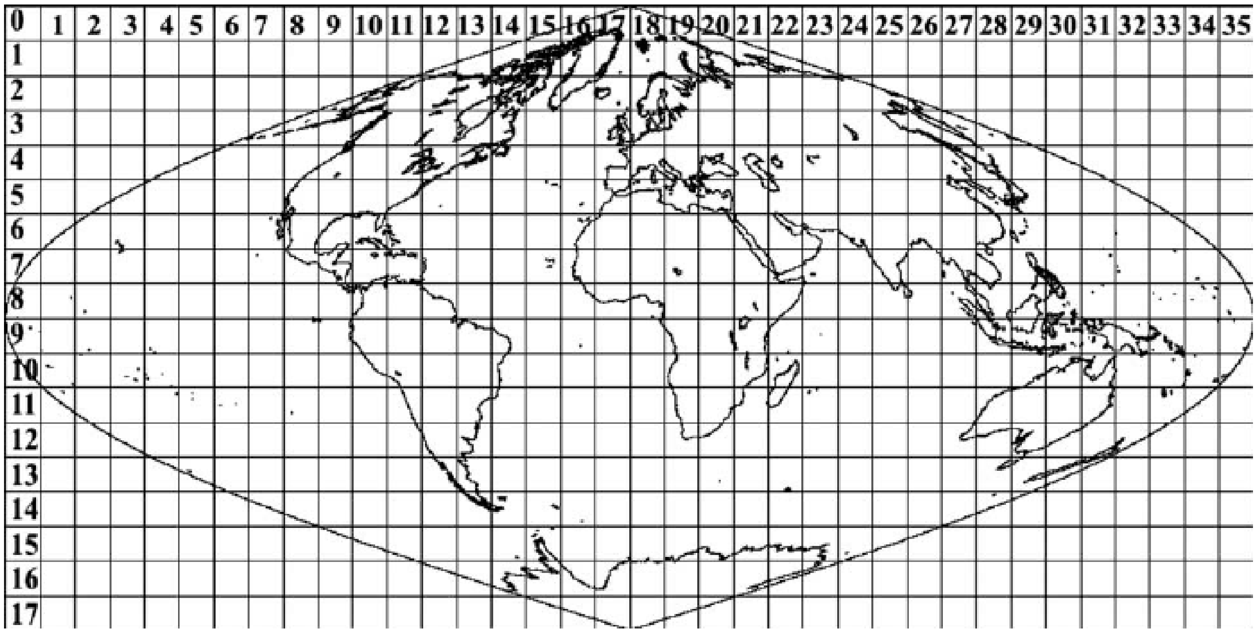


Fig. 7: MODIS tiles

3.1.1 NASA Earthdata access

3.1.1.1 Register at NASA Earthdata

Before you attempt to do this section, you will need to register at NASA Earthdata.

We have set up these notes so that you don't have to put your username and password in plain text. Instead, you need to enter your username and password when prompted by `cylog`. The password is stored in an encrypted file, although it can be accessed as plain text within your Python session.

N.B. using ‘‘cylog().login()‘‘ is only intended to work with access to NASA Earthdata and to prevent you having to expose your username and password in these notes.

`cylog().login()` returns the tuple `(username,password)` in plain text.

```
from geog0111.cylog import cylog
import requests

url = 'https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/2018.09.30/'

# grab the HTML information
html = requests.get(url,auth=cylog(init=False).login()).text

# test a few lines of the html
if html[:20] == '<!DOCTYPE HTML PUBLI':
    print('this seems to be ok ... ')
    print('use cylog().login() anywhere you need to specify the tuple (username,
↪password)')
```

```
this seems to be ok ...
use cylog().login() anywhere you need to specify the tuple (username,password)
```

The NASA servers go down for weekly maintenance, usually on Wednesday afternoon (UK time), so you might not want to attempt this exercise then.

3.1.2 `gdal`

We should now check to see if you have `gdal` properly installed.

```
import gdal
version = gdal.VersionInfo()

if int(version) >= 2020400:
    print('gdal ok',version)
else:
    print('gdal problem',version,'2.2.4+ expected')
```

```
gdal ok 2020400
```

If there is a problem and you are on the geography system, we should be able to fix it for you.

If you are not on the geography system, try running:

```
conda env update -f environment.yml
```

before going any further. If an update occurs, shutdown and restart your notebooks.

3.2 Automatic downloading of NASA MODIS products

In this section, you will learn how to:

- scan the directories (on the Earthdata server) where the MODIS data are stored
- get the dataset filename for a given tile, date and product
- get to URL associated with the dataset
- use the URL to pull the dataset over to store in the local file system

3.3 GDAL masking

In this section you will learn how to:

- load locally stored files into gdal
- select a particular dataset
- form a virtual 'stitched' dataset from multiple files
- apply a mask to the data from a vector boundary
- crop the dataset

3.4 GDAL stacking and interpoilation

In this section you will learn how to:

- access and interpret MODIS LAI QA data
- use `gdal` to create cropped spatial virtual datasets

- use gdal to create a time series virtual dataset
- use convolution in scipy to smooth a temporal dataset, and thereby interpolate
- produce a movie from a time series of imagery, as an animated gif

Additional information on colvolution is given in section 3.4a

2.8.8 3.5 Movies

In this section you see some LAI movies.

3.6 Summary

In this session, we have learned to use some geospatial tools using GDAL in Python. A good set of working notes on how to use GDAL has been developed that you will find useful for further reading, as well as looking at the advanced section.

We have also very briefly introduced dealing with vector datasets in ogr, but this was mainly through the use of a pre-defined function that will take an ESRI shapefile (vector dataset), warp this to the projection of a raster dataset, and produce a mask for a given layer in the vector file.

You should by now know how to:

- access and download NASA (or similar) datasets
- form tiled spatial dataxsets using gdal VRT files
- make a time series of spatial data
- interpret QA information from bit fields in a dataset and use this to create a weighting
- use convolution to smooth and interpolate a dataset
- generate associated plots and animations

2.8.9 3 Geospatial processing with gdal

Table of Contents

GDAL is the workhorse of geospatial processing. Basically, GDAL offers a common library to access a vast number of formats (if you want to see how vast, check this). In addition to letting you open and convert obscure formats to something more useful, a lot of functionality in terms of processing raster data is available (for example, working with projections, combining datasets, accessing remote datasets, etc).

For vector data, the counterpart to GDAL is OGR (which is now a part of the GDAL library anyway), which also supports many vector formats. The combination of both libraries is a very powerful tool to work with geospatial data, not only from Python, but from many other popular computer languages.

In this session, we will introduce the gdal geospatial module which can read a wide range of raster scientific data formats. We will also introduce the related ogr vector package.

In pacticular, we will learn how to:

- access and download NASA geophysical datasets (specifically, the MODIS LAI/FPAR product)
- apply a vector mask to the dataset
- apply quality control flags to the data
- stack datasets into a 3D numpy dataset for further analysis, including interpolation of missing values

- visualise the data
- store the stacked dataset

These are all tasks that you will be required to do for thepart 1 formal assessment**of this course. You will however be using a different NASA dataset.**

3.1 MODIS LAI product

To introduce geospatial processing, we will use a dataset from the MODIS LAI product over the UK.

You should note that the dataset you need to use for your assessed practical is a MODIS dataset with similar characteristics to the one in this example.

The data product MOD15 LAI/FPAR has been generated from NASA MODIS sensors Terra and Aqua data since 2002. We are now in dataset collection 6 (the data version to use).

```
LAI is defined as the one-sided green leaf area per unit ground area in broadleaf␣
↪canopies and as half the total needle surface area per unit ground area in␣
↪coniferous canopies. FPAR is the fraction of photosynthetically active radiation␣
↪(400-700 nm) absorbed by green vegetation. Both variables are used for calculating␣
↪surface photosynthesis, evapotranspiration, and net primary production, which in␣
↪turn are used to calculate terrestrial energy, carbon, water cycle processes, and␣
↪biogeochemistry of vegetation. Algorithm refinements have improved quality of␣
↪retrievals and consistency with field measurements over all biomes, with a focus on␣
↪woody vegetation.
```

We use such data to map and understand about the dynamics of terrestrial vegetation / carbon, for example, for climate studies.

The raster data are arranged in tiles, indexed by row and column, to cover the globe:

Fig. 8: MODIS tiles

Exercise 3.1.1

The pattern on the tile names is `hXXvYY` where `XX` is the horizontal coordinate and `YY` the vertical.

- use the map above to work out the names of the two tiles that we will need to access data over the UK
- set the variable `tiles` to contain these two names in a list

For example, for the two tiles covering Madegascar, we would set:

```
tiles = ['h22v10','h22v11']
```

```
# do exercise here
# ANSWER

tiles = ['h17v03','h17v03']
```

3.1.1 NASA Earthdata access

3.1.1.1 Register at NASA Earthdata

Before you attempt to do this section, you will need to register at NASA Earthdata.

We have set up these notes so that you don't have to put your username and password in plain text. Instead, you need to enter your username and password when prompted by `cylog`. The password is stored in an encrypted file, although it can be accessed as plain text within your Python session.

N.B. using ''cylog().login()'' is only intended to work with access to NASA Earthdata and to prevent you having to expose your username and password in these notes.

`cylog().login()` returns the tuple `(username,password)` in plain text.

```
import geog0111.nasa_requests as nasa_requests
from geog0111.cylog import cylog
%matplotlib inline

url = 'https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/2018.09.30/'

# grab the HTML information
try:
    html = nasa_requests.get(url).text
    # test a few lines of the html
    if html[:20] == '<!DOCTYPE HTML PUBLI':
        print('this seems to be ok ... ')
        print('use cylog().login() anywhere you need to specify the tuple (username,
↪password)')
except:
    print('login error ... try entering your username password again')
    print('then re-run this cell until it works')
    cylog(init=True)
```

The NASA servers go down for weekly maintenance, usually on Wednesday afternoon (UK time), so you might not want to attempt this exercise then.

3.1.2 gdal

We should now check to see if you have `gdal` properly installed.

```
import gdal
version = gdal.VersionInfo()

if int(version) >= 2020400:
    print('gdal ok',version)
else:
    print('gdal problem',version,'2.2.4+ expected')
```

If there is a problem and you are on the geography system, we should be able to fix it for you.

If you are not on the geography system, try running:

```
conda env update -f environment.yml
```

before going any further. If an update occurs, shutdown and restart your notebooks.

3.2 Automatic downloading of NASA MODIS products

In this section, you will learn how to:

- scan the directories (on the Earthdata server) where the MODIS data are stored
- get the dataset filename for a given tile, date and product

- get to URL associated with the dataset
- use the URL to pull the dataset over to store in the local file system

3.3 GDAL masking

In this section you will learn how to:

- load locally stored files into gdal
- select a particular dataset
- form a virtual 'stitched' dataset from multiple files
- apply a mask to the data from a vector boundary
- crop the dataset

3.4 GDAL stacking and interpolating

In this section you will learn how to:

- generate a numpy time series of spatial data
- interpolate/smooth the dataset

3.5 Summary

In this session, we have learned to use some geospatial tools using GDAL in Python. A good set of working notes on how to use GDAL has been developed that you will find useful for further reading, as well as looking at the advanced section.

We have also very briefly introduced dealing with vector datasets in `ogr`, but this was mainly through the use of a pre-defined function that will take an ESRI shapefile (vector dataset), warp this to the projection of a raster dataset, and produce a mask for a given layer in the vector file.

If there is time in the class, we will develop some exercises to examine the datasets we have generated and/or to explore some different datasets or different locations.

2.8.10 3 Geospatial processing with `gdal`

Table of Contents

GDAL is the workhorse of geospatial processing. Basically, GDAL offers a common library to access a vast number of formats (if you want to see how vast, check this). In addition to letting you open and convert obscure formats to something more useful, a lot of functionality in terms of processing raster data is available (for example, working with projections, combining datasets, accessing remote datasets, etc).

For vector data, the counterpart to GDAL is OGR (which is now a part of the GDAL library anyway), which also supports many vector formats. The combination of both libraries is a very powerful tool to work with geospatial data, not only from Python, but from many other popular computer languages.

In this session, we will introduce the `gdal` geospatial module which can read a wide range of raster scientific data formats. We will also introduce the related `ogr` vector package.

In pacticular, we will learn how to:

- access and download NASA geophysical datasets (specifically, the MODIS LAI/FPAR product)

- apply a vector mask to the dataset
- apply quality control flags to the data
- stack datasets into a 3D numpy dataset for further analysis, including interpolation of missing values
- visualise the data
- store the stacked dataset

These are all tasks that you will be required to do for the part 1 formal assessment **of this course. You will however be using a different NASA dataset.**

3.1 MODIS LAI product

To introduce geospatial processing, we will use a dataset from the MODIS LAI product over the UK.

You should note that the dataset you need to use for your assessed practical is a MODIS dataset with similar characteristics to the one in this example.

The data product MOD15 LAI/FPAR has been generated from NASA MODIS sensors Terra and Aqua data since 2002. We are now in dataset collection 6 (the data version to use).

```
LAI is defined as the one-sided green leaf area per unit ground area in broadleaf␣
↪canopies and as half the total needle surface area per unit ground area in␣
↪coniferous canopies. FPAR is the fraction of photosynthetically active radiation␣
↪(400-700 nm) absorbed by green vegetation. Both variables are used for calculating␣
↪surface photosynthesis, evapotranspiration, and net primary production, which in␣
↪turn are used to calculate terrestrial energy, carbon, water cycle processes, and␣
↪biogeochemistry of vegetation. Algorithm refinements have improved quality of␣
↪retrievals and consistency with field measurements over all biomes, with a focus on␣
↪woody vegetation.
```

We use such data to map and understand about the dynamics of terrestrial vegetation / carbon, for example, for climate studies.

The raster data are arranged in tiles, indexed by row and column, to cover the globe:

Fig. 9: MODIS tiles

Exercise 3.1.1

The pattern on the tile names is `hXXvYY` where `XX` is the horizontal coordinate and `YY` the vertical.

- use the map above to work out the names of the two tiles that we will need to access data over the UK
- set the variable `tiles` to contain these two names in a list

For example, for the two tiles covering Madegascar, we would set:

```
tiles = ['h22v10','h22v11']
```

```
# do exercise here
```

3.1.1 NASA Earthdata access

3.1.1.1 Register at NASA Earthdata

Before you attempt to do this section, you will need to register at NASA Earthdata.

We have set up these notes so that you don't have to put your username and password in plain text. Instead, you need to enter your username and password when prompted by `cylog`. The password is stored in an encrypted file, although it can be accessed as plain text within your Python session.

N.B. using ''cylog().login()'' is only intended to work with access to NASA Earthdata and to prevent you having to expose your username and password in these notes.

`cylog().login()` returns the tuple `(username,password)` in plain text.

```
import geog0111.nasa_requests as nasa_requests
from geog0111.cylog import cylog
%matplotlib inline

url = 'https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/2018.09.30/'

# grab the HTML information
try:
    html = nasa_requests.get(url).text
    # test a few lines of the html
    if html[:20] == '<!DOCTYPE HTML PUBLI':
        print('this seems to be ok ... ')
        print('use cylog().login() anywhere you need to specify the tuple (username,
↪password)')
except:
    print('login error ... try entering your username password again')
    print('then re-run this cell until it works')
    cylog(init=True)
```

```
this seems to be ok ...
use cylog().login() anywhere you need to specify the tuple (username,password)
```

The NASA servers go down for weekly maintenance, usually on Wednesday afternoon (UK time), so you might not want to attempt this exercise then.

3.1.2 `gdal`

We should now check to see if you have `gdal` properly installed.

```
import gdal
version = gdal.VersionInfo()

if int(version) >= 2020400:
    print('gdal ok',version)
else:
    print('gdal problem',version,'2.2.4+ expected')
```

```
gdal ok 2020400
```

If there is a problem and you are on the geography system, we should be able to fix it for you.

If you are not on the geography system, try running:

```
conda env update -f environment.yml
```

before going any further. If an update occurs, shutdown and restart your notebooks.

3.2 Automatic downloading of NASA MODIS products

In this section, you will learn how to:

- scan the directories (on the Earthdata server) where the MODIS data are stored
- get the dataset filename for a given tile, date and product
- get to URL associated with the dataset
- use the URL to pull the dataset over to store in the local file system

3.3 GDAL masking

In this section you will learn how to:

- load locally stored files into gdal
- select a particular dataset
- form a virtual 'stitched' dataset from multiple files
- apply a mask to the data from a vector boundary
- crop the dataset

3.4 GDAL stacking and interpolating

In this section you will learn how to:

- generate a numpy time series of spatial data
- interpolate/smooth the dataset

3.X Summary

In this session, we have learned to use some geospatial tools using GDAL in Python. A good set of working notes on how to use GDAL has been developed that you will find useful for further reading, as well as looking at the advanced section.

We have also very briefly introduced dealing with vector datasets in `ogr`, but this was mainly through the use of a pre-defined function that will take an ESRI shapefile (vector dataset), warp this to the projection of a raster dataset, and produce a mask for a given layer in the vector file.

If there is time in the class, we will develop some exercises to examine the datasets we have generated and/or to explore some different datasets or different locations.

2.8.11 3.2 Accessing MODIS Data products

Table of Contents

3.2.1 Introduction

[up to 3.0]

In this section, you will learn how to:

- scan the directories (on the Earthdata server) where the MODIS data are stored
- get the dataset filename for a given tile, date and product
- get to URL associated with the dataset
- use the URL to pull the dataset over to store in the local file system

You should already know:

- basic use of Python (sections 1 and 2)
- the MODIS product grid system
- the two tiles needed to cover the UK

```
tiles = ['h17v03', 'h18v03']
```

- what LAI is and the code for the MODIS LAI/FPAR product MOD15
- your username and password for NASA Earthdata, or have previously entered this with `` `cylog `` <geog0111/cylog.py>`__.

Let's first just test your NASA login:

```
import geog0111.nasa_requests as nasa_requests
from geog0111.cylog import cylog

url = 'https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/2018.09.30/'

# grab the HTML information
try:
    html = nasa_requests.get(url).text
    # test a few lines of the html
    if html[:20] == '<!DOCTYPE HTML PUBLI':
        print('this seems to be ok ... ')
        print('use cylog().login() anywhere you need to specify the tuple (username,
↪password)')
except:
    print('login error ... try entering your username password again')
    print('then re-run this cell until it works')
    cylog(init=True)
```

We use the local class `geog0111.nrequests` here, in place of the usual `requests` as this lets the user avoid exposure to some of the tricky bits of getting data from the NASA server.

3.2.2 Accessing NASA MODIS URLs

Although you can access MODIS datasets through the NASA Earthdata interface, there are many occasions that we would want to just automatically pull datasets. This is particularly true when you want a time series of data that might involve many files. For example, for analysing LAI or other variables over space/time) we will want to write code that pulls the time series of data.

This is also something you will need to do the your assessed practical.

If the data we want to use are accessible to us as a URL, we can simply use `requests` as in previous exercises.

Sometimes, we will be able to specify the parameters of the dataset we want, e.g. using JSON. At othertimes (as in the case here) we might need to do a little work ourselves to construct the particular URL we want.

If you visit the site https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006, you will see 'date' style links (e.g. `2018.09.30`) through to sub-directories.

In these, e.g. https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/2018.09.30/ you will find URLs of a set of files.

The files pointed to by the URLs are the MODIS MOD15 4-day composite 500 m LAI/FPAR product MCD15A3H.

There are links to several datasets on the page, including 'quicklook files' that are jpeg format images of the datasets, e.g.:

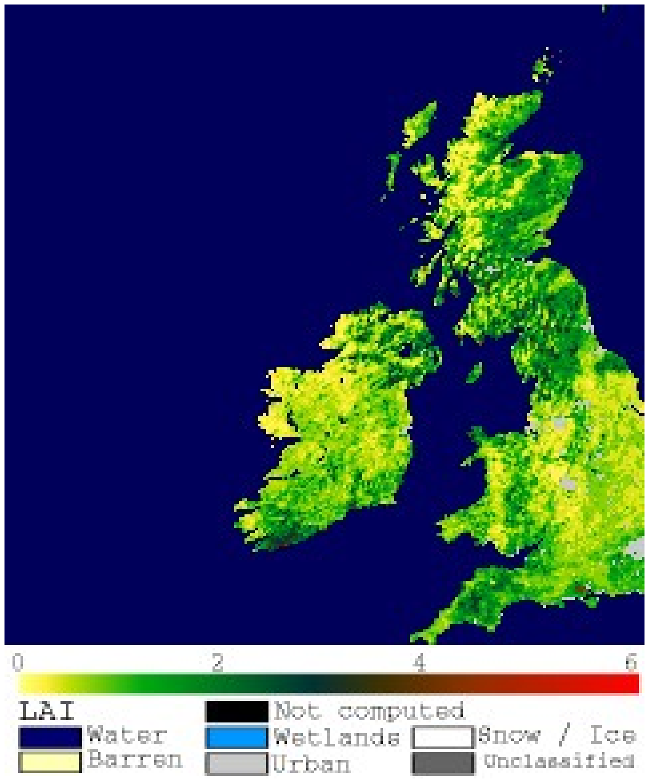


Fig. 10: MCD15A3H.A2018273.h17v03

as well as `xml` files and `hdf` datasets.

3.2.2.1 `datetime`

The URL we have used above, https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/2018.09.30/ starts with a call to the server directory `MOTA`, so we can think of `https://e4ftl01.cr.usgs.gov/MOTA` asd the base level URL.

The rest of the directoy information `MCD15A3H.006/2018.09.30` tells us:

- the product name `MCD15A3H`
- the product version `006`
- the date of the dataset `2018.09.30`

There are several ways we could specify the date information. The most 'human readable' is probably `YYYY.MM.DD` as given here.

Sometimes we will want to refer to it by 'day of year' (`doy`) (sometimes mistakenly referred to as Julian day) for a particular year. Day of year will be an integer that goes from 1 to 365 or 366 (inclusive).

We can use the Python `datetime` to do this:

import datetime

year = 2018

for doy in [1,60,365,366]: # set it up as Jan 1st, plus doy - 1 d = datetime.datetime(year,1,1) + datetime.timedelta(doy-1)

```
# note the careful formatting to include zeros in datestr
datestr = f'{d.year:4d}.{d.month:02d}.{d.day:02d}'

print(f'doy {doy:3d} in {year} is {datestr}')
```

Exercise 3.2.1

- copy the above code, and change the year to a leap year to see if it works as expected
- write some code that loops over each day in the year and converts from `doy` to the format of `datestr` above.
- modify the code so that it forms the full directory URL for the MODIS dataset, e.g. `https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/2018.09.30/` for each `doy`
- use what you have learned to write a function called `get_url()`, which you give the year and day of year and which returns the full URL. It should use keywords to define `product`, `version` and `base_url`.
- For homework, tidy up your function, making sure you document it properly. Think aboiut what might happen if you enter incorrect information.

Hint:

1. number of days in year

   ndays_in_year = (datetime.datetime(year,12,31) - datetime.datetime(year,1,1)).days + 1

Remember that `doy` goes from 1 to 365 or 366 (inclusive).

2. `datestr` format

We use `datestr = f'{d.year:4d}.{d.month:02d}.{d.day:02d}'` as the date string format. The elements such as `{d.year:4d}` mean that `d.year` is interpreted as an integer (`d`) of length `4`. When we put a `0` in front, such as in `02d` the resultant string is 'padded' with `0`. Try something like:

```
value = 10
print(f'{value:X10f}')
```

3. some bigger hints ...

To get the full URL, you will probably want to define something along the lines of:

```
url = f'{base_url}/{product}.{version:03d}/{datestr}'
```

assuming version is an integer.

```
# do exercise here
# ANSWERS

import datetime
# leap year
# so we expect Dec 31st is doy 366
```

(continues on next page)

(continued from previous page)

```
year = 2004
for doy in [1,60,365,366]:
# set it up as Jan 1st, plus doy - 1
    d = datetime.datetime(year,1,1) + datetime.timedelta(doy-1)

    # note the careful formatting to include zeros in datestr
    datestr = f'{d.year:4d}.{d.month:02d}.{d.day:02d}'
    filestr = 'https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/' + datestr
    print(f'doy {doy:3d} in {year} is {filestr}')


def get_url(year,doy,basestr='https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/'):
    '''
    which you give the year and day of year and which returns the full URL.
    It should use keywords to define product, version and base_url
    '''
    d = datetime.datetime(year,1,1) + datetime.timedelta(doy-1)
    datestr = f'{d.year:4d}.{d.month:02d}.{d.day:02d}'
    filestr = basestr + datestr
    return(filestr)

print(get_url(2000,1))
```

```
doy   1 in 2004 is https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/2004.01.01
doy  60 in 2004 is https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/2004.02.29
doy 365 in 2004 is https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/2004.12.30
doy 366 in 2004 is https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/2004.12.31
https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/2000.01.01
```

3.2.2.2 html

When we access this ‘listing’ (directory links such as https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/2018.09.30/) from Python, we will obtain the information in HTML. We don’t expect you to know this language in any great depth, but knowing some of the basics is oftem useful.

```
import geog0111.nasa_requests as nasa_requests
from geog0111.get_url import get_url
import datetime

doy,year = 273,2018
# use your get_url function
# or the one supplied in geog0111
url = get_url(doy,year).url
print(url)

# pull the html
html = nasa_requests.get(url).text

# print a few lines of the html
print(html[:951])
# etc
print('\n','-'*30,'etc','-'*30)
# at the end
print(html[-964:])
```

```
https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/2018.09.30
```

In HTML the code text such as:

```
<a href="MCD15A3H.A2018273.h35v10.006.2018278143650.hdf">MCD15A3H.A2018273.h35v10.006.
↪2018278143650.hdf</a>
```

specifies an HTML link, that will appear as

```
MCD15A3H.A2018273.h35v10.006.2018278143650.hdf 2018-10-05 09:42  7.6K
```

and link to the URL specified in the `href` field: `MCD15A3H.A2018273.h35v10.006.2018278143650.hdf`.

We could interpret this information by searching for strings etc., but the package `BeautifulSoup` can help us a lot in this.

```
import geog0111.nasa_requests as nasa_requests
from geog0111.get_url import get_url
from bs4 import BeautifulSoup

doy,year = 273,2018
url = get_url(doy,year).url
html = nasa_requests.get(url).text

# use BeautifulSoup
# to get all urls referenced with
# html code <a href="some_url">
soup = BeautifulSoup(html,'lxml')
links = [mylink.attrs['href'] for mylink in soup.find_all('a')]
```

Exercise E3.2.2

- copy the code in the block above and print out some of the linformation in the list `links` (e.g. the last 20 entries)
- using an implicit loop, make a list called `hdf_filenames` of only those filenames (links) that have `hdf` as their filename extension.

Hint 1: first you might select an example item from the `links` list:

```
item = links[-1]
print('item is',item)
```

and print:

```
item[-3:]
```

but maybe better (why would this be?) is:

```
item.split('.')[-1]
```

Hint 2: An implicit loop is a construct of the form:

```
[item for item in links]
```

In an implicit for loop, we can actually add a conditional statement if we like, e.g. try:

```
hdf_filenames = [item for item in links if item[-5] == '4']
```

This will print out item if the condition `item[-5] == '4'` is met. That's a bit of a pointless test, but illustrates the pattern required. Try this now with the condition you want to use to select hdf files.

```
# do exercise here
# ANSWER

import geog0111.nasa_requests as nasa_requests
from geog0111.get_url import get_url
from bs4 import BeautifulSoup

doy,year = 273,2018
url = get_url(doy,year).url
html = nasa_requests.get(url).text

# use BeautifulSoup
# to get all urls referenced with
# html code <a href="some_url">
soup = BeautifulSoup(html,'lxml')
links = [mylink.attrs['href'] for mylink in soup.find_all('a')]
print(links[-20:])

hdfs = [mylink.attrs['href'] for mylink in soup.find_all('a') if mylink.attrs['href'].
↪split('.')[-1] == 'hdf']
print(hdfs[-20:])
```

3.2.3 MODIS filename format

The hdf filenames are of the form:

```
MCD15A3H.A2018273.h35v10.006.2018278143650.hdf
```

where:

- the first field (MCD15A3H) gives the product code
- the second (A2018273) gives the observation date: day of year 273, 2018 here
- the third (h35v10) gives the 'MODIS tile' code for the data location
- the remaining fields specify the product version number (006) and a code representing the processing date.

If we want a particular dataset, we would assume then that we know the information to construct the first four fields.

We then have the task remaining of finding an address of the pattern:

```
MCD15A3H.A2018273.h17v03.006.*.hdf
```

where * represents a wildcard (unknown element of the URL/filename).

Putting together the code from above to get a list of the hdf files:

```
#from geog0111.nasa_requests import nasa_requests
from bs4 import BeautifulSoup
from geog0111.get_url import get_url
import geog0111.nasa_requests as nasa_requests

doy,year = 273,2018
url = get_url(doy,year).url
html = nasa_requests.get(url).text
```

(continues on next page)

(continued from previous page)

```
soup = BeautifulSoup(html,'lxml')
links = [mylink.attrs['href'] for mylink in soup.find_all('a')]

# get all files that end 'hdf' as in example above
hdf_filenames = [item for item in links if item.split('.')[-1] == 'hdf']
```

We now want to specify a particular tile or tiles to access.

In this case, we want to look at the field `item.split('.')[-4]` and check to see if it is the list `tiles`.

Exercise 3.2.3

- copy the code above and print out the first 10 values in the list `hdf_filenames`. Can you recognise where the tile information is in the string?

Now, let's check what we get when we look at `item.split('.')[-4]`.

- set a variable called `tiles` containing the names of the UK tiles (as in Exercise 3.1.1)
- write a loop `for item in links:` to loop over each item in the list `links`
- inside this loop set the condition `if item.split('.')[-1] == 'hdf':` to select only `hdf` files, as above
- inside this conditional statement, print out `item.split('.')[-4]` to see if it looks like the tile names
- having confirmed that you are getting the right information, add another conditional statement to see if `item.split('.')[-4] in tiles`, and then print only those filenames that pass both of your tests
- see if you can combine the two tests (the two `if` statements) into a single one

Hint 1: if you print all of the tilenames, this will go on for quite some time. Instead it may be better to use `print(item.split('.')[-4],end=' ')`, which will put a space, rather than a newline between each item printed.

Hint 2: recall what the logical statement `(A and B)` gives when thinking about the combined `if` statement

```
# do exercise here
# ANSWER
hdf_filenames = [item for item in links if item.split('.')[-1] == 'hdf']
print hdf_filenames[-20:]
```

You should end up with something like:

```
import geog0111.nasa_requests as nasa_requests
from bs4 import BeautifulSoup
from geog0111.get_url import get_url

doy,year = 273,2018
tiles = ['h17v03', 'h18v03']

url = get_url(doy,year).url
html = nasa_requests.get(url).text
soup = BeautifulSoup(html,'lxml')
links = [mylink.attrs['href'] for mylink in soup.find_all('a')]

tile_filenames = [item for item in links \
                      if (item.split('.')[-1] == 'hdf') and \
                         (item.split('.')[-4] in tiles)]
```

Exercise E3.2.4

- print out the first 10 items in `tile_filenames` and check the result is as you expect.
- write a function called `modis_tiles()` that takes as input `doy`, `year` and `tiles` and returns a list of the modis tile **urls**.

Hint

1. Don't forget to put in a mechanism to allow you to change the default `base_url`, `product` and `version` (as you did for the function `get_url()`)
2. In some circumstances, yopu can get repeats of filenames in the list. One way to get around this is to convert the list to a `numpy` array, and use `` `np.unique() `` <https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.unique.html>`__ to remove duplicates.

   ```
   import numpy as np
   tile_filenames = np.unique(tile_filenames)
   ```

```
# do exercise here
```

You should end up with something like:

```
from geog0111.modis_tiles import modis_tiles

doy,year = 273,2018
tiles = ['h17v03', 'h18v03']

tile_urls = modis_tiles(doy,year,tiles)
```

Exercise E3.2.5

- print out the first 10 items in `tile_urls` and check the result is as you expect.

```
# do exercise here
```

3.2.4 Saving binary data to a file

We suppose that we want to save the dataset to a local file on the system.

To do that, we need to know how to save a binary dataset to a file. To do this well, we should also consider factors such as whether we want to save a file we already have.

Before we go any further we should check:

- that the directory exists (if not, create it)
- that the file doesn't already exist (else, don't bother)

We can conveniently use methods in `` `pathlib.Path `` <https://docs.python.org/3/library/pathlib.html>`__ for this.

So, import `Path`:

```
from pathlib import Path
```

We suppose we might want to put a file (variable `filename`) into the directory `destination_folder`:

To test if a directory exists and create if not:

```
dest_path = Path(destination_folder)
if not dest_path.exists():
    dest_path.mkdir()
```

To make a compound name of `dest_path` and `filename`:

```
output_fname = dest_path.joinpath(filename)
```

To test if a file exists:

```
if not output_fname.exists():
    print(f"{str(output_fname))} doesn't exist yet ..."})
```

Exercise E3.2.6

- set a variable `destination_folder` to `data` and write code to create this folder ('directory') if it doesn't already exist.
- set a variable `filename` to `test.bin` and write code to check to see if this file is in the folder `destination_folder`. If not, print a message to say so.

```
# do exercise here
```

We now try to read the binary file `data/test_image.bin`.

This involves opening a binary file for reading:

```
fp = open(input_fname, 'rb')
```

Then reading the data:

```
data = fp.read()
```

Then close `fp`

```
fp.close()
```

```
input_fname = 'data/test_image.bin'
fp = open(input_fname, 'rb')
data = fp.read()
fp.close()
print(f'data read is {len(data)} bytes')
```

```
data read is 9136806 bytes
```

And now, write the data as `data/test.bin`.

This involves opening a binary file for writing:

```
fp = open(output_fname, 'wb')
```

Then reading the data:

```
d = fp.write(data)
```

and closing as before:

```
fp.close()
```

```
output_fname = 'data/test.bin'
fp = open(output_fname, 'wb')
d = fp.write(data)
print(f'data written is {d} bytes')
```

```
data written is 9136806 bytes
```

We can avoid the need for the `close` by using the construct:

```
with open(output_fname, 'wb') as fp:
    d = fp.write(data)
```

```
d = 0
with open(output_fname, 'wb') as fp:
    d = fp.write(data)
print(f'data written is {d} bytes')
```

```
data written is 9136806 bytes
```

Exercise E3.2.7

With the ideas above, write some code to:

- check to see if the output directory `data` exists
- if not, create it
- check to see if the input file `data/test_image.bin` exists
- if so, read it in to `data`
- check to see if the output file `data/test.bin` exists
- if not (and if you read data), save `data` to this file
- once you are happy with the code operation, write a function: `save_data(data,filename, destination_folder)` that takes the binary dataset `data` and writes it to the file `filename` in directory `destination_folder`. It should return the n umber of bytes written, and should check to see if files / directories exist and act accordingly.
- add a keyword option to `save_data()` that will overwrite the filename, even if it already exists.

```
# do exercise here
```

You should now know how to save a binary data file.

3.2.4 downloading the data file

The following code uses the `nasa_requests` library to pull some binary data from a URL.

The response is tested (`r.ok`), and if it is ok, then we split the url to derive the filename, and print this out.

The binary dataset is available as `r.content`, which we store to the variable `data` here:

```
import geog0111.nasa_requests as nasa_requests
from geog0111.modis_tiles import modis_tiles
from pathlib import Path

doy,year = 273,2018
tiles = ['h17v03', 'h18v03']
destination_folder = 'data'

tile_urls = modis_tiles(doy,year,tiles)

```

(continues on next page)

(continued from previous page)

```
# loop over urls
for url in tile_urls:
    r = nasa_requests.get(url)

    # check response
    if r.ok:
        # get the filename from the url
        filename = url.split('/')[-1]
        # get the binary data
        data = r.content

        print(filename)
    else:
        print (f'response from {url} not good')
```

```
MCD15A3H.A2018273.h17v03.006.2018278143630.hdf
MCD15A3H.A2018273.h18v03.006.2018278143633.hdf
```

Exercise E3.2.8

- use the code above to write a function `get_modis_files()` that takes as input `doy`, `year` and `tiles`, has a default `destination_folder` of `data`, that downloads the appropriate datasets (if they don't already exist). It should have similar defaults to `modis_tiles()`. It should return a list of the output filenames.

```
# do exercise here
```

You should end up with something like:

```
import geog0111.nasa_requests as nasa_requests
from geog0111.save_data import save_data

doy,year = 273,2018
tiles = ['h17v03', 'h18v03']
destination_folder = 'data'

tile_urls = modis_tiles(doy,year,tiles)

# loop over urls
for url in tile_urls:
    r = nasa_requests.get(url)

    # check response
    if r.ok:
        # get the filename from the url
        filename = url.split('/')[-1]
        # get the binary data
        d = save_data(r.content,filename,destination_folder)
        print(filename,d)
    else:
        print (f'response from {url} not good')
```

```
MCD15A3H.A2018273.h17v03.006.2018278143630.hdf 0
MCD15A3H.A2018273.h18v03.006.2018278143633.hdf 0
```

3.2.5 Visualisation

We will learn more fully how to visualise these later, but just to show that the datasets exist.

You might want to look at the FIPS country codes for selecting boundary data.

```
import requests
import shutil
'''
Get the world borders shapefile that we will need
'''
tm_borders_url = "http://thematicmapping.org/downloads/TM_WORLD_BORDERS-0.3.zip"

r = requests.get(tm_borders_url)
with open("data/TM_WORLD_BORDERS-0.3.zip", 'wb') as fp:
    fp.write (r.content)

shutil.unpack_archive("data/TM_WORLD_BORDERS-0.3.zip",
                     extract_dir="data/")
```

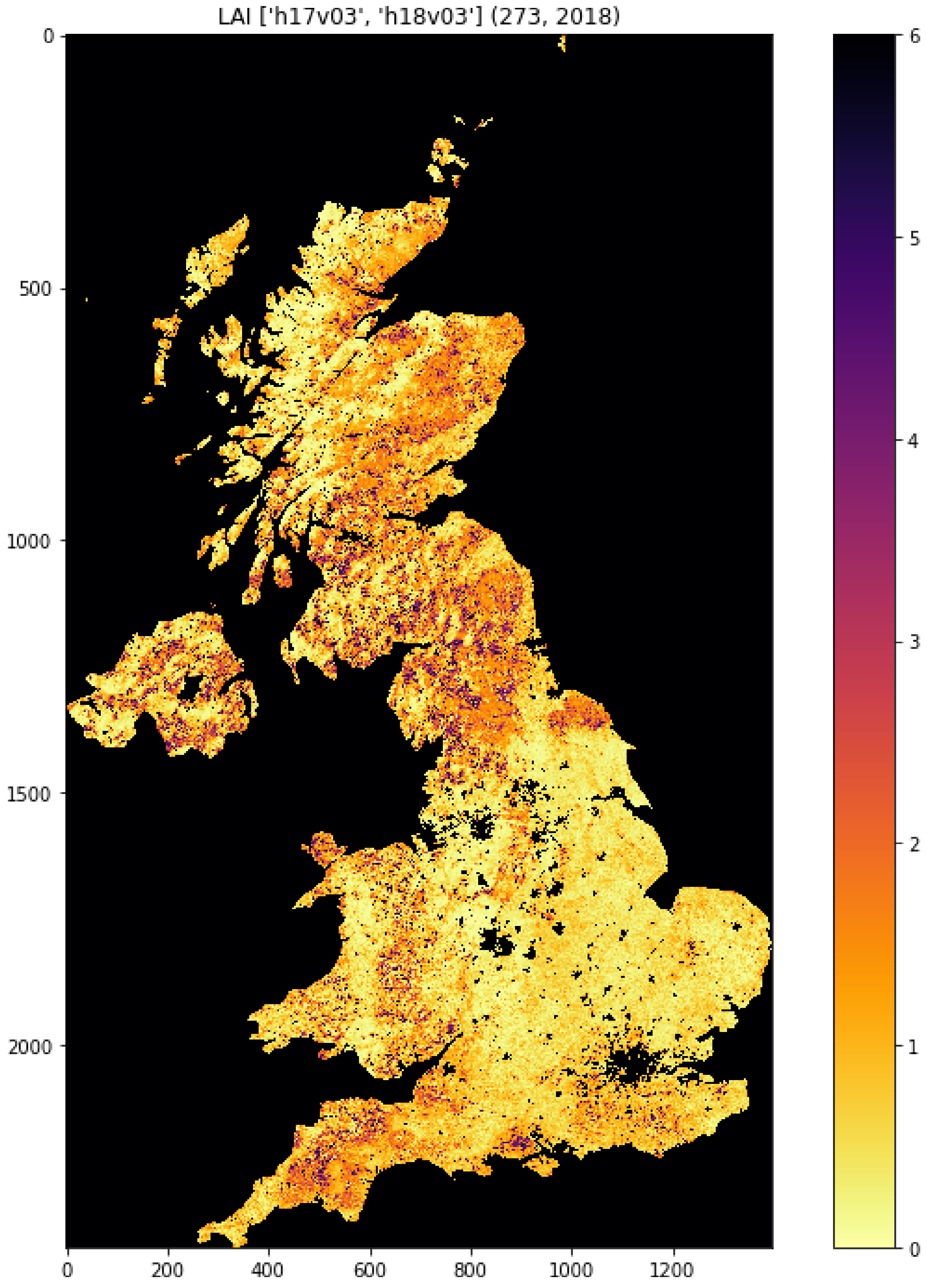
```
from geog0111.get_modis_files import get_modis_files
import gdal
import matplotlib.pylab as plt
import numpy as np

def mosaic_and_mask_data(gdal_fnames, vector_file, vector_where):
    stitch_vrt = gdal.BuildVRT("", gdal_fnames)
    g = gdal.Warp("", stitch_vrt,
                 format = 'MEM', dstNodata=200,
                  cutlineDSName = vector_file,
                  cutlineWhere = vector_where)
    return g

doy,year = 273,2018
tiles = ['h17v03', 'h18v03']
destination_folder = 'data'

filenames = get_modis_files(doy,year,tiles,base_url='https://e4ftl01.cr.usgs.gov/MOTA
↪',\
                                           version=6,\
                                           product='MCD15A3H')

# this part is to access a particular dataset in the file
gdal_fnames = [f'HDF4_EOS:EOS_GRID:"{file_name:s}":MOD_Grid_MCD15A3H:Lai_500m'
               for file_name in filenames]


g = mosaic_and_mask_data(gdal_fnames, "data/TM_WORLD_BORDERS-0.3.shp",
                         "FIPS='UK'")

lai = np.array(g.ReadAsArray()).astype(float) * 0.1 # for LAI scaling
# valid data mask
mask = np.nonzero(lai < 20)
min_y = mask[0].min()
max_y = mask[0].max() + 1

min_x = mask[1].min()
max_x = mask[1].max() + 1
```

(continues on next page)

(continued from previous page)

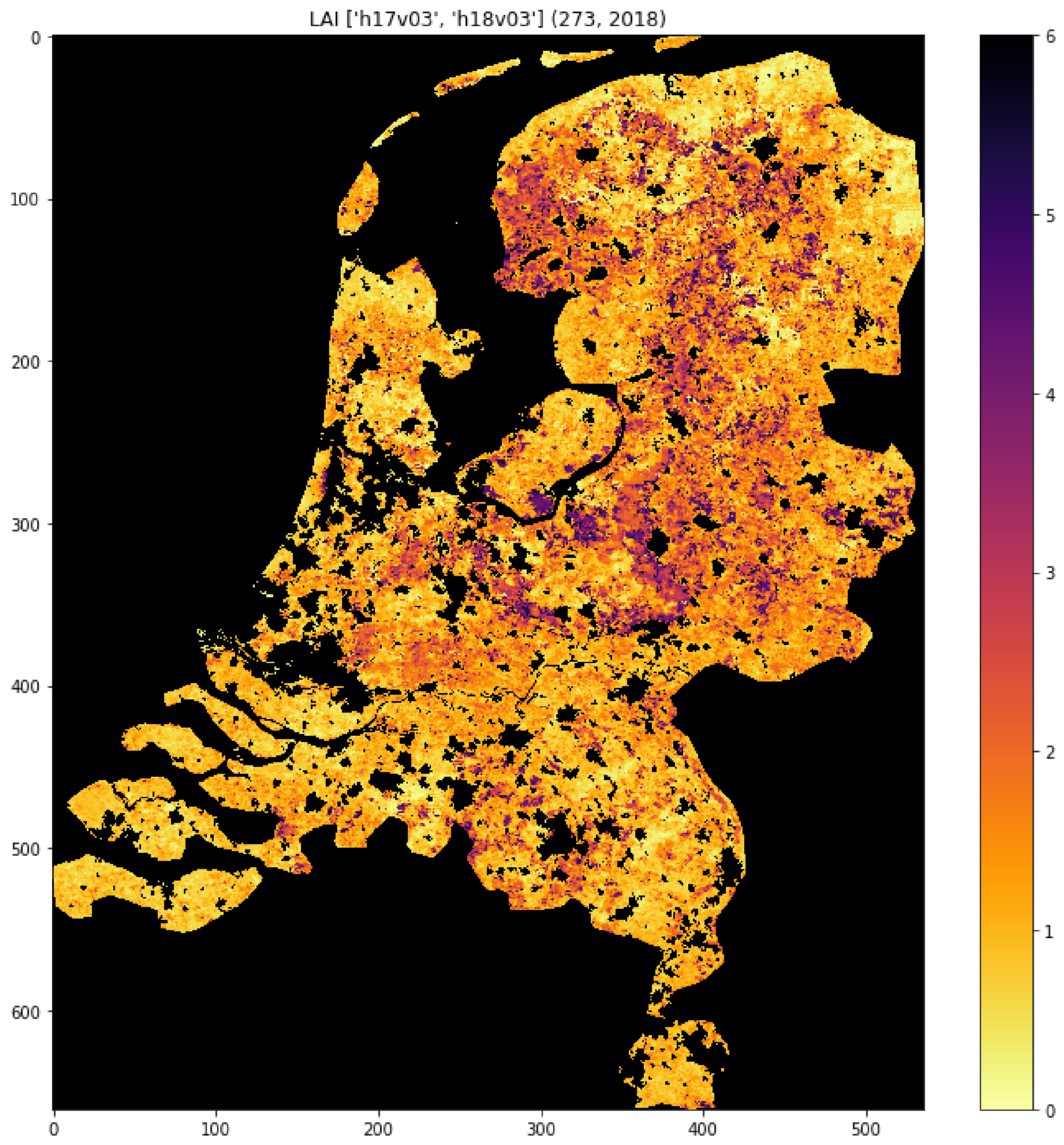
```
lai = lai[min_y:max_y,
               min_x:max_x]

fig = plt.figure(figsize=(12,12))
im = plt.imshow(lai, interpolation="nearest", vmin=0, vmax=6,
             cmap=plt.cm.inferno_r)
plt.title('LAI'+' '+str(tiles)+' '+str((doy,year)))
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x1196614a8>
```


LAI ['h17v03', 'h18v03'] (273, 2018)
0
500
1000
1500
2000
0
200
400
600
800
1000
1200
6
5
4
3
2
1
0

```
from geog0111.get_modis_files import get_modis_files
import gdal
import matplotlib.pylab as plt
import numpy as np

def mosaic_and_mask_data(gdal_fnames, vector_file, vector_where):
    stitch_vrt = gdal.BuildVRT("", gdal_fnames)
    g = gdal.Warp("", stitch_vrt,
                 format = 'MEM', dstNodata=200,
                  cutlineDSName = vector_file,
                  cutlineWhere = vector_where)
    return g

doy,year = 273,2018
tiles = ['h17v03', 'h18v03']
destination_folder = 'data'

filenames = get_modis_files(doy,year,tiles,base_url='https://e4ftl01.cr.usgs.gov/MOTA
↪',\
                                           version=6,\
                                           product='MCD15A3H')

# this part is to access a particular dataset in the file
gdal_fnames = [f'HDF4_EOS:EOS_GRID:"{file_name:s}":MOD_Grid_MCD15A3H:Lai_500m'
               for file_name in filenames]

g = mosaic_and_mask_data(gdal_fnames, "data/TM_WORLD_BORDERS-0.3.shp",
                         "FIPS='NL'")

lai = np.array(g.ReadAsArray()).astype(float) * 0.1 # for LAI scaling
# valid data mask
mask = np.nonzero(lai < 20)
min_y = mask[0].min()
max_y = mask[0].max() + 1

min_x = mask[1].min()
max_x = mask[1].max() + 1

lai = lai[min_y:max_y,
               min_x:max_x]

fig = plt.figure(figsize=(12,12))
im = plt.imshow(lai, interpolation="nearest", vmin=0, vmax=6,
             cmap=plt.cm.inferno_r)
plt.title('LAI'+' '+str(tiles)+' '+str((doy,year)))
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x119591940>
```



Exercise 3.2.7 Homework

- Have a look at the information for `MOD10A1 product <http://www.icess.ucsb.edu/modis/SnowUsrGuide/usrguide_1dtil.html>‘__, which is the 500 m MODIS daily snow cover product.
- Use what you have learned here to download the MOD10A product over the UK

Hint: * The data are on a different server `https://n5eil01u.ecs.nsidc.org/MOST` * the template for the snow cover dataxset is `f'HDF4_EOS:EOS_GRID:"{file_name:s}":MOD_Grid_Snow_500m:NDSI_Snow_Cover'` * today-10 may not be the best example doy: choose something in winter * valid snow cover values are 0 to 100 (use this to set `vmin=0, vmax=100` when plotting)

N.B. You will be required to download this dataset for your assessed practical, so it is a good idea to sort code

for this now

```
# do exercise here
```

3.2.6 Summary

In this session, we have learned how to download MODIS datasets from NASA Earthdata.

We have developed and tested functions that group together the commands we want, ultimately arriving at the function `get_modis_files(doy,year,tiles,**kwargs)`.

We have seen ((if you've done the homework) that such code is re-useable and can directly be used for your assessed practical.

2.8.12 3.2 Accessing MODIS Data products

Table of Contents

3.2.1 Introduction

[up to 3.0]

In this section, you will learn how to:

- scan the directories (on the Earthdata server) where the MODIS data are stored
- get the dataset filename for a given tile, date and product
- get to URL associated with the dataset
- use the URL to pull the dataset over to store in the local file system

You should already know:

- basic use of Python (sections 1 and 2)
- the MODIS product grid system
- the two tiles needed to cover the UK

```
tiles = ['h17v03', 'h18v03']
```

- what LAI is and the code for the MODIS LAI/FPAR product MOD15
- your username and password for NASA Earthdata, or have previously entered this with `cylog <geog0111/cylog.py>`__.

Let's first just test your NASA login:

```
import geog0111.nasa_requests as nasa_requests
from geog0111.cylog import cylog

url = 'https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/2018.09.30/'

# grab the HTML information
try:
    html = nasa_requests.get(url).text
    # test a few lines of the html
    if html[:20] == '<!DOCTYPE HTML PUBLI':
        print('this seems to be ok ... ')
        print('use cylog().login() anywhere you need to specify the tuple (username,
↪password)')
except:
    print('login error ... try entering your username password again')
    print('then re-run this cell until it works')
    cylog(init=True)
```

```
this seems to be ok ...
use cylog().login() anywhere you need to specify the tuple (username,password)
```

We use the local class `geog0111.nrequests` here, in place of the usual `requests` as this lets the user avoid exposure to some of the tricky bits of getting data from the NASA server.

3.2.2 Accessing NASA MODIS URLs

Although you can access MODIS datasets through the NASA Earthdata interface, there are many occasions that we would want to just automatically pull datasets. This is particularly true when you want a time series of data that might involve many files. For example, for analysing LAI or other variables over space/time) we will want to write code that pulls the time series of data.

This is also something you will need to do the your assessed practical.

If the data we want to use are accessible to us as a URL, we can simply use `requests` as in previous exercises.

Sometimes, we will be able to specify the parameters of the dataset we want, e.g. using JSON. At othertimes (as in the case here) we might need to do a little work ourselves to construct the particular URL we want.

If you visit the site https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006, you will see 'date' style links (e.g. `2018.09.30`) through to sub-directories.

In these, e.g. https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/2018.09.30/ you will find URLs of a set of files.

The files pointed to by the URLs are the MODIS MOD15 4-day composite 500 m LAI/FPAR product MCD15A3H.

There are links to several datasets on the page, including 'quicklook files' that are jpeg format images of the datasets, e.g.:

as well as `xml` files and `hdf` datasets.

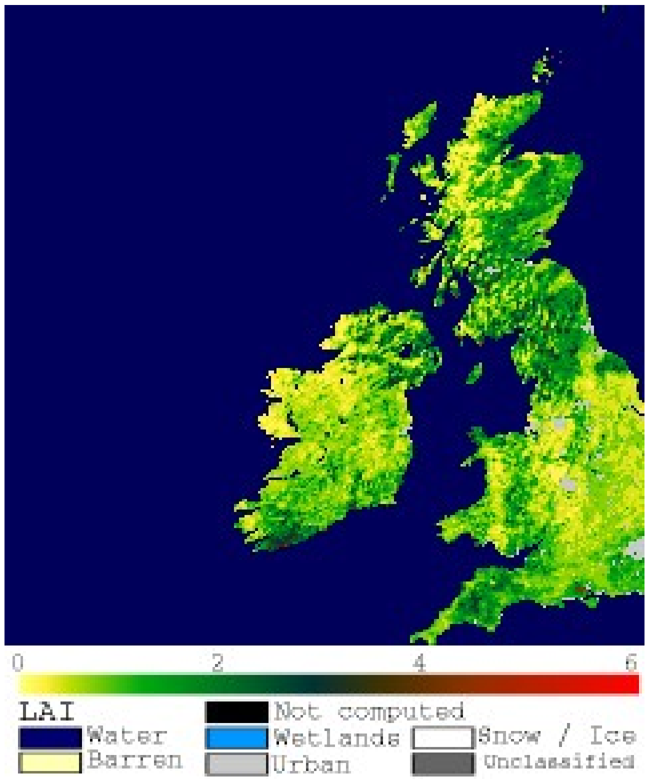


Fig. 11: MCD15A3H.A2018273.h17v03

3.2.2.1 `datetime`

The URL we have used above, https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/2018.09.30/ starts with a call to the server directory `MOTA`, so we can think of `https://e4ftl01.cr.usgs.gov/MOTA` asd the base level URL.

The rest of the directoy information `MCD15A3H.006/2018.09.30` tells us:

- the product name `MCD15A3H`
- the product version `006`
- the date of the dataset `2018.09.30`

There are several ways we could specify the date information. The most 'human readable' is probably `YYYY.MM.DD` as given here.

Sometimes we will want to refer to it by 'day of year' (`doy`) (sometimes mistakenly referred to as Julian day) for a particular year. Day of year will be an integer that goes from 1 to 365 or 366 (inclusive).

We can use the Python `datetime` to do this:

import datetime

year = 2018

for doy in [1,60,365,366]: # set it up as Jan 1st, plus doy - 1 d = datetime.datetime(year,1,1) + datetime.timedelta(doy-1)

```
# note the careful formatting to include zeros in datestr
datestr = f'{d.year:4d}.{d.month:02d}.{d.day:02d}'
```

(continues on next page)

(continued from previous page)

```
print(f'doy {doy:3d} in {year} is {datestr}')
```

Exercise 3.2.1

- copy the above code, and change the year to a leap year to see if it works as expected
- write some code that loops over each day in the year and converts from `doy` to the format of `datestr` above.
- modify the code so that it forms the full directory URL for the MODIS dataset, e.g. `https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/2018.09.30/` for each `doy`
- use what you have learned to write a function called `get_url()`, which you give the year and day of year and which returns the full URL. It should use keywords to define `product`, `version` and `base_url`.
- For homework, tidy up your function, making sure you document it properly. Think aboiut what might happen if you enter incorrect information.

Hint:

1. number of days in year

   ndays_in_year = (datetime.datetime(year,12,31) - datetime.datetime(year,1,1)).days + 1

Remember that `doy` goes from 1 to 365 or 366 (inclusive).

2. `datestr` format

We use `datestr = f'{d.year:4d}.{d.month:02d}.{d.day:02d}'` as the date string format. The elements such as `{d.year:4d}` mean that `d.year` is interpreted as an integer (`d`) of length `4`. When we put a `0` in front, such as in `02d` the resultant string is 'padded' with `0`. Try something like:

```
value = 10
print(f'{value:X10f}')
```

3. some bigger hints …

To get the full URL, you will probably want to define something along the lines of:

```
url = f'{base_url}/{product}.{version:03d}/{datestr}'
```

assuming version is an integer.

```
# do exercise here
```

3.2.2.2 html

When we access this 'listing' (directory links such as https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/2018.09.30/) from Python, we will obtain the information in HTML. We don't expect you to know this language in any great depth, but knowing some of the basics is oftem useful.

```
import geog0111.nasa_requests as nasa_requests
from geog0111.get_url import get_url
import datetime

doy,year = 273,2018
# use your get_url function
# or the one supplied in geog0111
```

(continues on next page)

(continued from previous page)

```
url = get_url(doy,year).url
print(url)

# pull the html
html = nasa_requests.get(url).text

# print a few lines of the html
print(html[:951])
# etc
print('\n','-'*30,'etc','-'*30)
# at the end
print(html[-964:])
```

```
https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/2018.09.30
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<html>
 <head>
  <title>Index of /MOTA/MCD15A3H.006/2018.09.30</title>
 </head>
 <body>
<pre>
****************************************************************************

                         U.S. GOVERNMENT COMPUTER

This US Government computer is for authorized users only.  By accessing this
system you are consenting to complete monitoring with no expectation of␣
↪privacy.
Unauthorized access or use may subject you to disciplinary action and criminal
prosecution.

Attention user: You are downloading data from NASA's Land Processes␣
↪Distributed
Active Archive Center (LP DAAC) located at the USGS Earth Resources␣
↪Observation and
Science (EROS) Center.

Downloading these data requires a NASA Earthdata Login username and password.
To obtain a NASA Earthdata Login account, please visit
<a href="https://urs.earthdata.nasa.gov/users/new">https://urs.earthdata.nasa.
↪gov/users/new/</a>

 ------------------------------ etc ------------------------------

<img src="/icons/unknown.gif" alt="[   ]"> <a href="MCD15A3H.A2018273.h35v08.
↪006.2018278143649.hdf.xml">MCD15A3H.A2018273.h35v08.006.2018278143649.hdf.
↪xml</a>      2018-10-05 09:42  7.6K
<img src="/icons/unknown.gif" alt="[   ]"> <a href="MCD15A3H.A2018273.h35v09.
↪006.2018278143649.hdf">MCD15A3H.A2018273.h35v09.006.2018278143649.hdf</a>  ␣
↪       2018-10-05 09:42  207K
<img src="/icons/unknown.gif" alt="[   ]"> <a href="MCD15A3H.A2018273.h35v09.
↪006.2018278143649.hdf.xml">MCD15A3H.A2018273.h35v09.006.2018278143649.hdf.
↪xml</a>      2018-10-05 09:42  7.6K
<img src="/icons/unknown.gif" alt="[   ]"> <a href="MCD15A3H.A2018273.h35v10.
```

```
↪006.2018278143650.hdf">MCD15A3H.A2018273.h35v10.006.2018278143650.hdf</a>  ␣
↪        2018-10-05 09:42  298K
<img src="/icons/unknown.gif" alt="[   ]"> <a href="MCD15A3H.A2018273.h35v10.
↪006.2018278143650.hdf.xml">MCD15A3H.A2018273.h35v10.006.2018278143650.hdf.
↪xml</a>       2018-10-05 09:42  7.6K
<hr></pre>
</body></html>
```

In HTML the code text such as:

```
<a href="MCD15A3H.A2018273.h35v10.006.2018278143650.hdf">MCD15A3H.A2018273.h35v10.006.
↪2018278143650.hdf</a>
```

specifies an HTML link, that will appear as

```
MCD15A3H.A2018273.h35v10.006.2018278143650.hdf 2018-10-05 09:42  7.6K
```

and link to the URL specified in the `href` field: `MCD15A3H.A2018273.h35v10.006.2018278143650.hdf`.

We could interpret this information by searching for strings etc., but the package `BeautifulSoup` can help us a lot in this.

```
import geog0111.nasa_requests as nasa_requests
from geog0111.get_url import get_url
from bs4 import BeautifulSoup

doy,year = 273,2018
url = get_url(doy,year).url
html = nasa_requests.get(url).text

# use BeautifulSoup
# to get all urls referenced with
# html code <a href="some_url">
soup = BeautifulSoup(html,'lxml')
links = [mylink.attrs['href'] for mylink in soup.find_all('a')]
```

Exercise E3.2.2

- copy the code in the block above and print out some of the linformation in the list `links` (e.g. the last 20 entries)
- using an implicit loop, make a list called `hdf_filenames` of only those filenames (links) that have `hdf` as their filename extension.

Hint 1: first you might select an example item from the `links` list:

```
item = links[-1]
print('item is',item)
```

and print:

```
item[-3:]
```

but maybe better (why would this be?) is:

```
item.split('.')[-1]
```

Hint 2: An implicit loop is a construct of the form:

```
[item for item in links]
```

In an implicit for loop, we can actually add a conditional statement if we like, e.g. try:

```
hdf_filenames = [item for item in links if item[-5] == '4']
```

This will print out `item` if the condition `item[-5] == '4'` is met. That's a bit of a pointless test, but illustrates the pattern required. Try this now with the condition you want to use to select `hdf` files.

```
# do exercise here
```

3.2.3 MODIS filename format

The `hdf` filenames are of the form:

```
MCD15A3H.A2018273.h35v10.006.2018278143650.hdf
```

where:

- the first field (`MCD15A3H`) gives the product code
- the second (`A2018273`) gives the observation date: day of year `273`, `2018` here
- the third (`h35v10`) gives the 'MODIS tile' code for the data location
- the remaining fields specify the product version number (`006`) and a code representing the processing date.

If we want a particular dataset, we would assume then that we know the information to construct the first four fields.

We then have the task remaining of finding an address of the pattern:

```
MCD15A3H.A2018273.h17v03.006.*.hdf
```

where `*` represents a wildcard (unknown element of the URL/filename).

Putting together the code from above to get a list of the `hdf` files:

```
#from geog0111.nasa_requests import nasa_requests
from bs4 import BeautifulSoup
from geog0111.get_url import get_url
import geog0111.nasa_requests as nasa_requests

doy,year = 273,2018
url = get_url(doy,year).url
html = nasa_requests.get(url).text
soup = BeautifulSoup(html,'lxml')
links = [mylink.attrs['href'] for mylink in soup.find_all('a')]

# get all files that end 'hdf' as in example above
hdf_filenames = [item for item in links if item.split('.')[-1] == 'hdf']
```

We now want to specify a particular tile or tiles to access.

In this case, we want to look at the field `item.split('.')[-4]` and check to see if it is the list `tiles`.

Exercise 3.2.3

- copy the code above and print out the first 10 values in the list `hdf_filenames`. Can you recognise where the tile information is in the string?

Now, let's check what we get when we look at `item.split('.')[-4]`.

- set a variable called `tiles` containing the names of the UK tiles (as in Exercise 3.1.1)
- write a loop `for item in links:` to loop over each item in the list `links`
- inside this loop set the condition `if item.split('.')[-1] == 'hdf':` to select only `hdf` files, as above
- inside this conditional statement, print out `item.split('.')[-4]` to see if it looks like the tile names
- having confirmed that you are getting the right information, add another conditional statement to see if `item.split('.')[-4] in tiles`, and then print only those filenames that pass both of your tests
- see if you can combine the two tests (the two `if` statements) into a single one

Hint 1: if you print all of the tilenames, this will go on for quite some time. Instead it may be better to use `print(item.split('.')[-4],end=' ')`, which will put a space, rather than a newline between each item printed.

Hint 2: recall what the logical statement `(A and B)` gives when thinking about the combined `if` statement

```
# do exercise here
```

You should end up with something like:

```
import geog0111.nasa_requests as nasa_requests
from bs4 import BeautifulSoup
from geog0111.get_url import get_url

doy,year = 273,2018
tiles = ['h17v03', 'h18v03']

url = get_url(doy,year).url
html = nasa_requests.get(url).text
soup = BeautifulSoup(html,'lxml')
links = [mylink.attrs['href'] for mylink in soup.find_all('a')]

tile_filenames = [item for item in links \
                      if (item.split('.')[-1] == 'hdf') and \
                         (item.split('.')[-4] in tiles)]
```

Exercise E3.2.4

- print out the first 10 items in `tile_filenames` and check the result is as you expect.
- write a function called `modis_tiles()` that takes as input `doy`, `year` and `tiles` and returns a list of the modis tile **urls**.

Hint

1. Don't forget to put in a mechanism to allow you to change the default `base_url`, `product` and `version` (as you did for the function `get_url()`)
2. In some circumstances, yopu can get repeats of filenames in the list. One way to get around this is to convert the list to a `numpy` array, and use `` `np.unique() `` <https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.unique.html>'__ to remove duplicates.

   ```
   import numpy as np
   tile_filenames = np.unique(tile_filenames)
   ```

```
# do exercise here
```

You should end up with something like:

```
from geog0111.modis_tiles import modis_tiles

doy,year = 273,2018
tiles = ['h17v03', 'h18v03']

tile_urls = modis_tiles(doy,year,tiles)
```

Exercise E3.2.5

- print out the first 10 items in `tile_urls` and check the result is as you expect.

```
# do exercise here
```

3.2.4 Saving binary data to a file

We suppose that we want to save the dataset to a local file on the system.

To do that, we need to know how to save a binary dataset to a file. To do this well, we should also consider factors such as whether we want to save a file we already have.

Before we go any further we should check:

- that the directory exists (if not, create it)
- that the file doesn't already exist (else, don't bother)

We can conveniently use methods in `` ` ``pathlib.Path <https://docs.python.org/3/library/pathlib.html>`__ for this.

So, import `Path`:

```
from pathlib import Path
```

We suppose we might want to put a file (variable `filename`) into the directory `destination_folder`:

To test if a directory exists and create if not:

```
dest_path = Path(destination_folder)
if not dest_path.exists():
    dest_path.mkdir()
```

To make a compound name of `dest_path` and `filename`:

```
output_fname = dest_path.joinpath(filename)
```

To test if a file exists:

```
if not output_fname.exists():
    print(f"{str(output_fname))} doesn't exist yet ..."})
```

Exercise E3.2.6

- set a variable `destination_folder` to `data` and write code to create this folder ('directory') if it doesn't already exist.
- set a variable `filename` to `test.bin` and write code to check to see if this file is in the folder `destination_folder`. If not, print a message to say so.

```
# do exercise here
```

We now try to read the binary file `data/test_image.bin`.

This involves opening a binary file for reading:

```
fp = open(input_fname, 'rb')
```

Then reading the data:

```
data = fp.read()
```

Then close `fp`

```
fp.close()
```

```
input_fname = 'data/test_image.bin'
fp = open(input_fname, 'rb')
data = fp.read()
fp.close()
print(f'data read is {len(data)} bytes')
```

```
data read is 9136806 bytes
```

And now, write the data as `data/test.bin`.

This involves opening a binary file for writing:

```
fp = open(output_fname, 'wb')
```

Then reading the data:

```
d = fp.write(data)
```

and closing as before:

```
fp.close()
```

```
output_fname = 'data/test.bin'
fp = open(output_fname, 'wb')
d = fp.write(data)
print(f'data written is {d} bytes')
```

```
data written is 9136806 bytes
```

We can avoid the need for the `close` by using the construct:

```
with open(output_fname, 'wb') as fp:
    d = fp.write(data)
```

```
d = 0
with open(output_fname, 'wb') as fp:
    d = fp.write(data)
print(f'data written is {d} bytes')
```

```
data written is 9136806 bytes
```

Exercise E3.2.7

With the ideas above, write some code to:

- check to see if the output directory `data` exists
- if not, create it
- check to see if the input file `data/test_image.bin` exists
- if so, read it in to `data`
- check to see if the output file `data/test.bin` exists
- if not (and if you read data), save `data` to this file
- once you are happy with the code operation, write a function: `save_data(data,filename, destination_folder)` that takes the binary dataset `data` and writes it to the file `filename` in directory `destination_folder`. It should return the n umber of bytes written, and should check to see if files / directories exist and act accordingly.
- add a keyword option to `save_data()` that will overwrite the filename, even if it already exists.

```
# do exercise here
```

You should now know how to save a binary data file.

3.2.4 downloading the data file

The following code uses the `nasa_requests` library to pull some binary data from a URL.

The response is tested (`r.ok`), and if it is ok, then we split the url to derive the filename, and print this out.

The binary dataset is available as `r.content`, which we store to the variable `data` here:

```
import geog0111.nasa_requests as nasa_requests
from geog0111.modis_tiles import modis_tiles
from pathlib import Path

doy,year = 273,2018
tiles = ['h17v03', 'h18v03']
destination_folder = 'data'

tile_urls = modis_tiles(doy,year,tiles)

# loop over urls
for url in tile_urls:
    r = nasa_requests.get(url)

    # check response
    if r.ok:
        # get the filename from the url
        filename = url.split('/')[-1]
        # get the binary data
        data = r.content

        print(filename)
```

(continues on next page)

(continued from previous page)

```
    else:
        print (f'response from {url} not good')
```

```
MCD15A3H.A2018273.h17v03.006.2018278143630.hdf
MCD15A3H.A2018273.h18v03.006.2018278143633.hdf
```

Exercise E3.2.8

- use the code above to write a function `get_modis_files()` that takes as input `doy`, `year` and `tiles`, has a default `destination_folder` of `data`, that downloads the appropriate datasets (if they don't already exist). It should have similar defaults to `modis_tiles()`. It should return a list of the output filenames.

```
# do exercise here
```

You should end up with something like:

```
import geog0111.nasa_requests as nasa_requests
from geog0111.save_data import save_data

doy,year = 273,2018
tiles = ['h17v03', 'h18v03']
destination_folder = 'data'

tile_urls = modis_tiles(doy,year,tiles)

# loop over urls
for url in tile_urls:
    r = nasa_requests.get(url)

    # check response
    if r.ok:
        # get the filename from the url
        filename = url.split('/')[-1]
        # get the binary data
        d = save_data(r.content,filename,destination_folder)
        print(filename,d)
    else:
        print (f'response from {url} not good')
```

```
MCD15A3H.A2018273.h17v03.006.2018278143630.hdf 0
MCD15A3H.A2018273.h18v03.006.2018278143633.hdf 0
```

3.2.5 Visualisation

We will learn more fully how to visualise these later, but just to show that the datasets exist.

You might want to look at the FIPS country codes for selecting boundary data.

```
import requests
import shutil
'''
Get the world borders shapefile that we will need
'''
tm_borders_url = "http://thematicmapping.org/downloads/TM_WORLD_BORDERS-0.3.zip"

```

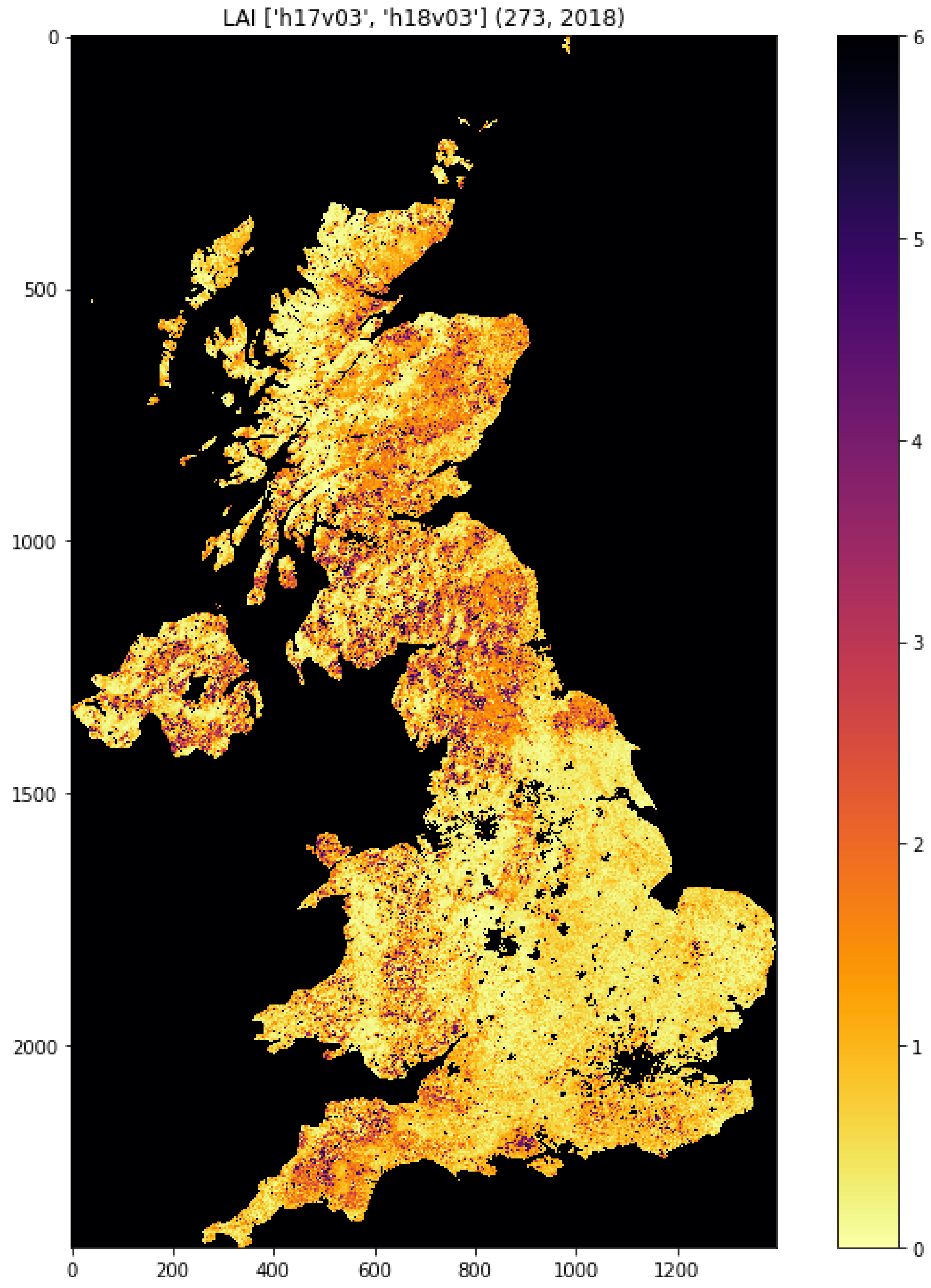
(continues on next page)

(continued from previous page)

```
r = requests.get(tm_borders_url)
with open("data/TM_WORLD_BORDERS-0.3.zip", 'wb') as fp:
    fp.write (r.content)

shutil.unpack_archive("data/TM_WORLD_BORDERS-0.3.zip",
                     extract_dir="data/")
```

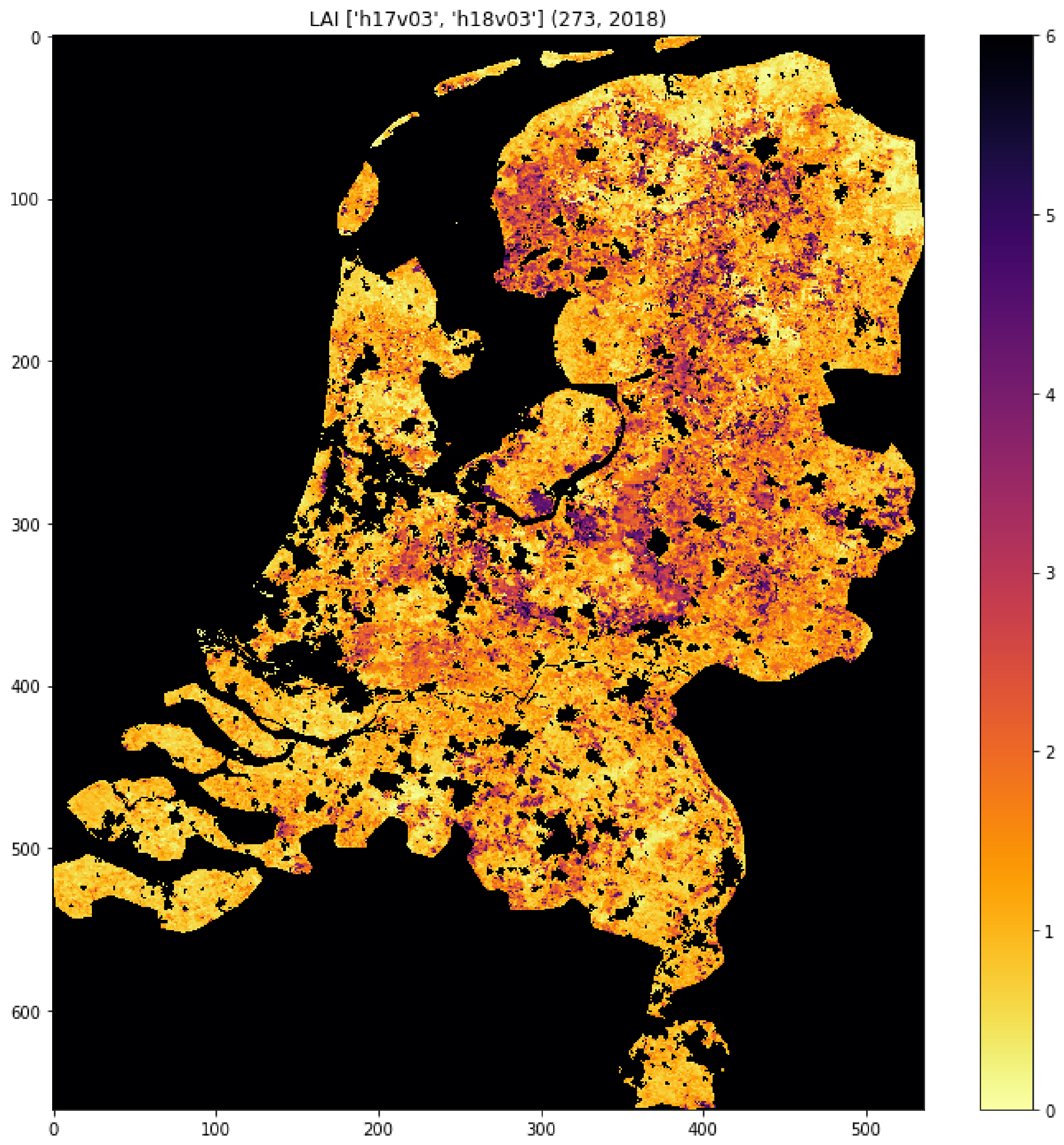
```
from geog0111.get_modis_files import get_modis_files
import gdal
import matplotlib.pylab as plt
import numpy as np

def mosaic_and_mask_data(gdal_fnames, vector_file, vector_where):
    stitch_vrt = gdal.BuildVRT("", gdal_fnames)
    g = gdal.Warp("", stitch_vrt,
                 format = 'MEM', dstNodata=200,
                  cutlineDSName = vector_file,
                  cutlineWhere = vector_where)
    return g

doy,year = 273,2018
tiles = ['h17v03', 'h18v03']
destination_folder = 'data'

filenames = get_modis_files(doy,year,tiles,base_url='https://e4ftl01.cr.usgs.gov/MOTA
↪',\
                                           version=6,\
                                           product='MCD15A3H')

# this part is to access a particular dataset in the file
gdal_fnames = [f'HDF4_EOS:EOS_GRID:"{file_name:s}":MOD_Grid_MCD15A3H:Lai_500m'
               for file_name in filenames]


g = mosaic_and_mask_data(gdal_fnames, "data/TM_WORLD_BORDERS-0.3.shp",
                         "FIPS='UK'")

lai = np.array(g.ReadAsArray()).astype(float) * 0.1 # for LAI scaling
# valid data mask
mask = np.nonzero(lai < 20)
min_y = mask[0].min()
max_y = mask[0].max() + 1

min_x = mask[1].min()
max_x = mask[1].max() + 1

lai = lai[min_y:max_y,
               min_x:max_x]

fig = plt.figure(figsize=(12,12))
im = plt.imshow(lai, interpolation="nearest", vmin=0, vmax=6,
             cmap=plt.cm.inferno_r)
plt.title('LAI'+' '+str(tiles)+' '+str((doy,year)))
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x1196614a8>
```


LAI ['h17v03', 'h18v03'] (273, 2018)
0
500
1000
1500
2000
0
200
400
600
800
1000
1200
6
5
4
3
2
1
0

```
from geog0111.get_modis_files import get_modis_files
import gdal
import matplotlib.pylab as plt
import numpy as np

def mosaic_and_mask_data(gdal_fnames, vector_file, vector_where):
    stitch_vrt = gdal.BuildVRT("", gdal_fnames)
    g = gdal.Warp("", stitch_vrt,
                  format = 'MEM', dstNodata=200,
                   cutlineDSName = vector_file,
                   cutlineWhere = vector_where)
    return g

doy,year = 273,2018
tiles = ['h17v03', 'h18v03']
destination_folder = 'data'

filenames = get_modis_files(doy,year,tiles,base_url='https://e4ftl01.cr.usgs.gov/MOTA
↪',\
                                           version=6,\
                                           product='MCD15A3H')

# this part is to access a particular dataset in the file
gdal_fnames = [f'HDF4_EOS:EOS_GRID:"{file_name:s}":MOD_Grid_MCD15A3H:Lai_500m'
               for file_name in filenames]

g = mosaic_and_mask_data(gdal_fnames, "data/TM_WORLD_BORDERS-0.3.shp",
                         "FIPS='NL'")

lai = np.array(g.ReadAsArray()).astype(float) * 0.1 # for LAI scaling
# valid data mask
mask = np.nonzero(lai < 20)
min_y = mask[0].min()
max_y = mask[0].max() + 1

min_x = mask[1].min()
max_x = mask[1].max() + 1

lai = lai[min_y:max_y,
               min_x:max_x]

fig = plt.figure(figsize=(12,12))
im = plt.imshow(lai, interpolation="nearest", vmin=0, vmax=6,
             cmap=plt.cm.inferno_r)
plt.title('LAI'+' '+str(tiles)+' '+str((doy,year)))
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x119591940>
```



Exercise 3.2.7 Homework

- Have a look at the information for `MOD10A1 product <http://www.icess.ucsb.edu/modis/SnowUsrGuide/usrguide_1dtil.html>`__, which is the 500 m MODIS daily snow cover product.
- Use what you have learned here to download the MOD10A product over the UK

Hint: * The data are on a different server `https://n5eil01u.ecs.nsidc.org/MOST` * the template for the snow cover dataxset is `f'HDF4_EOS:EOS_GRID:"{file_name:s}":MOD_Grid_Snow_500m:NDSI_Snow_Cover'` * today-10 may not be the best example doy: choose something in winter * valid snow cover values are 0 to 100 (use this to set `vmin=0, vmax=100` when plotting)

N.B. You will be required to download this dataset for your assessed practical, so it is a good idea to sort code

for this now

```
# do exercise here
```

3.2.6 Summary

In this session, we have learned how to download MODIS datasets from NASA Earthdata.

We have developed and tested functions that group together the commands we want, ultimately arriving at the function `get_modis_files(doy,year,tiles,**kwargs)`.

We have seen ((if you've done the homework) that such code is re-useable and can directly be used for your assessed practical.

2.8.13 3.3 GDAL, and OGR masking

Table of Contents

[up to 3.0]

In this section, we'll look at combining both raster and vector data to provide a masked dataset ready to use. We will produce a combined dataset of leaf area index (LAI) over the UK derived from the MODIS sensor. The MODIS LAI product is produced every 4 days and it is provided spatially tiled. Each tile covers around 1200 km x 1200 km of the Earth's surface. Below you can see a map showing the MODIS tiling convention.

3.3.1 The MODIS LAI data

Let's first test your NASA login:

```
import geog0111.nasa_requests as nasa_requests
from geog0111.cylog import cylog
%matplotlib inline

url = 'https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/2018.09.30/'

# grab the HTML information
try:
    html = nasa_requests.get(url).text
    # test a few lines of the html
    if html[:20] == '<!DOCTYPE HTML PUBLI':
        print('this seems to be ok ... ')
        print('use cylog().login() anywhere you need to specify the tuple (username,
↪password)')
except:
    print('login error ... try entering your username password again')
    print('then re-run this cell until it works')
    cylog(init=True)
```

```
this seems to be ok ...
use cylog().login() anywhere you need to specify the tuple (username,password)
```

3.3.1.1 `try ... except ...`

Note that we have used a `try ... except` structure above to trap any errors.

```
import sys
try:
    # variable stupid not set
    print("I'm trying this but it will fail",stupid)
except NameError:
    '''
    trap the error
    (and ideally define some sensible behaviour)
    '''
    print("unset variable:",sys.exc_info()[1])
except:
    print("In case of other errors")
    print(sys.exc_info())
    # raise our own exception
    raise Exception('bad code')
```

```
unset variable: name 'stupid' is not defined
```

Generally, you should try to foresee the types of error you might generate, and provide specific traps for these so youy can control the code better.

In the case above, we allow the code execution to continue with a `NameError`, but raise a further exception in case of any other errors.

`sys.exc_info()` provides a tuple of information on what happened.

Exercise

- Write some code using `try ... except` to trap a `ZeroDivisionError`
- provide a sensible result in such a case

hint

If you divide by zero, the result will be infinity, which is often not what you want to happen. Instead, try dividing by a small number, such as that provided by `sys.float_info.epsilon`.

```
# do exercise here
```

3.3.1.2 Get data

You should by now be able to download MODIS data, but in this case, the data are provided (or downloaded for you) in the `data` folder as files `MCD15A3H.A2018273.h17v03.006.2018278143630.hdf` and `MCD15A3H.A2018273.h18v03.006.2018278143633.hdf` (and some files `*v04*hdf` we will need later) by running the code below.

```
from geog0111.geog_data import *

filenames = ['MCD15A3H.A2018273.h17v03.006.2018278143630.hdf', \
             'MCD15A3H.A2018273.h18v03.006.2018278143633.hdf',\
             'MCD15A3H.A2018273.h17v04.006.2018278143630.hdf',\
             'MCD15A3H.A2018273.h18v04.006.2018278143638.hdf']
destination_folder="data"

for file_name in filenames:
    f = procure_dataset(file_name,verbose=True,\
                        destination_folder=destination_folder)
    print(file_name,f)
```

```
MCD15A3H.A2018273.h17v03.006.2018278143630.hdf True
MCD15A3H.A2018273.h18v03.006.2018278143633.hdf True
MCD15A3H.A2018273.h17v04.006.2018278143630.hdf True
MCD15A3H.A2018273.h18v04.006.2018278143638.hdf True
```

We want to select the LAI layers, so let's have a look at the contents ('sub datasets') of one of the files.

To do this with `gdal`:

- make the full filename (folder name, plus the filename in that folder). Use `Path` for this, but convert to a string.
- open the file, store as `g`
- get the list `g.GetSubDatasets()` and loop over this

```
import gdal
from pathlib import Path
from geog0111.geog_data import *

filenames = ['MCD15A3H.A2018273.h17v03.006.2018278143630.hdf', \
             'MCD15A3H.A2018273.h18v03.006.2018278143633.hdf']
destination_folder="data"

for file_name in filenames:
    # form full filename as a string
    # and print with an underline of =
    file_name = Path(destination_folder).joinpath(file_name).as_posix()
```

(continues on next page)

(continued from previous page)

```
    print(file_name)
    print('='*len(file_name))

    # open the file as g
    g = gdal.Open(file_name)
    # loop over the subdatasets
    for d in g.GetSubDatasets():
        print(d)
```

```
data/MCD15A3H.A2018273.h17v03.006.2018278143630.hdf
===================================================
('HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2018273.h17v03.006.2018278143630.hdf":MOD_Grid_
↪MCD15A3H:Fpar_500m', '[2400x2400] Fpar_500m MOD_Grid_MCD15A3H (8-bit unsigned␣
↪integer)')
('HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2018273.h17v03.006.2018278143630.hdf":MOD_Grid_
↪MCD15A3H:Lai_500m', '[2400x2400] Lai_500m MOD_Grid_MCD15A3H (8-bit unsigned integer)
↪')
('HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2018273.h17v03.006.2018278143630.hdf":MOD_Grid_
↪MCD15A3H:FparLai_QC', '[2400x2400] FparLai_QC MOD_Grid_MCD15A3H (8-bit unsigned␣
↪integer)')
('HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2018273.h17v03.006.2018278143630.hdf":MOD_Grid_
↪MCD15A3H:FparExtra_QC', '[2400x2400] FparExtra_QC MOD_Grid_MCD15A3H (8-bit unsigned␣
↪integer)')
('HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2018273.h17v03.006.2018278143630.hdf":MOD_Grid_
↪MCD15A3H:FparStdDev_500m', '[2400x2400] FparStdDev_500m MOD_Grid_MCD15A3H (8-bit␣
↪unsigned integer)')
('HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2018273.h17v03.006.2018278143630.hdf":MOD_Grid_
↪MCD15A3H:LaiStdDev_500m', '[2400x2400] LaiStdDev_500m MOD_Grid_MCD15A3H (8-bit␣
↪unsigned integer)')
data/MCD15A3H.A2018273.h18v03.006.2018278143633.hdf
===================================================
('HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2018273.h18v03.006.2018278143633.hdf":MOD_Grid_
↪MCD15A3H:Fpar_500m', '[2400x2400] Fpar_500m MOD_Grid_MCD15A3H (8-bit unsigned␣
↪integer)')
('HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2018273.h18v03.006.2018278143633.hdf":MOD_Grid_
↪MCD15A3H:Lai_500m', '[2400x2400] Lai_500m MOD_Grid_MCD15A3H (8-bit unsigned integer)
↪')
('HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2018273.h18v03.006.2018278143633.hdf":MOD_Grid_
↪MCD15A3H:FparLai_QC', '[2400x2400] FparLai_QC MOD_Grid_MCD15A3H (8-bit unsigned␣
↪integer)')
('HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2018273.h18v03.006.2018278143633.hdf":MOD_Grid_
↪MCD15A3H:FparExtra_QC', '[2400x2400] FparExtra_QC MOD_Grid_MCD15A3H (8-bit unsigned␣
↪integer)')
('HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2018273.h18v03.006.2018278143633.hdf":MOD_Grid_
↪MCD15A3H:FparStdDev_500m', '[2400x2400] FparStdDev_500m MOD_Grid_MCD15A3H (8-bit␣
↪unsigned integer)')
('HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2018273.h18v03.006.2018278143633.hdf":MOD_Grid_
↪MCD15A3H:LaiStdDev_500m', '[2400x2400] LaiStdDev_500m MOD_Grid_MCD15A3H (8-bit␣
↪unsigned integer)')
```

So we see that the data is in `HDF4` format, and that it has a number of layers. The dataset/layer we're interested in

`HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2018273.h18v03.006.2018278143633.hdf":MOD_Grid_MCD15A3H:Lai_500m`.

3.3.1.3 File Naming Convention

This section taken from NASA MODIS product page.

Example File Name:

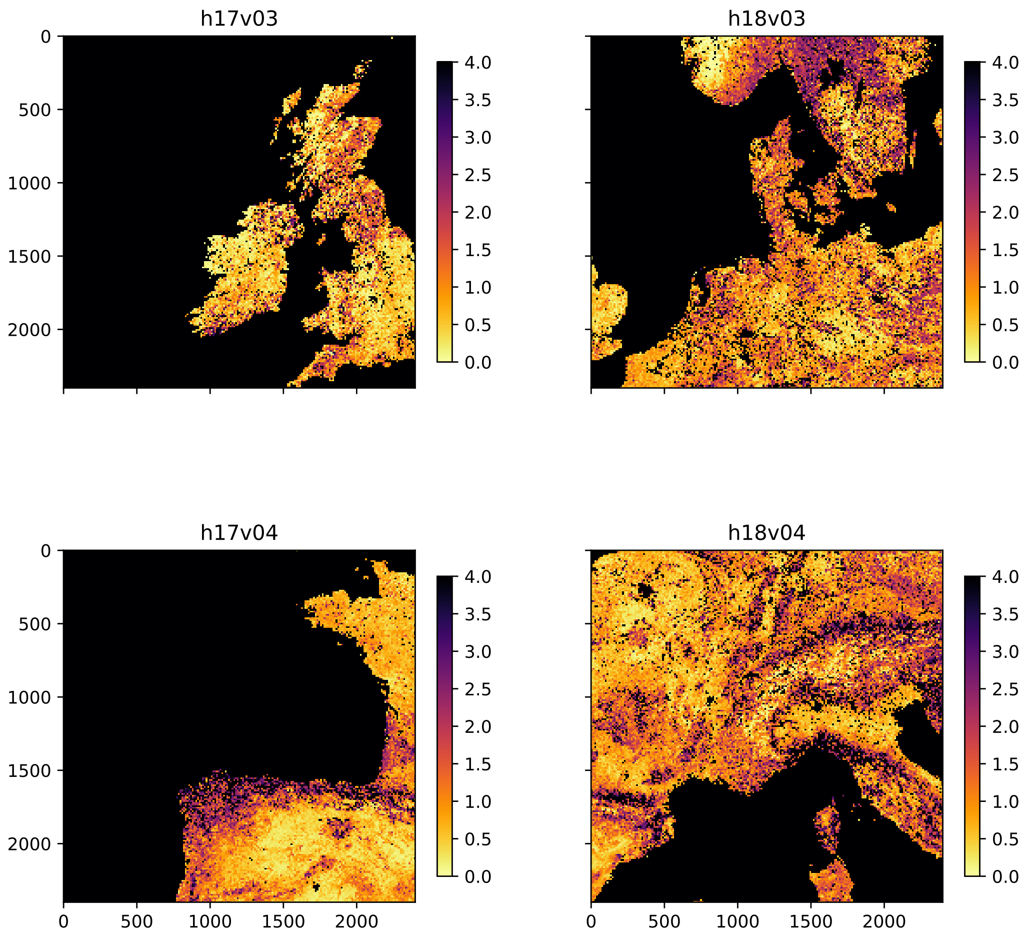
`data/MOD10A1.A2000055.h15v01.006.2016061160800.hdf`

`FOLDER/MOD[PID].A[YYYY][DDD].h[NN]v[NN].[VVV].[yyyy][ddd][hhmmss].hdf`

Refer to Table 3.3.1 for descriptions of the file name variables listed above.

Variable	Description
FOLDER	folder/directory name of file
MOD	MODIS/Terra (`MCD` means combined)
PID	Product ID
A	Acquisition date follows
YYYY	Acquisition year
DDD	Acquisition day of year
h[NN]v[NN]	Horizontal tile number and vertical tile number (see Grid for details.)
VVV	Version (Collection) number
yyyy	Production year
ddd	Production day of year
hhmmss	Production hour/minute/second in GMT
.hdf	HDF-EOS formatted data file

Table 3.3.1. Variables in the MODIS File Naming Convention



3.3.1.2 Dataset Naming Convention

Example Dataset Name:

```
HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2018273.h18v03.006.2018278143633.
hdf":MOD_Grid_MCD15A3H:Lai_500m
```

```
FORMAT:"FILENAME":MOD_Grid_PRODUCT:LAYER
```

Variable	Description
FORMAT	file format, `HDF4_EOS:EOS_GRID`
FILENAME	dataset file name, see below
PRODUCT	MODIS product code e.g. `MCD15A3H`
LAYER	sub-dataset name e.g. `Lai_500m`

Table 3.3.2. Variables in the MODIS Dataset Naming Convention

Exercise E3.3.1

- Check you're happy that the other datasets (e.g. `LaiStdDev_500m`) follow the same convention as `Lai_500m`
- work out what the dataset/layer name would be for the dataset product `MOD10A1` version `6` for the 1st January 2018, for tile `h25v06` for the layer `NDSI_Snow_Cover`. You will find product information on the relevant NASA page. You may not be able to access the production date/time, but just put a placeholder for that now.
- phrase the filename and layer name as 'f' strings, e.g. starting `f'HDF4_EOS:EOS_GRID:"{filename}":MOD_Grid_{}'` etc.

Hint:

You can explore the filenames by looking into the Earthdata link.

`images/BROWSE.MYD10A1.A2018001.h25v05.006.2018003025825.1.jpg`

```
# do exercise here
```

3.3.2 MODIS dataset access

3.3.2.1 `gdal.ReadAsArray()`

We can now access the dataset names and open the datasets in `gdal` directly, e.g.:

`HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2018273.h18v03.006.2018278143633.hdf":MOD_Grid_MCD15A3H:Lai_500m`

We can read the dataset with `g.ReadAsArray()`, after we have opened it. It returns a numpy array.

```
import gdal
import numpy as np

filename = 'data/MCD15A3H.A2018273.h17v03.006.2018278143630.hdf'
dataset_name = f'HDF4_EOS:EOS_GRID:"{filename:s}":MOD_Grid_MCD15A3H:Lai_500m'
print(f"dataset: {dataset_name}")

g = gdal.Open(dataset_name)
data = g.ReadAsArray()

print(type(data))
print('max:',data.max())
```

(continues on next page)

(continued from previous page)

```
print('max:',data.min())
# get unique values, for interst
print('unique values:',np.unique(data))
```

```
dataset: HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2018273.h17v03.006.2018278143630.hdf":MOD_
↪Grid_MCD15A3H:Lai_500m
<class 'numpy.ndarray'>
max: 255
max: 0
unique values: [  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16 ␣
↪17
  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53
  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70 250
 253 254 255]
```

Exercise E3.3.2

- print out some further summary statistics of the dataset
- print out the data type and `shape`
- how many rows and columns does the dataset have?

```
# do exercise here
```

3.3.2.2 Metadata

There will generally be a set of metadata associated with a geospatial dataset. This will describe e.g. the processing chain, special codes in the dataset, and projection and other information.

In `gdal`, w access the metedata using `g.GetMetadata()`. A dictionary is returned.

```
filename = 'data/MCD15A3H.A2018273.h17v03.006.2018278143630.hdf'
dataset_name = f'HDF4_EOS:EOS_GRID:"{filename:s}":MOD_Grid_MCD15A3H:Lai_500m'
g = gdal.Open(dataset_name)

print ("\nMetedata Keys:\n")
# get the metadata dictionary keys
for k in g.GetMetadata().keys():
    print(k)
```

```
Metedata Keys:

add_offset
add_offset_err
ALGORITHMPACKAGEACCEPTANCEDATE
ALGORITHMPACKAGEMATURITYCODE
ALGORITHMPACKAGENAME
ALGORITHMPACKAGEVERSION
ASSOCIATEDINSTRUMENTSHORTNAME.1
ASSOCIATEDINSTRUMENTSHORTNAME.2
ASSOCIATEDPLATFORMSHORTNAME.1
ASSOCIATEDPLATFORMSHORTNAME.2
ASSOCIATEDSENSORSHORTNAME.1
```

(continues on next page)

(continued from previous page)

```
ASSOCIATEDSENSORSHORTNAME.2
AUTOMATICQUALITYFLAG.1
AUTOMATICQUALITYFLAGEXPLANATION.1
calibrated_nt
CHARACTERISTICBINANGULARSIZE500M
CHARACTERISTICBINSIZE500M
DATACOLUMNS500M
DATAROWS500M
DAYNIGHTFLAG
DESCRREVISION
EASTBOUNDINGCOORDINATE
ENGINEERING_DATA
EXCLUSIONGRINGFLAG.1
GEOANYABNORMAL
GEOESTMAXRMSERROR
GLOBALGRIDCOLUMNS500M
GLOBALGRIDROWS500M
GRANULEBEGINNINGDATETIME
GRANULEDAYNIGHTFLAG
GRANULEENDINGDATETIME
GRINGPOINTLATITUDE.1
GRINGPOINTLONGITUDE.1
GRINGPOINTSEQUENCENO.1
HDFEOSVersion
HORIZONTALTILENUMBER
identifier_product_doi
identifier_product_doi_authority
INPUTPOINTER
LOCALGRANULEID
LOCALVERSIONID
LONGNAME
long_name
MAXIMUMOBSERVATIONS500M
MOD15A1_ANC_BUILD_CERT
MOD15A2_FILLVALUE_DOC
MOD15A2_FparExtra_QC_DOC
MOD15A2_FparLai_QC_DOC
MOD15A2_StdDev_QC_DOC
NADIRDATARESOLUTION500M
NDAYS_COMPOSITED
NORTHBOUNDINGCOORDINATE
NUMBEROFGRANULES
PARAMETERNAME.1
PGEVERSION
PROCESSINGCENTER
PROCESSINGENVIRONMENT
PRODUCTIONDATETIME
QAPERCENTCLOUDCOVER.1
QAPERCENTEMPIRICALMODEL
QAPERCENTGOODFPAR
QAPERCENTGOODLAI
QAPERCENTGOODQUALITY
QAPERCENTINTERPOLATEDDATA.1
QAPERCENTMAINMETHOD
QAPERCENTMISSINGDATA.1
QAPERCENTOTHERQUALITY
QAPERCENTOUTOFBOUNDSDATA.1
```

(continues on next page)

(continued from previous page)

```
RANGEBEGINNINGDATE
RANGEBEGINNINGTIME
RANGEENDINGDATE
RANGEENDINGTIME
REPROCESSINGACTUAL
REPROCESSINGPLANNED
scale_factor
scale_factor_err
SCIENCEQUALITYFLAG.1
SCIENCEQUALITYFLAGEXPLANATION.1
SHORTNAME
SOUTHBOUNDINGCOORDINATE
SPSOPARAMETERS
SYSTEMFILENAME
TileID
UM_VERSION
units
valid_range
VERSIONID
VERTICALTILENUMBER
WESTBOUNDINGCOORDINATE
_FillValue
```

Let’s look at some of these metadata fields:

```python
import gdal
import numpy as np

filename = 'data/MCD15A3H.A2018273.h17v03.006.2018278143630.hdf'
dataset_name = f'HDF4_EOS:EOS_GRID:"{filename:s}":MOD_Grid_MCD15A3H:Lai_500m'
print(f"dataset: {dataset_name}")

g = gdal.Open(dataset_name)
# get the metadata dictionary keys
for k in ["LONGNAME","CHARACTERISTICBINSIZE500M",\
          "MOD15A2_FILLVALUE_DOC",\
          "GRINGPOINTLATITUDE.1","GRINGPOINTLONGITUDE.1",\
          'scale_factor']:
    print(k,g.GetMetadata()[k])
```

```
dataset: HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2018273.h17v03.006.2018278143630.hdf":MOD_
↪Grid_MCD15A3H:Lai_500m
LONGNAME MODIS/Terra+Aqua Leaf Area Index/FPAR 4-Day L4 Global 500m SIN Grid
CHARACTERISTICBINSIZE500M 463.312716527778
MOD15A2_FILLVALUE_DOC MOD15A2 FILL VALUE LEGEND
255 = _Fillvalue, assigned when:
    * the MOD09GA suf. reflectance for channel VIS, NIR was assigned its _Fillvalue,␣
↪or
    * land cover pixel itself was assigned _Fillvalus 255 or 254.
254 = land cover assigned as perennial salt or inland fresh water.
253 = land cover assigned as barren, sparse vegetation (rock, tundra, desert.)
252 = land cover assigned as perennial snow, ice.
251 = land cover assigned as "permanent" wetlands/inundated marshlands.
250 = land cover assigned as urban/built-up.
249 = land cover assigned as "unclassified" or not able to determine.
```

(continues on next page)

(continued from previous page)

```
GRINGPOINTLATITUDE.1 49.7394264948349, 59.9999999946118, 60.0089388384779, 49.
↪7424953501575
GRINGPOINTLONGITUDE.1 -15.4860189105775, -19.9999999949462, 0.0325645816155362, 0.
↪0125638874822839
scale_factor 0.1
```

So we see that the datasets use the MODIS Sinusoidal projection. Also we see that the pixel spacing is around 463m, there is a scale factor of 0.1 to be applied etc.

Exercise E3.3.3

look at the metadata to discover:

- the number of rows and columns in the dataset
- the range of valid values

```
# do exercise here
```

3.3.3 Reading and displaying data

3.3.3.1 glob

Let us now suppose that we want to examine an hdf file that we have previously downloaded and stored in the directory data.

How can we get a view into this directory to the the names of the files there?

The answer to this is glob, which we can access from the pathlib module.

Let's look in the data directory:

```
from pathlib import Path

# look in this directory
in_directory = Path('data')

filenames = in_directory.glob('*')
print('files in the directory',in_directory,':')
for f in filenames:
    print(f.name)
```

```
files in the directory data :
MCD15A3H.A2016005.h17v04.006.2016013011406.hdf
MCD15A3H.A2016001.h18v04.006.2016007073726.hdf
MCD15A3H.A2016021.h18v03.006.2016026124743.hdf
MCD15A3H.A2018273.h17v04.006.2018278143630.hdf
MCD15A3H.A2016021.h17v03.006.2016026124738.hdf
MCD15A3H.A2016013.h17v03.006.2016020015242.hdf
airtravel.csv
MCD15A3H.A2016033.h17v04.006.2016043140634.hdf
MCD15A3H.A2016009.h18v03.006.2016014073048.hdf
test_image.bin
MCD15A3H.A2016033.h18v03.006.2016043140641.hdf
MCD15A3H.A2018273.h18v03.006.2018278143633.hdf
MCD15A3H.A2016021.h18v04.006.2016026124707.hdf
```

(continues on next page)

(continued from previous page)

```
MCD15A3H.A2016005.h17v03.006.2016013012017.hdf
satellites-1957-2019.gz
MCD15A3H.A2016017.h17v04.006.2016027192758.hdf
TM_WORLD_BORDERS-0.3.prj
MCD15A3H.A2016029.h17v03.006.2016043140323.hdf
saved_daymet.csv
TM_WORLD_BORDERS-0.3.zip
MCD15A3H.A2016013.h17v04.006.2016020020246.hdf
MCD15A3H.A2016009.h18v04.006.2016014074158.hdf
MCD15A3H.A2016017.h17v03.006.2016027192752.hdf
MCD15A3H.A2018273.h18v04.006.2018278143638.hdf
MCD15A3H.A2016029.h18v04.006.2016043140353.hdf
MCD15A3H.A2016025.h17v03.006.2016034034334.hdf
MCD15A3H.A2016013.h18v03.006.2016020014424.hdf
MCD15A3H.A2016025.h18v03.006.2016034034341.hdf
MCD15A3H.A2016009.h17v04.006.2016014072006.hdf
daymet_tmax.csv
MCD15A3H.A2016021.h17v04.006.2016026124414.hdf
MCD15A3H.A2016017.h18v03.006.2016027193558.hdf
MCD15A3H.A2016029.h17v04.006.2016043140330.hdf
MCD15A3H.A2016025.h17v04.006.2016034035837.hdf
MCD15A3H.A2016013.h18v04.006.2016020014435.hdf
MCD15A3H.A2016001.h17v03.006.2016007075833.hdf
MCD15A3H.A2016017.h18v04.006.2016027193356.hdf
TM_WORLD_BORDERS-0.3.dbf
MCD15A3H.A2016029.h18v03.006.2016043140341.hdf
Readme.txt
test.bin
TM_WORLD_BORDERS-0.3.shx
NOAA.csv
MCD15A3H.A2016025.h18v04.006.2016034034846.hdf
MCD15A3H.A2016033.h18v04.006.2016043140709.hdf
TM_WORLD_BORDERS-0.3.shp
MCD15A3H.A2016033.h17v03.006.2016043140622.hdf
MCD15A3H.A2016005.h18v03.006.2016013012348.hdf
MCD15A3H.A2016005.h18v04.006.2016013012025.hdf
MCD15A3H.A2016037.h17v03.006.2016043140850.hdf
MCD15A3H.A2016009.h17v03.006.2016014071957.hdf
MCD15A3H.A2018273.h17v03.006.2018278143630.hdf
MCD15A3H.A2016001.h17v04.006.2016007074809.hdf
MCD15A3H.A2016001.h18v03.006.2016007073724.hdf
```

We use the argument `'data/*'` where `*` is a wildcard. Any filenames that match this pattern will be returned as a list.

If we want the list sorted, we need to use the `sorted()` method. This is similar to the list `sort` we have seen previously, but returns the sorted list.

The wildcard `*` here means a match to zero or more characters, so this is matching all names in the directory `data`. The wildcard `**` would mean all files here and all sub-directories.

We could be more subtle with this, e.g. matching only files ending `hdf`:

```
from pathlib import Path

filenames = sorted(Path('data').glob('*'))
```

(continues on next page)

(continued from previous page)

```
for f in filenames:
    print(f.name)
```

```
MCD15A3H.A2016001.h17v03.006.2016007075833.hdf
MCD15A3H.A2016001.h17v04.006.2016007074809.hdf
MCD15A3H.A2016001.h18v03.006.2016007073724.hdf
MCD15A3H.A2016001.h18v04.006.2016007073726.hdf
MCD15A3H.A2016005.h17v03.006.2016013012017.hdf
MCD15A3H.A2016005.h17v04.006.2016013011406.hdf
MCD15A3H.A2016005.h18v03.006.2016013012348.hdf
MCD15A3H.A2016005.h18v04.006.2016013012025.hdf
MCD15A3H.A2016009.h17v03.006.2016014071957.hdf
MCD15A3H.A2016009.h17v04.006.2016014072006.hdf
MCD15A3H.A2016009.h18v03.006.2016014073048.hdf
MCD15A3H.A2016009.h18v04.006.2016014074158.hdf
MCD15A3H.A2016013.h17v03.006.2016020015242.hdf
MCD15A3H.A2016013.h17v04.006.2016020020246.hdf
MCD15A3H.A2016013.h18v03.006.2016020014424.hdf
MCD15A3H.A2016013.h18v04.006.2016020014435.hdf
MCD15A3H.A2016017.h17v03.006.2016027192752.hdf
MCD15A3H.A2016017.h17v04.006.2016027192758.hdf
MCD15A3H.A2016017.h18v03.006.2016027193558.hdf
MCD15A3H.A2016017.h18v04.006.2016027193356.hdf
MCD15A3H.A2016021.h17v03.006.2016026124738.hdf
MCD15A3H.A2016021.h17v04.006.2016026124414.hdf
MCD15A3H.A2016021.h18v03.006.2016026124743.hdf
MCD15A3H.A2016021.h18v04.006.2016026124707.hdf
MCD15A3H.A2016025.h17v03.006.2016034034334.hdf
MCD15A3H.A2016025.h17v04.006.2016034035837.hdf
MCD15A3H.A2016025.h18v03.006.2016034034341.hdf
MCD15A3H.A2016025.h18v04.006.2016034034846.hdf
MCD15A3H.A2016029.h17v03.006.2016043140323.hdf
MCD15A3H.A2016029.h17v04.006.2016043140330.hdf
MCD15A3H.A2016029.h18v03.006.2016043140341.hdf
MCD15A3H.A2016029.h18v04.006.2016043140353.hdf
MCD15A3H.A2016033.h17v03.006.2016043140622.hdf
MCD15A3H.A2016033.h17v04.006.2016043140634.hdf
MCD15A3H.A2016033.h18v03.006.2016043140641.hdf
MCD15A3H.A2016033.h18v04.006.2016043140709.hdf
MCD15A3H.A2016037.h17v03.006.2016043140850.hdf
MCD15A3H.A2018273.h17v03.006.2018278143630.hdf
MCD15A3H.A2018273.h17v04.006.2018278143630.hdf
MCD15A3H.A2018273.h18v03.006.2018278143633.hdf
MCD15A3H.A2018273.h18v04.006.2018278143638.hdf
NOAA.csv
Readme.txt
TM_WORLD_BORDERS-0.3.dbf
TM_WORLD_BORDERS-0.3.prj
TM_WORLD_BORDERS-0.3.shp
TM_WORLD_BORDERS-0.3.shx
TM_WORLD_BORDERS-0.3.zip
airtravel.csv
daymet_tmax.csv
satellites-1957-2019.gz
saved_daymet.csv
test.bin
test_image.bin
```

Exercise 3.3.4

- adapt the code above to return only hdf filenames for the tile h18v03

```
# do exercise here
```

3.3.3.2 reading and displaying image data

Let's now read some data as above.

we do this with:

```
g.Open(gdal_fname)
data = g.ReadAsArray()
```

Originally the data are `uint8` (unsigned 8 bit data), but we need to multiply them by `scale_factor` (0.1 here) to convert to physical units. This also casts the data type to `float`.

We can straightforwardly plot the images using `matplotlib`. We first importt the library:

```
import matplotlib.pylab as plt
```

Then set up the figure size:

```
plt.figure(figsize=(10,10))
```

Plot the image:

```
plt.imshow( data, vmin=0, vmax=6,cmap=plt.cm.inferno_r)
```

where here `data` is a 2-D dataset. We can set limits to the image scaling (`vmin`, `vmax`), so that we emphasise a particular range of values, and we can apply custom colourmaps (`cmap=plt.cm.inferno_r`).

Finally here, we set a title, and plot a colour wedge to show the data scale. The `scale=0.8` here allows us to align the size of the scale with the plotted image size.

```
plt.title(dataset_name)
plt.colorbar(shrink=0.8)
```

If we want to save the plotted image to a file, e.g. in the directory `images`, we use:

```
plt.savefig(out_filename)
```

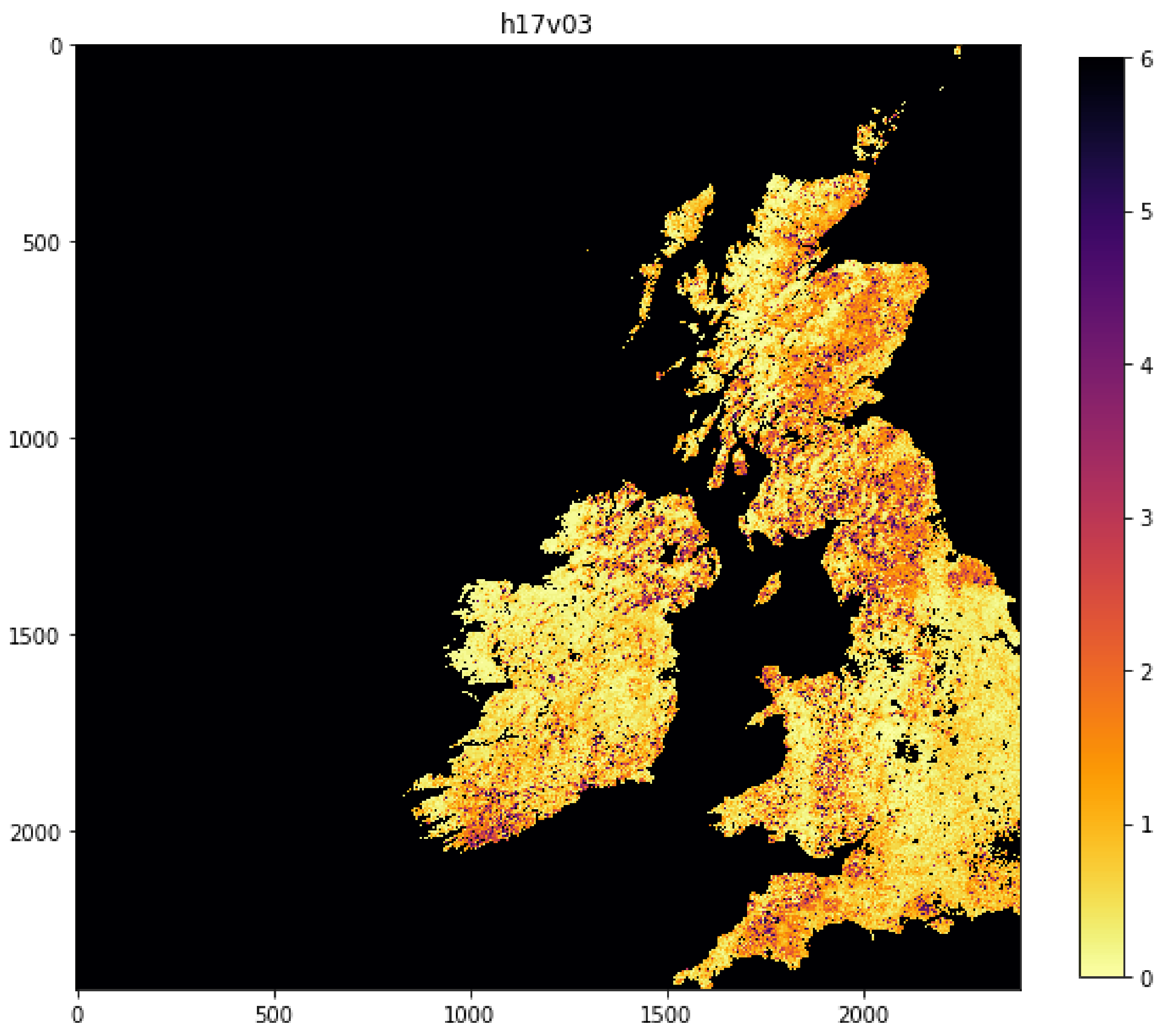
```
import gdal
from pathlib import Path
import matplotlib.pylab as plt

# get only v03 hdf names
filenames = sorted(Path('data').glob('*2018*v03*.hdf'))


out_directory = Path('images')

for filename in filenames:
    # pull the tile name from the filename
    # to use as plot title
    tile = filename.name.split('.')[2]
```

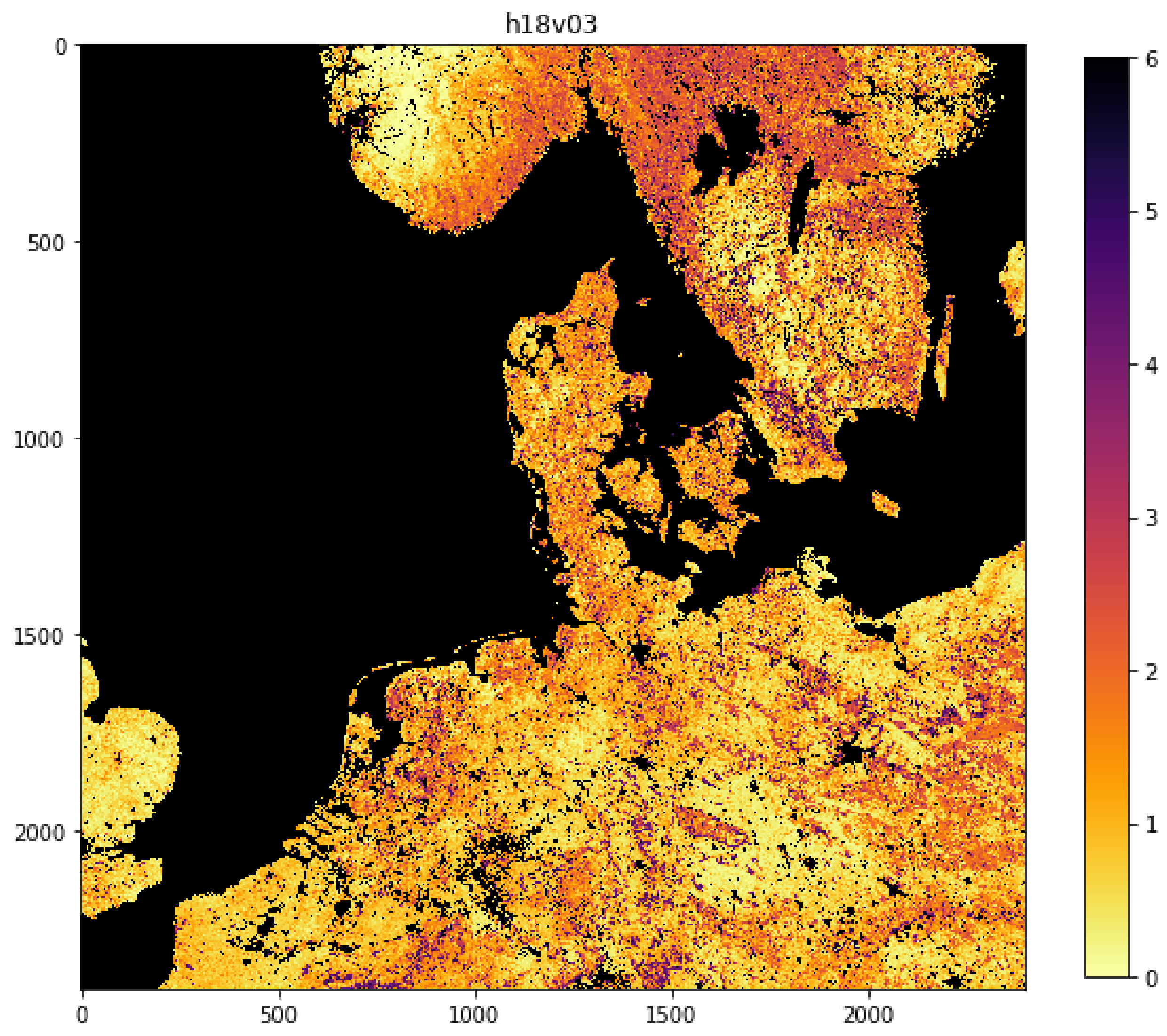
(continues on next page)

(continued from previous page)

```
    dataset_name = f'HDF4_EOS:EOS_GRID:"{str(filename):s}\":MOD_Grid_MCD15A3H:Lai_500m
↪'
    g = gdal.Open(dataset_name)
    data = g.ReadAsArray()
    scale_factor = float(g.GetMetadata()['scale_factor'])

    print(dataset_name,scale_factor)
    print('*'*len(dataset_name))
    print(type(data),data.dtype,data.shape,'\n')

    data = data * scale_factor
    print(type(data),data.dtype,data.shape,'\n')
    plt.figure(figsize=(10,10))
    plt.imshow( data, vmin=0, vmax=6,cmap=plt.cm.inferno_r)
    plt.title(tile)
    plt.colorbar(shrink=0.8)

    # save figure as png
    plot_name = filename.stem + '.png'
    print(plot_name)
    out_filename = out_directory.joinpath(plot_name)
    plt.savefig(out_filename)
```

```
HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2018273.h17v03.006.2018278143630.hdf":MOD_
↪Grid_MCD15A3H:Lai_500m 0.1
***************************************************************************************************
<class 'numpy.ndarray'> uint8 (2400, 2400)

<class 'numpy.ndarray'> float64 (2400, 2400)

MCD15A3H.A2018273.h17v03.006.2018278143630.png
HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2018273.h18v03.006.2018278143633.hdf":MOD_
↪Grid_MCD15A3H:Lai_500m 0.1
***************************************************************************************************
<class 'numpy.ndarray'> uint8 (2400, 2400)

<class 'numpy.ndarray'> float64 (2400, 2400)

MCD15A3H.A2018273.h18v03.006.2018278143633.png
```


h17v03
0
500
1000
1500
2000
0
500
1000
1500
2000
6
5
4
3
2
1
0



```
# Let's check the images we saved are there!
# and access some file info while we are here
# using pathlib
from pathlib import Path
from datetime import datetime

for f in Path('images').glob('MCD*2018*v03*.png'):

    # get the file size in bytes
    size_in_B = f.stat().st_size

    # get the file modification time (ns)
    mod_date_ns = f.stat().st_mtime_ns
    mod_date = datetime.fromtimestamp(mod_date_ns // 1000000000)

    print(f'{f} {size_in_B} Bytes {mod_date}')
```

```
images/MCD15A3H.A2018273.h18v03.006.2018278143633.png 297318 Bytes 2018-10-19 16:59:20
images/MCD15A3H.A2018273.h17v03.006.2018278143630.png 142654 Bytes 2018-10-19 16:59:19
```

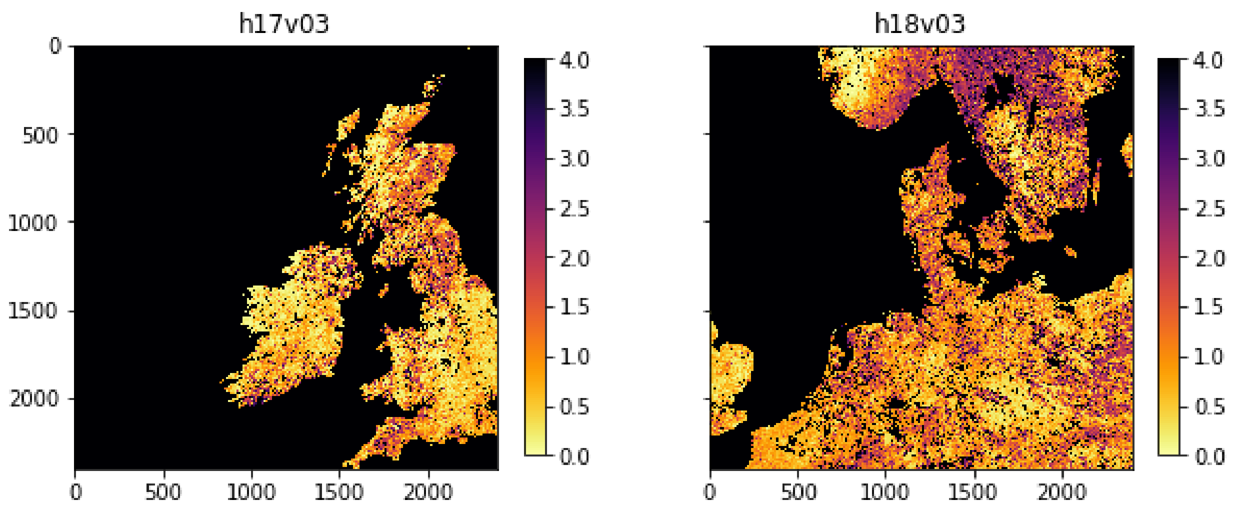
3.3.3.3 subplot plotting

Often, we want to have several figures on the same plot. We can do this with `plt.subplots()`:

The way we set the title and other features is slightly diifferent, but there are many example of different plot types on the web we can follow as examples.

```
import gdal
from pathlib import Path
import matplotlib.pylab as plt
import numpy as np

filenames = sorted(Path('data').glob('*2018*v03*.hdf'))

out_directory = Path('images')

'''
Set up subplots of 1 row x 2 columns
'''
fig, axs = plt.subplots(nrows=1, ncols=2, sharex=True, sharey=True,
                        figsize=(10,5))
# need to force axs collapse to a 2D array
# for indexing to be easy T here is transpose
# to get row/col the right way around
axs = np.array(axs).T.flatten()

for i,filename in enumerate(filenames):
    # pull the tile name from the filename
    # to use as plot title
    tile = filename.name.split('.')[2]


    dataset_name = f'HDF4_EOS:EOS_GRID:"{str(filename):s}\":MOD_Grid_MCD15A3H:Lai_500m
↪'
    g = gdal.Open(dataset_name)
    data = g.ReadAsArray() * float(g.GetMetadata()['scale_factor'])

    img = axs[i].imshow(data, interpolation="nearest", vmin=0, vmax=4,
                 cmap=plt.cm.inferno_r)
    axs[i].set_title(tile)
    plt.colorbar(img,ax=axs[i],shrink=0.7)

# save figure as pdf this time
plot_name = 'joinedup.pdf'
print(plot_name)
out_filename = out_directory.joinpath(plot_name)
plt.savefig(out_filename)
```

```
joinedup.pdf
```



Exercise 3.3.5

We now want to use the additional files:

```
MCD15A3H.A2018273.h17v04.006.2018278143630.hdf
MCD15A3H.A2018273.h18v04.006.2018278143638.hdf
```

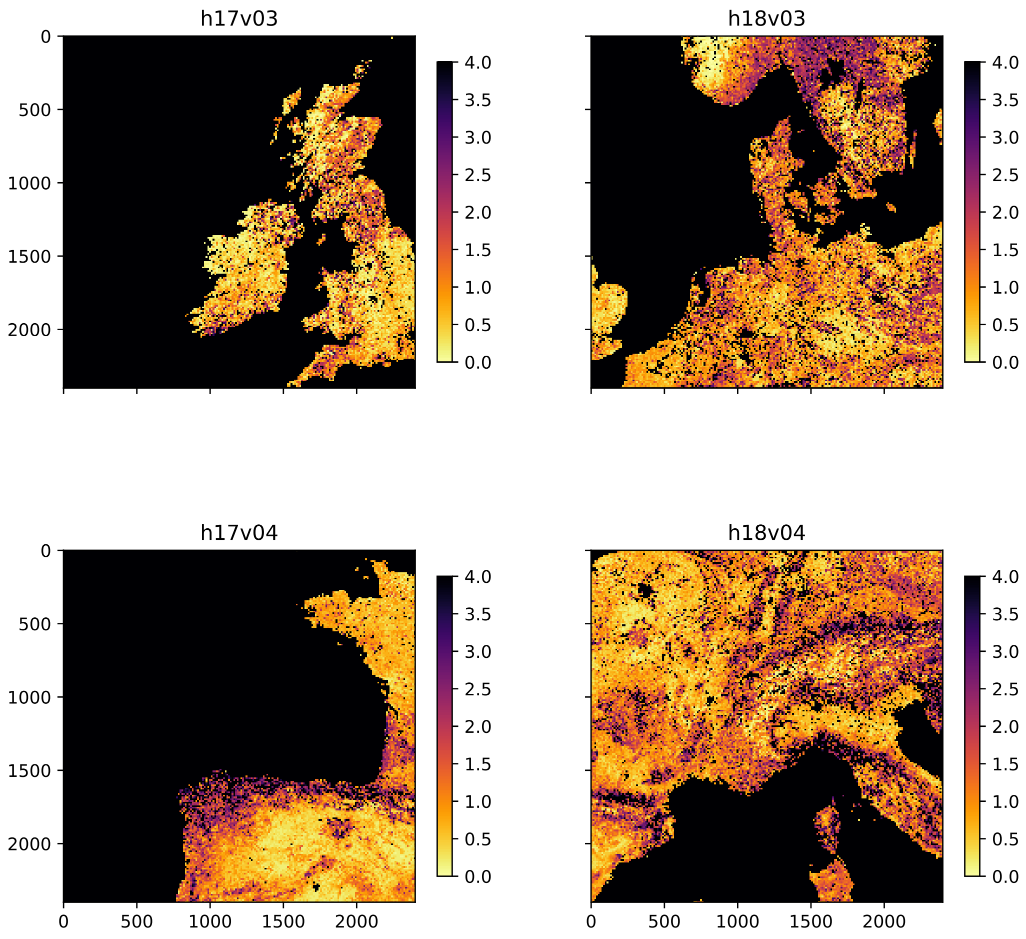
- copy and change the code above to use files of the pattern `*v0[3,4]*.hdf`
- use subplot as above to plot a 2x2 set of subplots of these data.

Hint

The code should look much like that above, but you need to give the fiuller list of filenames and set the subplot shape.

The code `[3,4]` in the pattern `*v0[3,4]*.hdf` means match either `3` or `4`, so the pattern must be `*v03*.hdf` or `*v03*.hdf`.

The result should look like:



```
# do exercise here
```

3.3.3.3 tile stitching

You may want to generate a single view of the 4 tiles.

We could achieve this by stitching things together "by hand"…

recipe:

- First, lets generate a 3D dataset with all 4 tiles, so we have the images stored as members of a list `data[0]`,`data[1]`,`data[2]` and `data[3]`:

```
data = []
for filename in filenames:
    dataname = f'HDF4_EOS:EOS_GRID:"{str(filename):s}":MOD_Grid_MCD15A3H:Lai_500m'
    g = gdal.Open(dataname)
    data.append(g.ReadAsArray() * scale)
```

- then, we produce vertical stacks of the first two and last two files. This can be done in various ways, but it is perhaps clearest to use `np.vstack()`

```
top = np.vstack([data[0],data[1]])
bot = np.vstack([data[2],data[3]])
```

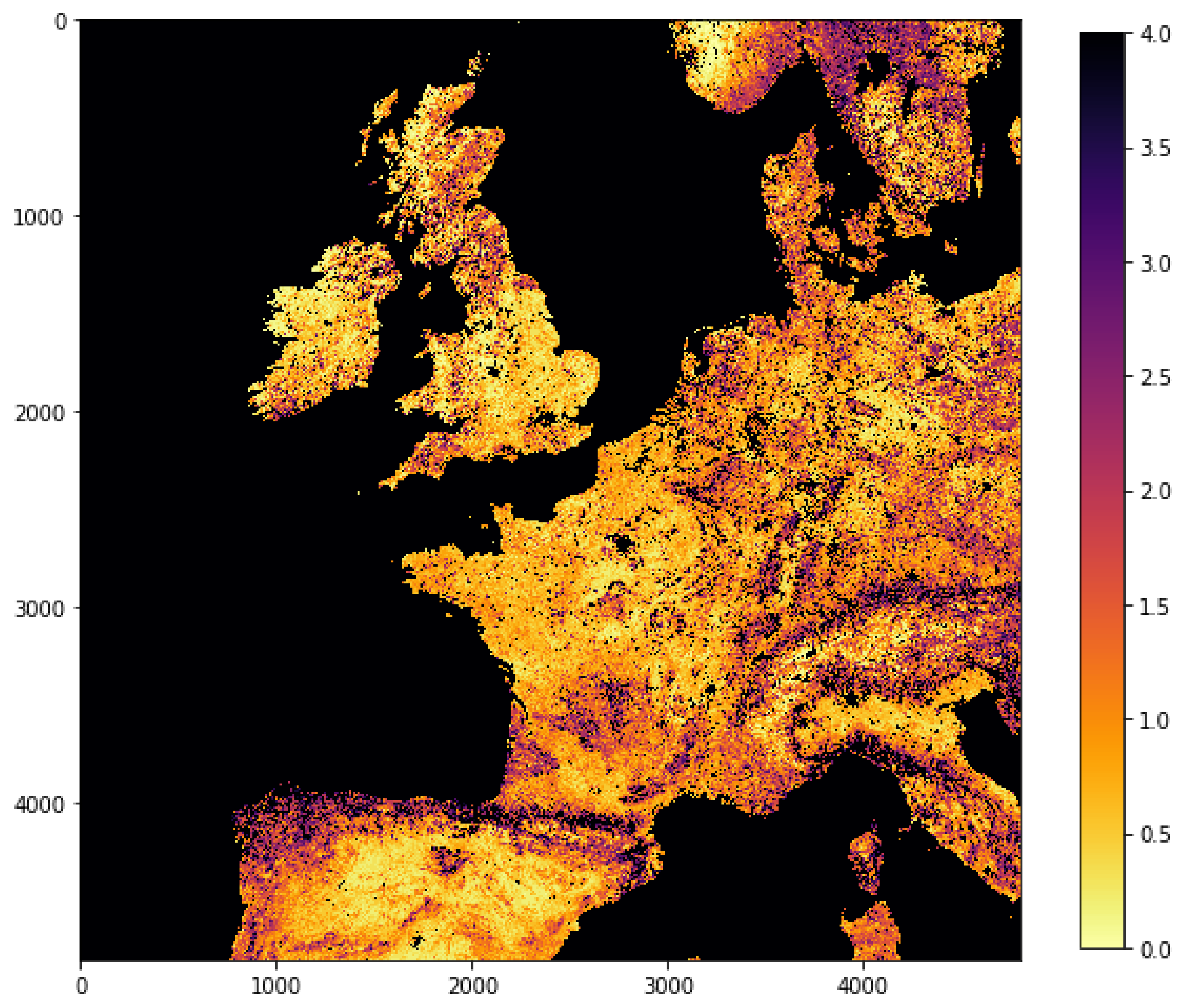
- then, produce a horizontal stack of these stacks:

```
lai_stich = np.hstack([top,bot])
```

and plot the dataset

```
import gdal
from pathlib import Path
import matplotlib.pylab as plt

scale = 0.1

filenames = sorted(Path('data').glob('*2018*v0*.hdf'))

data = []
for filename in filenames:
    dataname = f'HDF4_EOS:EOS_GRID:"{str(filename)}":MOD_Grid_MCD15A3H:Lai_500m'
    g = gdal.Open(dataname)
    # append each image to the data list
    data.append(g.ReadAsArray() * scale)

top = np.vstack([data[0],data[1]])
bot = np.vstack([data[2],data[3]])

lai_stich = np.hstack([top,bot])

plt.figure(figsize=(10,10))
plt.imshow(lai_stich, interpolation="nearest", vmin=0, vmax=4,
          cmap=plt.cm.inferno_r)
plt.colorbar(shrink=0.8)
```

```
<matplotlib.colorbar.Colorbar at 0x126b0fac8>
```



Exercise 3.3.6

- examine how the `vstack` and `hstack` methods work. Print out the shape of the array after stacking to appreciate this.
- how big (in pixels) is the whole dataset now?
- If a `float` is 64 bits, how many bytes is this data array likely to be?

```
# do exercise here
```

3.3.3.4 `gdal` virtual file

However, stitching in this way is problematic if you want to mosaic many tiles, as you need to read in all the data in memory. Also,some tiles may be missing. GDAL allows you to create a mosaic as virtual file format, using gdal.BuildVRT (check the documentation).

This function takes two inputs: the output filename (`stitch_up.vrt`) and a set of GDAL format filenames. It returns the open output dataset, so that we can check what it looks like with e.g. `gdal.Info`

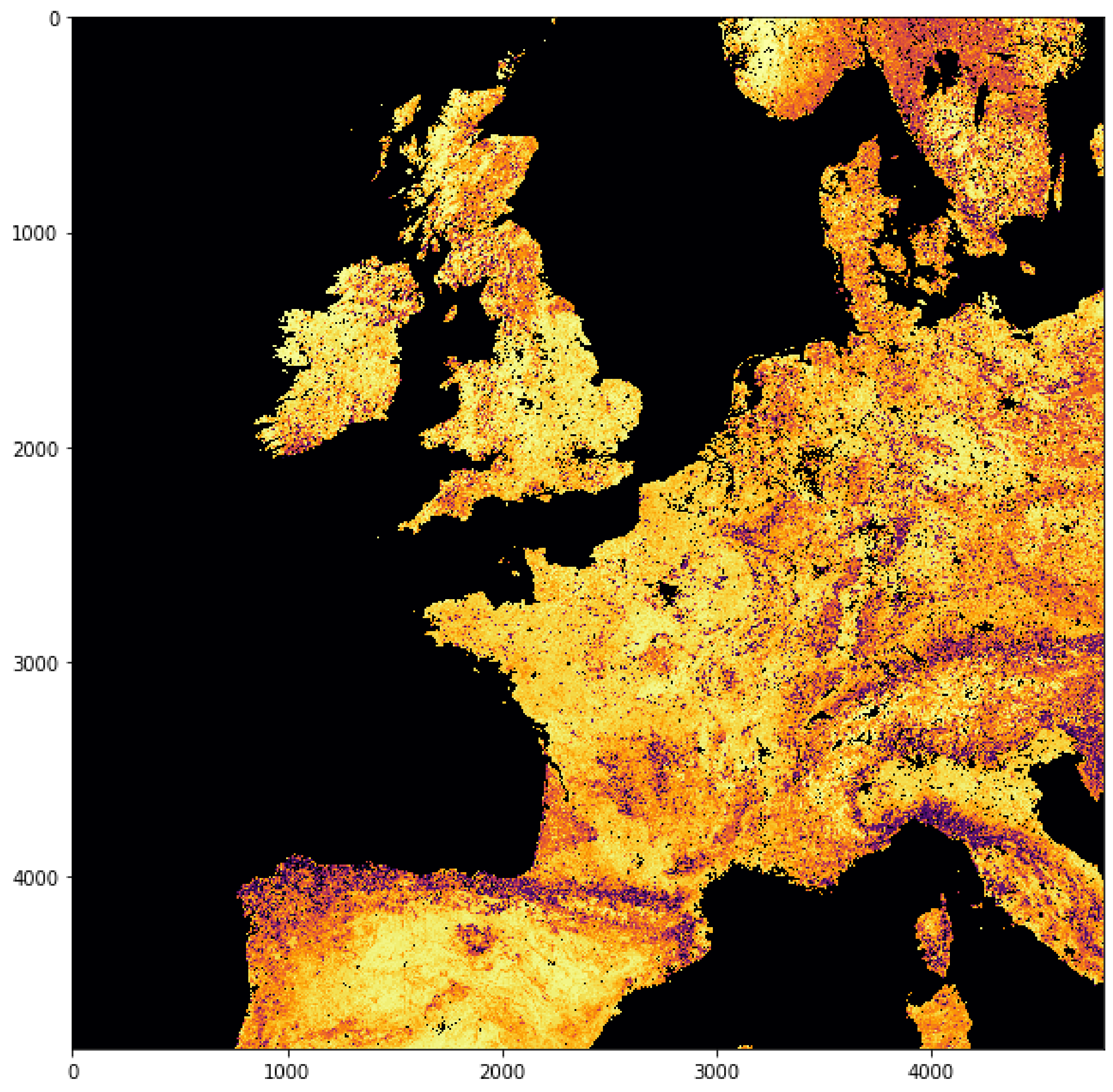
```
import gdal
from pathlib import Path

# need to convert filenames to strings
# which we can do with p.as_posix() or str(p)
filenames = sorted([p.as_posix() for p in Path('data').glob('*273*v0[3,4]*.hdf')])
datanames = [f'HDF4_EOS:EOS_GRID:"{str(filename)}":MOD_Grid_MCD15A3H:Lai_500m' \
                for filename in filenames]
stitch_vrt = gdal.BuildVRT("stitch_up.vrt", datanames)

print(gdal.Info(stitch_vrt))
```

```
Driver: VRT/Virtual Raster
Files: stitch_up.vrt
Size is 4800, 4800
Coordinate System is:
PROJCS["unnamed",
    GEOGCS["Unknown datum based upon the custom spheroid",
        DATUM["Not specified (based on custom spheroid)",
            SPHEROID["Custom spheroid",6371007.181,0]],
        PRIMEM["Greenwich",0],
        UNIT["degree",0.0174532925199433]],
    PROJECTION["Sinusoidal"],
    PARAMETER["longitude_of_center",0],
    PARAMETER["false_easting",0],
    PARAMETER["false_northing",0],
    UNIT["Meter",1]]
Origin = (-1111950.519667000044137,6671703.117999999783933)
Pixel Size = (463.312716527916677,-463.312716527708290)
Corner Coordinates:
Upper Left  (-1111950.520, 6671703.118) ( 20d 0' 0.00"W, 60d 0' 0.00"N)
Lower Left  (-1111950.520, 4447802.079) ( 13d 3'14.66"W, 40d 0' 0.00"N)
Upper Right ( 1111950.520, 6671703.118) ( 20d 0' 0.00"E, 60d 0' 0.00"N)
Lower Right ( 1111950.520, 4447802.079) ( 13d 3'14.66"E, 40d 0' 0.00"N)
Center      (       0.000, 5559752.598) (  0d 0' 0.01"E, 50d 0' 0.00"N)
Band 1 Block=128x128 Type=Byte, ColorInterp=Gray
  NoData Value=255
```

So we see that we now have 4800 columns by 4800 rows dataset, centered around 0 degrees North, 0 degrees W. Let's plot the data. . .

```
# stitch_vrt is an already opened GDAL dataset, needs to be read in
plt.figure(figsize=(10,10))
plt.imshow(stitch_vrt.ReadAsArray()*0.1,
           interpolation="nearest", vmin=0, vmax=6,
          cmap=plt.cm.inferno_r)
```

```
<matplotlib.image.AxesImage at 0x10a0ecda0>
```



3.3.4 The country borders dataset

A number of vectors with countries and administrative subdivisions are available. The TM_WORLD_BORDERS shapefile is popular and in the public domain. You can see it, and have a look at the data here. We need to download and unzip this file... We'll use requests as before, and we'll unpack the zip file using ` shutil.unpack_archive <https://docs.python.org/3/library/shutil.html#shutil.unpack_archive>`__

```
import requests
import shutil

tm_borders_url = "http://thematicmapping.org/downloads/TM_WORLD_BORDERS-0.3.zip"

r = requests.get(tm_borders_url)
with open("data/TM_WORLD_BORDERS-0.3.zip", 'wb') as fp:
```

(continues on next page)

(continued from previous page)

```
    fp.write (r.content)

shutil.unpack_archive("data/TM_WORLD_BORDERS-0.3.zip",
                      extract_dir="data/")
```

Make sure you have the relevant files available in your `data` folder! We can then inspect the dataset using the command line tool `ogrinfo`. We can call it from the shell by appending the `!` symbol, and select that we want to check only the data for the UK (stored in the `FIPS` field with value `UK`):

It is worth noting that using OGR's queries trying to match a string, the string needs to be surrounded by `'`. You can also use more complicated SQL queries if you wanted to.

```
!ogrinfo -nomd -geom=NO -where "FIPS='UK'"  data/TM_WORLD_BORDERS-0.3.shp TM_WORLD_
↪BORDERS-0.3
```

```
INFO: Open of `data/TM_WORLD_BORDERS-0.3.shp'
      using driver `ESRI Shapefile' successful.

Layer name: TM_WORLD_BORDERS-0.3
Geometry: Polygon
Feature Count: 1
Extent: (-180.000000, -90.000000) - (180.000000, 83.623596)
Layer SRS WKT:
GEOGCS["GCS_WGS_1984",
    DATUM["WGS_1984",
        SPHEROID["WGS_84",6378137.0,298.257223563]],
    PRIMEM["Greenwich",0.0],
    UNIT["Degree",0.0174532925199433],
    AUTHORITY["EPSG","4326"]]
FIPS: String (2.0)
ISO2: String (2.0)
ISO3: String (3.0)
UN: Integer (3.0)
NAME: String (50.0)
AREA: Integer (7.0)
POP2005: Integer64 (10.0)
REGION: Integer (3.0)
SUBREGION: Integer (3.0)
LON: Real (8.3)
LAT: Real (7.3)
OGRFeature(TM_WORLD_BORDERS-0.3):206
  FIPS (String) = UK
  ISO2 (String) = GB
  ISO3 (String) = GBR
  UN (Integer) = 826
  NAME (String) = United Kingdom
  AREA (Integer) = 24193
  POP2005 (Integer64) = 60244834
  REGION (Integer) = 150
  SUBREGION (Integer) = 154
  LON (Real) = -1.600
  LAT (Real) = 53.000
```

We inmediately see that the coordinates for the UK are in several polygons, and in WGS84 (Latitude and Longitude in decimal degrees). This is incompatible with the MODIS data (SIN projection), but fortunately GDAL understands about coordinate systems.

We can use GDAL to quickly apply the vector feature for the UK as a mask. There are several ways of doing this, but the simplest is to use gdal.Warp (the link is to the command line tool). In this case, we just want to create:

- an in-memory (i.e. not saved to a file) dataset. We can use the format `MEM`, so no file is written out.
- where the `FIPS` field is equal to `'UK'`, we want the LAI to show, elsewhere, we set it to some value to indicate "no data" (e.g. -999)

The mosaicked version of the MODIS LAI product is in called `stitch_up.vrt`. Since we're not saving the output to a file (`MEM` output option), we can leave the output as an empty string `""`. The shapefile comes with the `cutline` options:

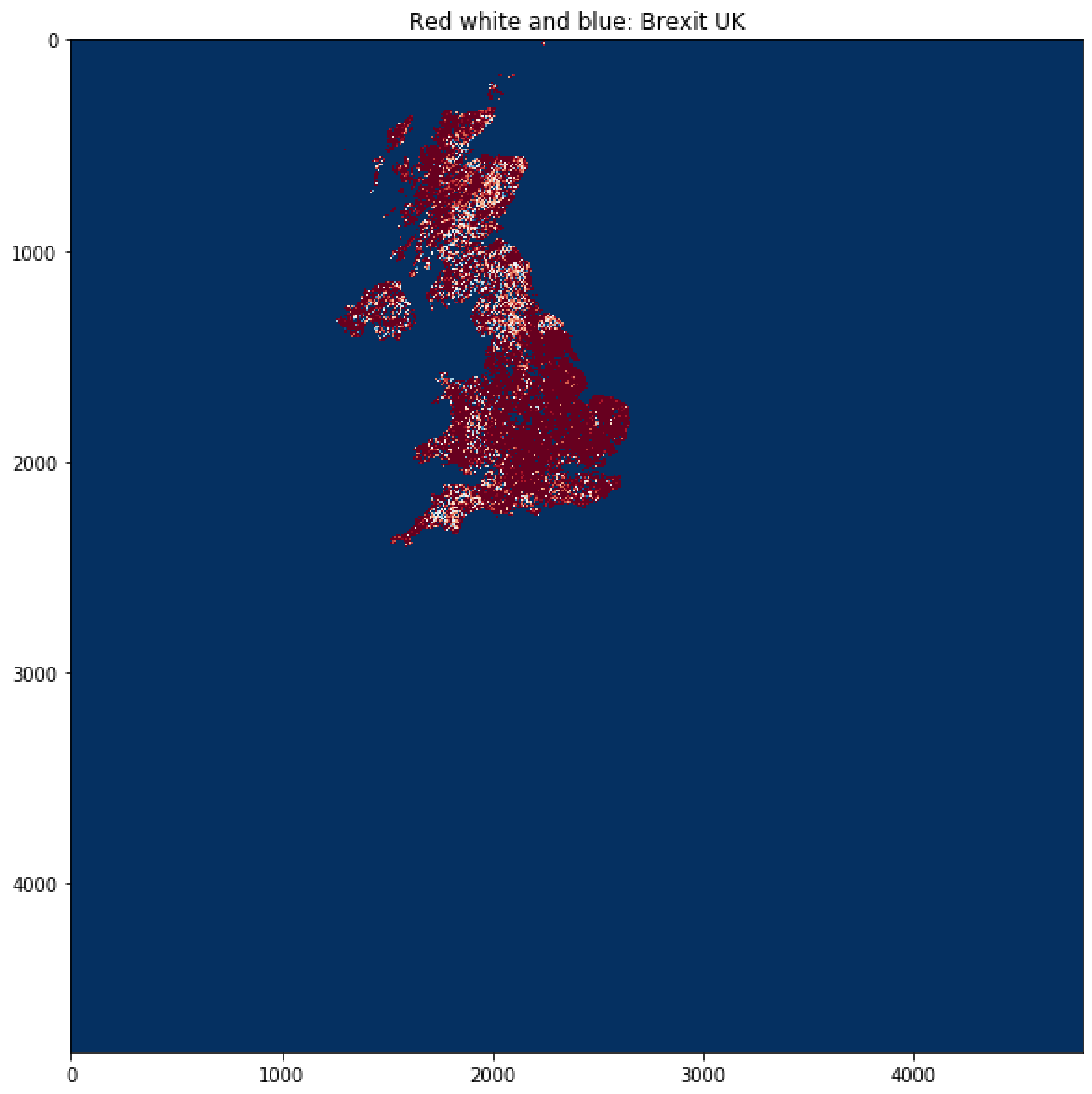
- `cutlineDSName` that's the name of the vector file we want to use as a cutline
- `cutlineWhere` that's the selection statement for the attribute table in the dataset.

To set the no data value to 200, we can use the option `dstNodata=200`. This is because very large values in the LAI product are already indicated to be invalid.

We can then just very quickly perform this and check. . .

```python
import gdal
import matplotlib.pylab as plt
from pathlib import Path

filenames = sorted([p.as_posix() for p in Path('data').glob('*2018*v0*.hdf')])
datanames = [f'HDF4_EOS:EOS_GRID:"{str(filename)}":MOD_Grid_MCD15A3H:Lai_500m' \
                for filename in filenames]
stitch_vrt = gdal.BuildVRT("stitch_up.vrt", datanames)


g = gdal.Warp("", "stitch_up.vrt",
         format = 'MEM',dstNodata=200,
          cutlineDSName = 'data/TM_WORLD_BORDERS-0.3.shp', cutlineWhere = "FIPS='UK'")

# read and plot data
masked_lai = g.ReadAsArray()*0.1
plt.figure(figsize=(10,10))
plt.title('Red white and blue: Brexit UK')
plt.imshow(masked_lai, interpolation="nearest", vmin=1, vmax=3,
          cmap=plt.cm.RdBu)
```

```
<matplotlib.image.AxesImage at 0x126942320>
```



So that works as expected, but since we haven't actually told GDAL anything about the output (other than apply the mask), we still have a 4800 pixel wide dataset.

You may want to crop it by looking for where the original dataset is valid (0 to 100 here). This will generally save a lot of computer memory. You'll be pleased to know that this is a great slicing application!

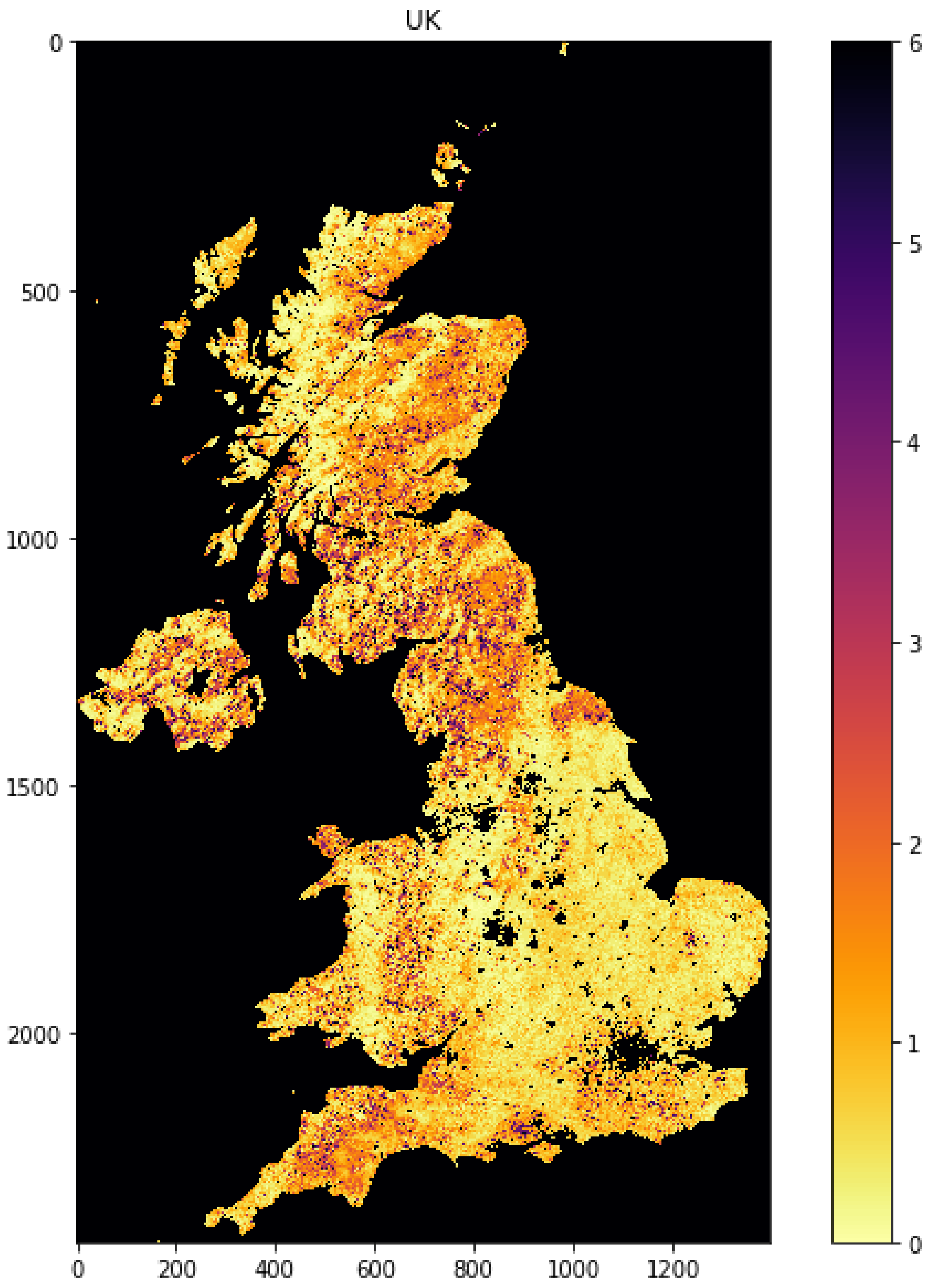
```python
import numpy as np

lai = g.ReadAsArray()

# data valid where lai <= 100 here
valid_mask = np.where(lai <= 100)

# work out the bounds of valid_mask
min_y      = valid_mask[0].min()
```

(continues on next page)

(continued from previous page)

```
max_y      = valid_mask[0].max() + 1

min_x      = valid_mask[1].min()
max_x      = valid_mask[1].max() + 1

# now slice, and scale LAI
lai = lai[min_y:max_y,
          min_x:max_x]*0.1

plt.figure(figsize=(10,10))
plt.imshow(lai, vmin=0, vmax=6,
           cmap=plt.cm.inferno_r)
plt.title('UK')
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x12a6486d8>
```



Exercise 3.3.7 Homework

- Develop a function that takes the list of dataset names and the information you passed to `gdal.Warp` (or a subset of this) and returns a cropped image of valid data.

- Use this function to show separate images of: France, Belgium, the Netherlands

```
# do exercise here
```

Exercise 3.3.8 Homework

- Download data for these same four tiles from the **MODIS snow cover** dataset for some particular date (in winter). Check the related quicklooks to see that the dataset isn't all covered in cloud.
- show the snow cover for one or more selected countries.
- calculate summary statistics for the datasets.

Hint the codes would be very similar to above, but watch out for the scaling factor not being the same (no scaling for the snow cover!). Also, watch out for the dataset being on a different NASA server to the LAI data (as in exercise above).

When you calculate summary statistics, make sure you ignore all invalid pixels. You could do that by generating a mask of the dataset (after you have clipped it) using `np.where()`, and only process those pixels, e.g.:

```
image[np.where(image<=100)].mean()
```

rather than

```
image.mean()
```

as the latter would include invalid pixels.

```
# do exercise here
```

2.8.14 3.4.4 Weighted interpolation

Table of Contents

3.4.4.1 Smoothing

There are many approaches to weighted interpolation. One such is to use a convolution smoothing operation.

In convolution, we combine a *signal* y with a *filter* f to achieve a filtered signal. For example, if we have an noisy signal, we will attempt to reduce the influence of high frequency information in the signal (a 'low pass' filter, as we let the low frequency information *pass*).

In this approach, we define a digital filter (convolution filter) that should be some sort of weighted average function. A typical filter is the Gaussian, defined by the parameter σ. The larger the value of σ, the 'wider' the filter, which means that the weighted average will be takjen over a greater extent.

We do not expect you to be overly concerned with the code in these sections below, as the main effort should be directed at understanding smoothing and interpolation at this point.

Let's look at this filter:

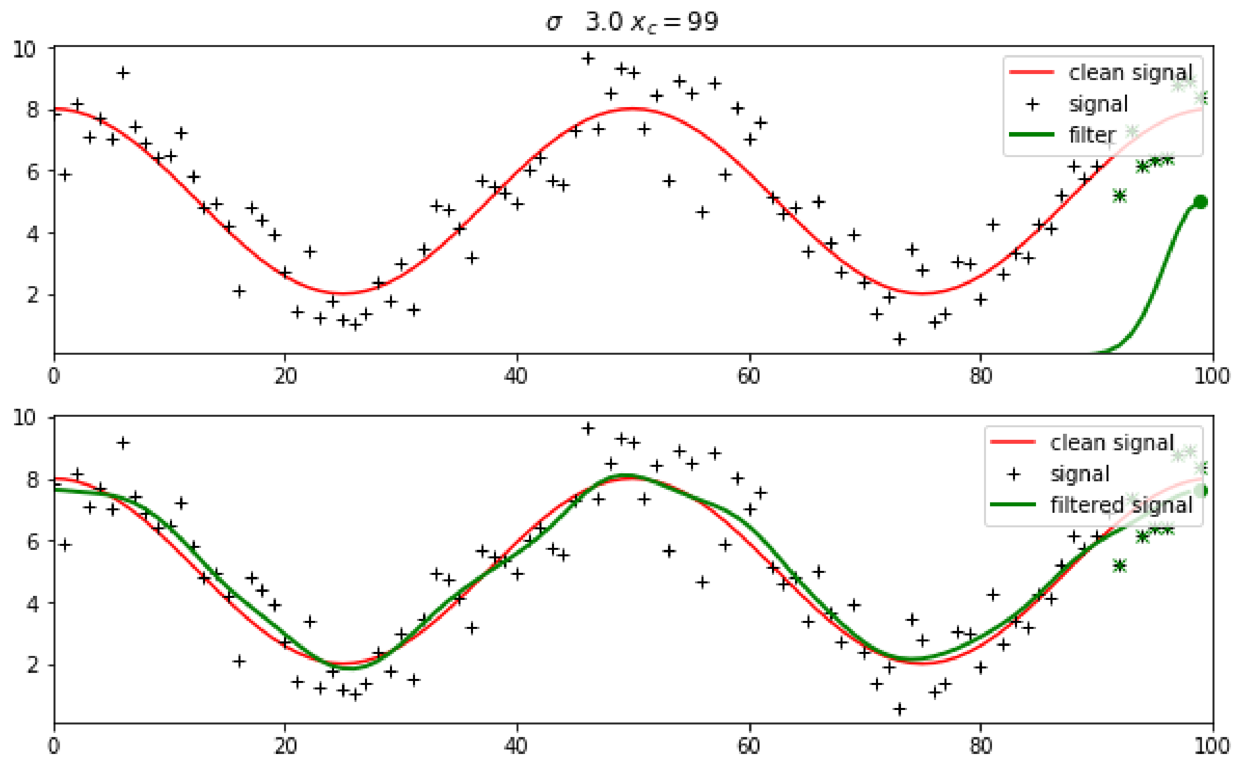
```
from IPython.display import HTML
from geog0111.demofilt1 import demofilt1

HTML(demofilt1().to_jshtml())
```

A digital (discrete) convolution uses a sampled signal and filter (represented on a grid).

Without going into the maths, the convolution operates by running the filter centred on x_c over the extent of the signal as illustrated below.

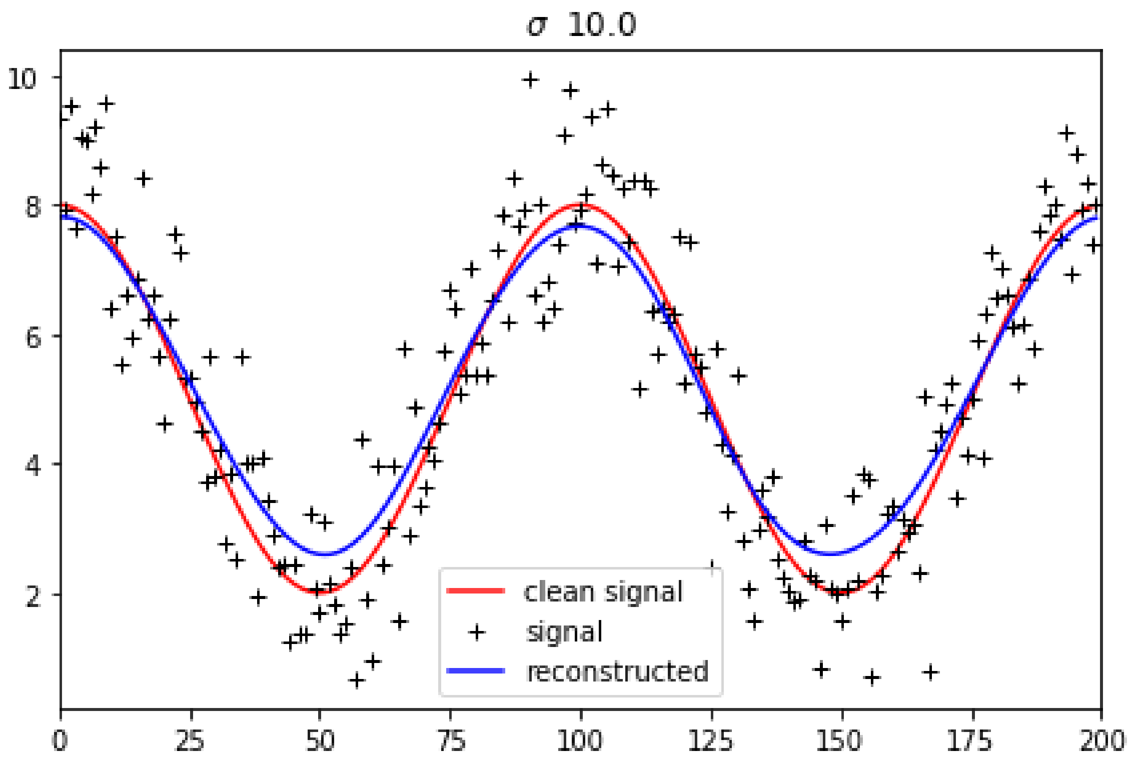
At each value of :math:'x_c', the filter is **multiplied by the signal**. Clearly, where the filter is zero, the influence of the signal is zero. So, the filter effectively selects a local window of data points (shown as green crosses below). The result, i.e. the filtered signal, is simply the weighted average of these local samples. This can be seen in the lower panel, where the green dot shows the filtered signal at x_c.

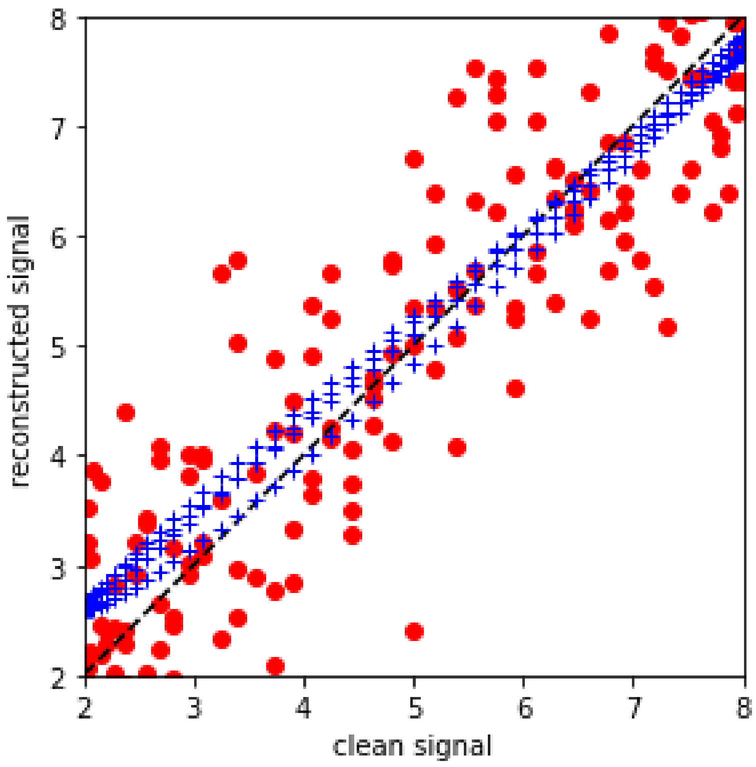
```
from IPython.display import HTML
from geog0111.demofilt3 import demofilt3

HTML(demofilt3().to_jshtml())
```



The convolution of a low pass ('smoothing') filter with a signal results in a smoothing of the signal:

```
from IPython.display import HTML
from geog0111.demofilt2 import demofilt2

HTML(demofilt2().to_jshtml())
```





The degree of smoothing is greater, the larger the value of σ (i.e. the broader the convolution filter). Eventually, the signal would go 'flat', i.e. be a constant (weighted mean) value, if we used a very broad filter.

For small values of σ though, the signal is still rather noisy.

There is a trade-off then between noise suppression and the degree of generalisation.

We could define some concept of an 'optimium' value of σ, for example using cross validation. But often we can decide empirically an appropriate filter width.

We perform the convolution using:

```
convolve1d(y, filter,mode='wrap')
```

from the `scipy.ndimage.filters` library.

The first argument is the signal to be filtered. The second is the filter (the Gaussian here).

We set

```
mode='wrap'
```

which defines the boundary conditions to be periodic ('wrapped around'). This would generally be appropriate for time series e.g. of one year.

3.4.4.2 weighted smoothing

In the convolution examples above, we apply the smoothing equally to all samples of the signal.

What if we knew that some samples were 'better' (quality) than others? What if some samples were missing? How could we incorporate this information?

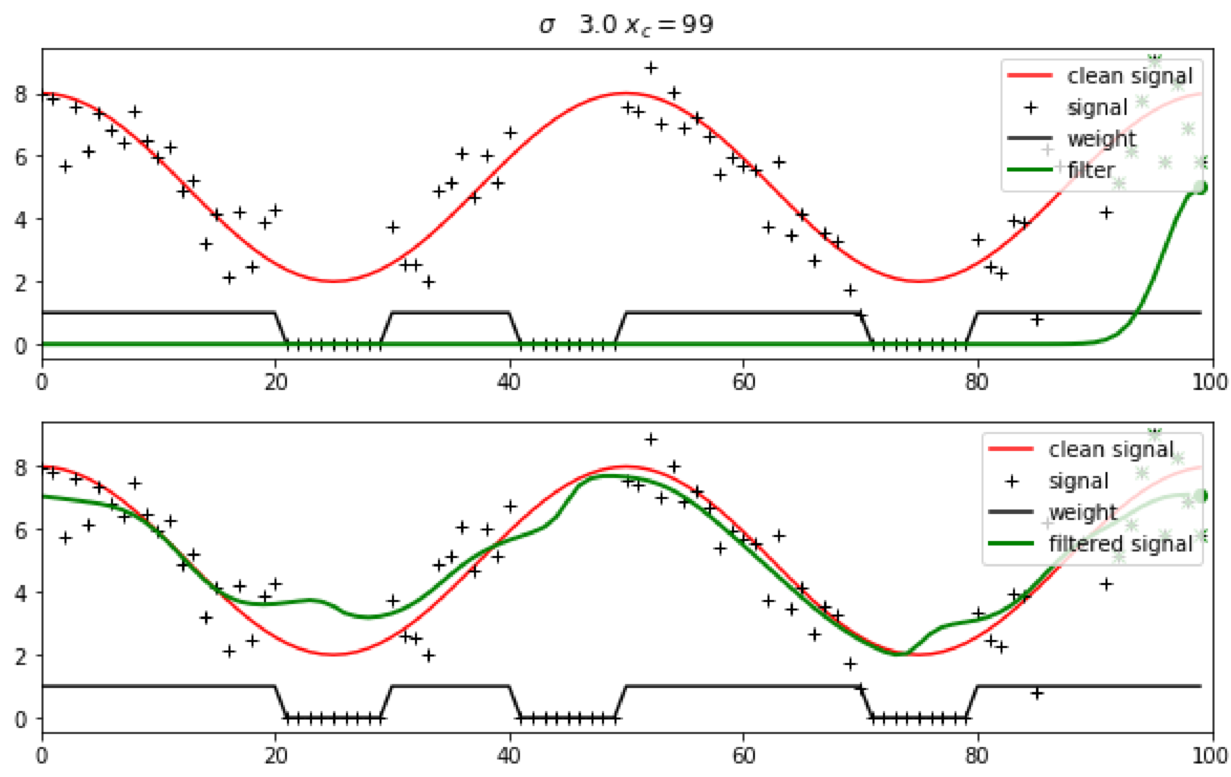
The answer is to *weight* the convolution.

For each sample point in the signal, we define a *weight*, where the weight is high if we trust the data point and low if we don't trust it much. We apply a weight of zero if a data point is missing.

Then, the weighted convolution is simply the result of applying the filter to (weight $\times$ signal), divided by the result of applying the filter to the weight alone. You can think of the denominator here as a form of 're-normalisation' of the filter.

We illustrate this by removing samples from the example above, and giving a weight of zero for the 'missing' observations.

```
from IPython.display import HTML
from geog0111.demofilt4 import demofilt4

HTML(demofilt4().to_jshtml())
```



The result is not as good as we got above, but that is hardly surprising: we are interpolating here and it is hard to interpolate over large gaps.

Since the gaps are quite large, we might benefit from using a larger filter extent. This is then liable to degrade the quality somewhat (over smooth) in data rich areas. These are typical trade-offs we must balance.

```
from IPython.display import HTML
from geog0111.demofilt5 import demofilt5

HTML(demofilt5().to_jshtml())
```

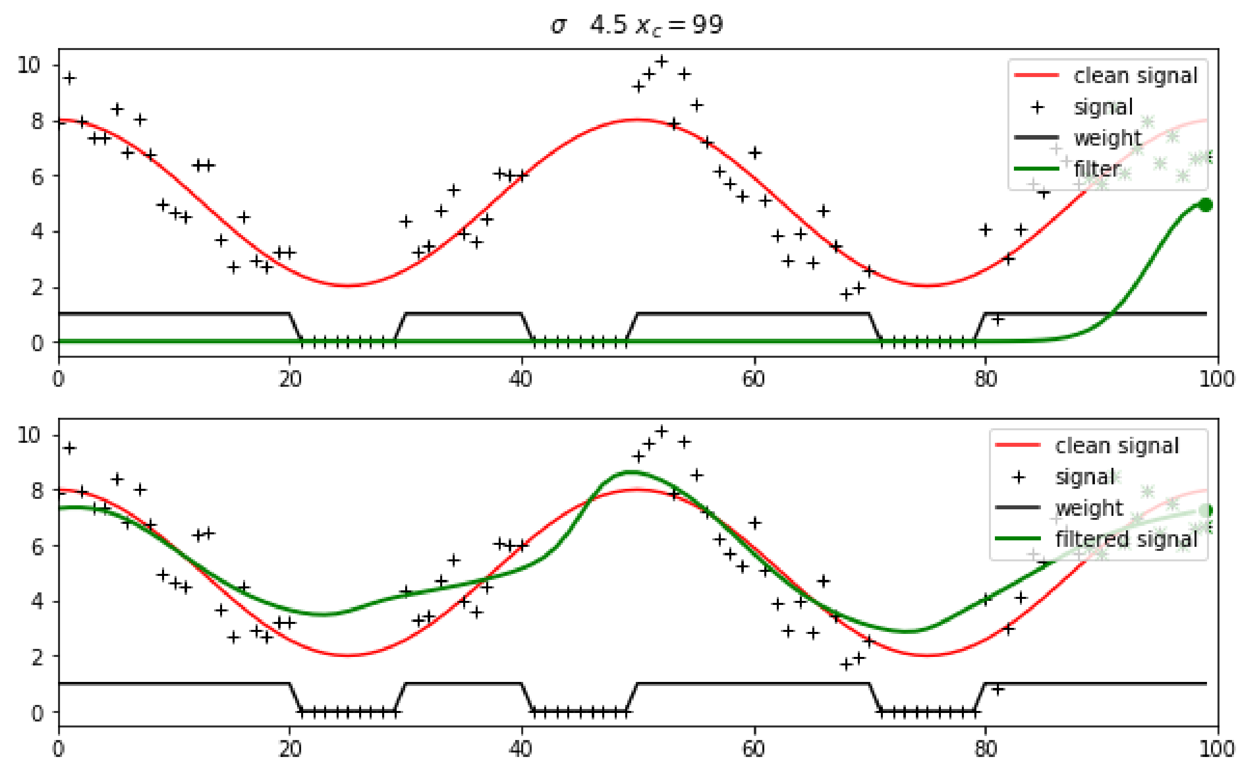


Table of Contents

2.8.15 3.4 Stacking and interpolating data

[up to 3.0]

3.4.1 Introduction

In this section, we will:

- develop code to produce a stacked dataset of spatio-temporal data on a grid
- interpolate over any missing data
- smooth the dataset

3.4.1.1 Test your login

Let's first test your NASA login:

```
z = 37373676
try:
    print(z)
except (NameError):
    print("Variable z is not defined")
```

```
37373676
```

```
import geog0111.nasa_requests as nasa_requests
from geog0111.cylog import cylog

url = 'https://e4ftl01.cr.usgs.gov/MOTA/MCD15A3H.006/2018.09.30/'

# grab the HTML information
try:
    html = nasa_requests.get(url).text
    # test a few lines of the html
    if html[:20] == '<!DOCTYPE HTML PUBLI':
        print('this seems to be ok ... ')
        print(
            'use cylog().login() anywhere you need to specify the tuple (username,
↪password)'
        )
except:
    print('login error ... ')
    print('try entering your username password again')
    print('then re-run this cell until it works')
    print('If its wednesday, ignore this!')
    # uncomment next line to reset password
    #cylog(init=True)
```

```
this seems to be ok ...
use cylog().login() anywhere you need to specify the tuple (username,password)
```

```
# get the MODIS LAI dataset for 2016/2017 for W. Europe
from geog0111.geog_data import procure_dataset
from pathlib import Path

files = list(Path('data').glob('MCD15A3H.A201[6-7]*h1[7-8]v0[3-4].006*hdf'))
if len(files) < 732:
    _ = procure_dataset("lai_files",verbose=False)
```

```
import gdal
import matplotlib.pyplot as plt
%matplotlib inline

g = gdal.Open("data/MCD15A3H.A2016001.h18v03.006.2016007073724.hdf")

sds = g.GetSubDatasets()
for s in sds:
    print(s[1])
```

```
[2400x2400] Fpar_500m MOD_Grid_MCD15A3H (8-bit unsigned integer)
[2400x2400] Lai_500m MOD_Grid_MCD15A3H (8-bit unsigned integer)
[2400x2400] FparLai_QC MOD_Grid_MCD15A3H (8-bit unsigned integer)
[2400x2400] FparExtra_QC MOD_Grid_MCD15A3H (8-bit unsigned integer)
[2400x2400] FparStdDev_500m MOD_Grid_MCD15A3H (8-bit unsigned integer)
[2400x2400] LaiStdDev_500m MOD_Grid_MCD15A3H (8-bit unsigned integer)
```

Now make sure you have the world borders ESRI shape file you need:

```
import requests
import shutil
from pathlib import Path

# zip file
zipfile = 'TM_WORLD_BORDERS-0.3.zip'
# URL
tm_borders_url = f"http://thematicmapping.org/downloads/{zipfile}"
# destibnation folder
destination_folder = Path('data')

# set up some filenames
zip_file = destination_folder.joinpath(zipfile)
shape_file = zip_file.with_name(zipfile.replace('zip', 'shp'))

# download zip if need to
if not Path(zip_file).exists():
    r = requests.get(tm_borders_url)
    with open(zip_file, 'wb') as fp:
        fp.write(r.content)

# extract shp from zip if need to
if not Path(shape_file).exists():
    shutil.unpack_archive(zip_file.as_posix(), extract_dir=destination_folder)
```

3.4.2 QA data

3.4.2.1 Why QA data?

The quality of data varies according to sampling and other factors. For satellite-derived data using optical wavelengths, the two main controls are orbital constraints and cloud cover. We generally define a ‘quaity’ data layer to express this.

To use a satellite-derived dataset, we need to look at ‘quality’ data for dataset (and/or uncertainty).

For example, if interpolating data, we would want to base the weight we put on any sample on the ‘quality’ of that sample. This will be expressed by either some QC (Quality Control) assessment (‘good’, ‘ok’, ‘bad’) or some measure of uncertainty (or both).

Here, we will use the QA information in the LAI product to generate a sample weighting scheme. We shall later use this weighting for data smoothing and interpolation.

First, let's access the LAI and QA datasets.

We can do this by specifying either `Lai_500m` or `FparLai_QC` in the dataset label.

3.4.2.2 Set condition variables

Let's set up the variables we will use, including a pattern to match the `tile` information we need.

Here, we have:

```
tile = 'h1[7-8]v0[3-4]'
```

which we can interpret as:

```
h17v03, h17v04, h18v03, or h18v04
```

The tile definition ius useful for us to use in any output names (so we can identify it from the name). But the string `h1[7-8]v0[3-4]` contains some 'awkward' characters, namely `[`, `]` and `-` that we might prefer not to use in a filename.

So we derive a new descriptor that we call `tile_` which is more 'filename friendly'.

Thye rest of the code below proceeds much the same as code we have previously used.

We build a gdal VRT file from the set of hdf files using `gdal.BuildVRT()`.

Then we crop to the vector defined by the `FIPS` variable in the shapefile (the country code here) using `gdal.Warp()`. We save this as a gdal VRT file, decsribed by the variable `clipped_file`.

When we make gdal calls, we need to force the system to write files to disc. This can be done by closing the (effective) file descriptors, or by deleting the variable `g` in this case. If you don't do that, you can hit file synchronisation problems. You should always close (or delete) file descriptors when you have finished with them.

You should be able to follow what goes on in the code block below. We will re-use these same ideas later, so it is worthwhile understanding the steps we go through now.

```
import gdal
import numpy as np
from pathlib import Path
from geog0111.create_blank_file import create_blank_file
from datetime import datetime

#------------------
# set up the dataset information
destination_folder = Path('data')
year = 2017
product = 'MCD15A3H'
version = 6
tile = 'h1[7-8]v0[3-4]'
doy = 149
params =  ['Lai_500m', 'FparLai_QC']
# Luxembourg
FIPS = "LU"
#------------------

# make a text-friendly version of tile
```

(continues on next page)

(continued from previous page)

```
tile_ = tile.replace('[','_').replace(']','_').replace('-','')+FIPS

# location of the shapefile
shape_file = destination_folder.\
                 joinpath('TM_WORLD_BORDERS-0.3.shp').as_posix()

# define strings for the ip and op files
ipfile = destination_folder.\
                joinpath(f'{product}.A{year}{doy:03d}.{tile_}.{version:03d}').as_
↪posix()

opfile = ipfile.replace(f'{doy:03d}.','').replace(tile,tile_)

print('ipfile',ipfile)
print('opfile',opfile)

# now glob the hdf files matching the pattern
filenames = list(destination_folder\
                .glob(f'{product}.A{year}{doy:03d}.{tile}.{version:03d}.*.hdf'))

# start with an empty list
ofiles = []

# loop over each parameter we need
for d in params:

    gdal_filenames = []
    for file_name in filenames:
        fname = f'HDF4_EOS:EOS_GRID:'+\
                f'"{file_name.as_posix()}":'+\
                f'MOD_Grid_MCD15A3H:{d}'
        gdal_filenames.append(fname)
    dataset_names = sorted(gdal_filenames)

    # mangle the dataset names
    dataset_names = sorted([f'HDF4_EOS:EOS_GRID:'+\
                         f'"{file_name.as_posix()}":'+\
                         f'MOD_Grid_MCD15A3H:{d}'\
                            for file_name in filenames])
    print(dataset_names)

    # derive some filenames for vrt files
    spatial_file = f'{opfile}.{doy:03d}.{d}.vrt'
    clipped_file = f'{opfile}.{doy:03d}_clip.{d}.vrt'

    # build the files
    g = gdal.BuildVRT(spatial_file, dataset_names)
    if(g):
        del(g)
        g = gdal.Warp(clipped_file,\
                                   spatial_file,\
                                   format='VRT', dstNodata=255,\
                                   cutlineDSName=shape_file,\
                                   cutlineWhere=f"FIPS='{FIPS}'",\
                                   cropToCutline=True)
        if (g):
            del(g)
```

(continues on next page)

(continued from previous page)

```
        ofiles.append(clipped_file)
print(ofiles)
```

```
ipfile data/MCD15A3H.A2017149.h1_78_v0_34_LU.006
opfile data/MCD15A3H.A2017h1_78_v0_34_LU.006
['HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2017149.h17v03.006.2017164112436.hdf":MOD_Grid_
↪MCD15A3H:Lai_500m', 'HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2017149.h17v04.006.
↪2017164112432.hdf":MOD_Grid_MCD15A3H:Lai_500m', 'HDF4_EOS:EOS_GRID:"data/MCD15A3H.
↪A2017149.h18v03.006.2017164112435.hdf":MOD_Grid_MCD15A3H:Lai_500m', 'HDF4_EOS:EOS_
↪GRID:"data/MCD15A3H.A2017149.h18v04.006.2017164112441.hdf":MOD_Grid_MCD15A3H:Lai_
↪500m']
['HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2017149.h17v03.006.2017164112436.hdf":MOD_Grid_
↪MCD15A3H:FparLai_QC', 'HDF4_EOS:EOS_GRID:"data/MCD15A3H.A2017149.h17v04.006.
↪2017164112432.hdf":MOD_Grid_MCD15A3H:FparLai_QC', 'HDF4_EOS:EOS_GRID:"data/MCD15A3H.
↪A2017149.h18v03.006.2017164112435.hdf":MOD_Grid_MCD15A3H:FparLai_QC', 'HDF4_EOS:EOS_
↪GRID:"data/MCD15A3H.A2017149.h18v04.006.2017164112441.hdf":MOD_Grid_
↪MCD15A3H:FparLai_QC']
['data/MCD15A3H.A2017h1_78_v0_34_LU.006.149_clip.Lai_500m.vrt', 'data/MCD15A3H.
↪A2017h1_78_v0_34_LU.006.149_clip.FparLai_QC.vrt']
```

```
import gdal
import numpy as np
from pathlib import Path
from geog0111.create_blank_file import create_blank_file
from datetime import datetime, timedelta


def find_mcdfiles(year, doy, tiles, folder):
    data_folder = Path(folder)
    # Find all MCD files
    mcd_files = []
    for tile in tiles:
        sel_files = data_folder.glob(
            f"MCD15*.A{year:d}{doy:03d}.{tile:s}.*hdf")
        for fich in sel_files:
            mcd_files.append(fich)
    return mcd_files


def create_gdal_friendly_names(filenames, layer):

    # Create GDAL friendly-names...
    gdal_filenames = []
    for file_name in filenames:
        fname = f'HDF4_EOS:EOS_GRID:'+\
                    f'"{file_name.as_posix()}":'+\
                    f'MOD_Grid_MCD15A3H:{layer:s}'

        gdal_filenames.append(fname)
    return gdal_filenames


def mosaic_and_clip(tiles,
                    doy,
                    year,
                    folder="data/",
```

(continues on next page)

(continued from previous page)

```
                    layer="Lai_500m",
                    shpfile="data/TM_WORLD_BORDERS-0.3.shp",
                    country_code="LU"):

    folder_path = Path(folder)
    # Find all files to mosaic together
    hdf_files = find_mcdfiles(year, doy, tiles, folder)

    # Create GDAL friendly-names...
    gdal_filenames = create_gdal_friendly_names(hdf_files, layer)

    g = gdal.Warp(
        "",
        gdal_filenames,
        format="MEM",
        dstNodata=255,
        cutlineDSName=shpfile,
        cutlineWhere=f"FIPS='{country_code:s}'",
        cropToCutline=True)
    data = g.ReadAsArray()
    return data


output = np.zeros((176, 123, 85))

today = datetime(2017, 1, 1)
for time in range(85):
    if today.year != 2017:
        break
    doy = int(today.strftime("%j"))
    print(today, doy)
    LU_mosaic = mosaic_and_clip(["h17v03", "h17v04", "h18v03", "h18v04"], doy, 2017)
    output[:, :, time] = LU_mosaic
    today = today + timedelta(days=4)
```

```
2017-01-01 00:00:00 1
2017-01-05 00:00:00 5
2017-01-09 00:00:00 9
2017-01-13 00:00:00 13
2017-01-17 00:00:00 17
2017-01-21 00:00:00 21
2017-01-25 00:00:00 25
2017-01-29 00:00:00 29
2017-02-02 00:00:00 33
2017-02-06 00:00:00 37
2017-02-10 00:00:00 41
2017-02-14 00:00:00 45
2017-02-18 00:00:00 49
2017-02-22 00:00:00 53
2017-02-26 00:00:00 57
2017-03-02 00:00:00 61
2017-03-06 00:00:00 65
2017-03-10 00:00:00 69
2017-03-14 00:00:00 73
2017-03-18 00:00:00 77
2017-03-22 00:00:00 81
```

(continues on next page)

(continued from previous page)

```
2017-03-26 00:00:00 85
2017-03-30 00:00:00 89
2017-04-03 00:00:00 93
2017-04-07 00:00:00 97
2017-04-11 00:00:00 101
2017-04-15 00:00:00 105
2017-04-19 00:00:00 109
2017-04-23 00:00:00 113
2017-04-27 00:00:00 117
2017-05-01 00:00:00 121
2017-05-05 00:00:00 125
2017-05-09 00:00:00 129
2017-05-13 00:00:00 133
2017-05-17 00:00:00 137
2017-05-21 00:00:00 141
2017-05-25 00:00:00 145
2017-05-29 00:00:00 149
2017-06-02 00:00:00 153
2017-06-06 00:00:00 157
2017-06-10 00:00:00 161
2017-06-14 00:00:00 165
2017-06-18 00:00:00 169
2017-06-22 00:00:00 173
2017-06-26 00:00:00 177
2017-06-30 00:00:00 181
2017-07-04 00:00:00 185
2017-07-08 00:00:00 189
2017-07-12 00:00:00 193
2017-07-16 00:00:00 197
2017-07-20 00:00:00 201
2017-07-24 00:00:00 205
2017-07-28 00:00:00 209
2017-08-01 00:00:00 213
2017-08-05 00:00:00 217
2017-08-09 00:00:00 221
2017-08-13 00:00:00 225
2017-08-17 00:00:00 229
2017-08-21 00:00:00 233
2017-08-25 00:00:00 237
2017-08-29 00:00:00 241
2017-09-02 00:00:00 245
2017-09-06 00:00:00 249
2017-09-10 00:00:00 253
2017-09-14 00:00:00 257
2017-09-18 00:00:00 261
2017-09-22 00:00:00 265
2017-09-26 00:00:00 269
2017-09-30 00:00:00 273
2017-10-04 00:00:00 277
2017-10-08 00:00:00 281
2017-10-12 00:00:00 285
2017-10-16 00:00:00 289
2017-10-20 00:00:00 293
2017-10-24 00:00:00 297
2017-10-28 00:00:00 301
2017-11-01 00:00:00 305
2017-11-05 00:00:00 309
```

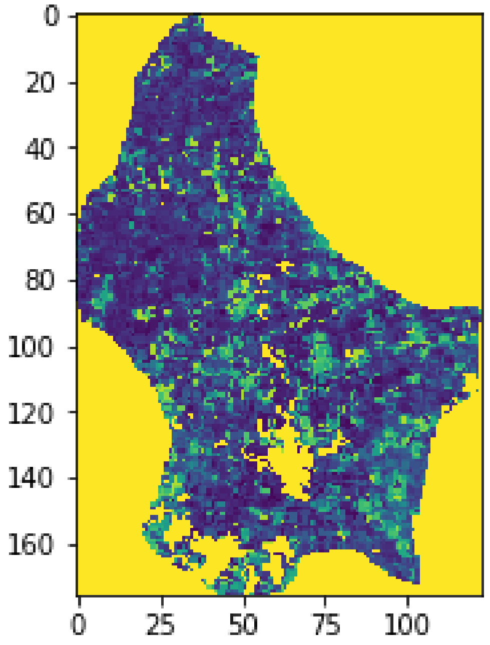
(continues on next page)

(continued from previous page)

```
2017-11-09 00:00:00 313
2017-11-13 00:00:00 317
2017-11-17 00:00:00 321
2017-11-21 00:00:00 325
2017-11-25 00:00:00 329
2017-11-29 00:00:00 333
2017-12-03 00:00:00 337
```

```
plt.imshow(output[:, :, 50], vmin=0, vmax=80)
```

```
<matplotlib.image.AxesImage at 0x120defda0>
```



Exercise 3.4.1

- examine the code block above, and write a function that takes as inputs the variables given in the block enclosed by `#------------------`, the dataset information
- the code should setup the VRT files and return the list of clipped dataset filenames: `['data/MCD15A3H.A2017h1_78_v0_34_LU.006.149_clip.Lai_500m.vrt', 'data/MCD15A3H.A2017h1_78_v0_34_LU.006.149_clip.FparLai_QC.vrt']` here.
- Make sure you test your function: check that it generates the files you expect from the inputs you give.
- try to develop an automated test to see that it has worked (Homework)

Hint

Be clear about what you are doing in your code.

The purpose of this function is to build clipped VRT files for the conditions you set.

The conditions are the parameters driving the function.

The list of files you develop are returned.

```
# do exercise here
```

An attempt at some of this is to try to split the code given cell into a few self-contained and easily testable functions. Broadly speaking, what the code does is

1. Create a list of HDF files that match a pattern (date stamp, as well as belonging to a set of tiles)
2. For each HDF file, select a layer using the GDAL selection method
3. Mosaic all the files for the given date
4. Apply a clipping mask from a vector file

Points 3 and 4 are really direct calls to GDAL functions, and can be combined into one, but we can spin off two simple functions for the first two tasks.

Finding the files requires knowledge of the dates (day of year and year), the location of the files (a folder), as well as the tiles. Rather than use a complex wildcard/regular expression like `h1[7-8]v0[3-4]`, we can just pass a list of tiles and loop over them. We return the filenames. In the available dataset, we have a couple of years of data with four tiles, so we can test this by checking that we get four tiles for different dates.

```python
def find_mcdfiles(year, doy, tiles, folder):
    """Finds MCD15 files in a given folder for a date and set of tiles
    #TODO Missing documentation
    """
    data_folder = Path(folder)
    # Find all MCD files
    mcd_files = []
    for tile in tiles:
        # Loop over all tiles, and search for files that have
        # the tile of interest
        sel_files = data_folder.glob(
            f"MCD15*.A{year:d}{doy:03d}.{tile:s}.*hdf")
        for fich in sel_files:
            mcd_files.append(fich)
    return mcd_files

# Test with two dates
results = find_mcdfiles(2017, 1, ["h17v03", "h17v04", "h18v03", "h18v04"],
                        folder="data/")
for result in results:
    print(result)

results = find_mcdfiles(2017, 45, ["h17v03", "h17v04", "h18v03", "h18v04"],
                        folder="data/")
for result in results:
    print(result)
```

```
data/MCD15A3H.A2017001.h17v03.006.2017014005341.hdf
data/MCD15A3H.A2017001.h17v04.006.2017014005344.hdf
data/MCD15A3H.A2017001.h18v03.006.2017014005401.hdf
data/MCD15A3H.A2017001.h18v04.006.2017014005359.hdf
data/MCD15A3H.A2017045.h17v03.006.2017053101326.hdf
data/MCD15A3H.A2017045.h17v04.006.2017053101338.hdf
data/MCD15A3H.A2017045.h18v03.006.2017053101154.hdf
data/MCD15A3H.A2017045.h18v04.006.2017053101351.hdf
```

The second bit of code broadly speaking just *embellishes* the previous file names by giving them a path to the internal dataset that GDAL can read. For the MCD15A3H product, this means that we build a string with the filename and the layer as

`HDF4_EOS:EOS_GRID:"<the filename>":MOD_Grid_MCD15A3H:<the layer>`

Already, we can see that f-strings will be a very good fit for this! As a testing framework for this function, we ought to be able to open the files with `gdal.Open` and get some numbers. . .

```
def create_gdal_friendly_names(filenames, layer):
    """Given a list of HDF filenames, and a layer, create
    a list of GDAL pointers to an internal layer in the
    filenames given.
    #TODO docstring needs improvements
    """
    # Create GDAL friendly-names...
    gdal_filenames = []
    for file_name in filenames:
        # Convert filename to a string. Could also do it with
        # str(file_name)
        fname = file_name.as_posix()
        # Create the GDAL pointer name
        fname = f'HDF4_EOS:EOS_GRID:"{fname:s}":MOD_Grid_MCD15A3H:{layer:s}'
        gdal_filenames.append(fname)
    return gdal_filenames

# Testing! Get some filenames from find_mcdfiles....
results = find_mcdfiles(2017, 45, ["h17v03", "h17v04", "h18v03", "h18v04"],
                        folder="data/")

gdal_filenames = create_gdal_friendly_names(results, "Lai_500m")
for gname in gdal_filenames:
    print(gdal.Info(gname, stats=True))
    break

# Check another layer...
gdal_filenames = create_gdal_friendly_names(results, "FparLai_QC")
for gname in gdal_filenames:
    print(gdal.Info(gname, stats=True))
    break
```

```
Driver: HDF4Image/HDF4 Dataset
Files: data/MCD15A3H.A2017045.h17v03.006.2017053101326.hdf
Size is 2400, 2400
Coordinate System is:
PROJCS["unnamed",
    GEOGCS["Unknown datum based upon the custom spheroid",
        DATUM["Not specified (based on custom spheroid)",
            SPHEROID["Custom spheroid",6371007.181,0]],
        PRIMEM["Greenwich",0],
        UNIT["degree",0.0174532925199433]],
    PROJECTION["Sinusoidal"],
    PARAMETER["longitude_of_center",0],
    PARAMETER["false_easting",0],
    PARAMETER["false_northing",0],
    UNIT["Meter",1]]
Origin = (-1111950.519667000044137,6671703.117999999783933)
Pixel Size = (463.312716527916677,-463.312716527916507)
Metadata:
  add_offset=0
```

(continues on next page)

(continued from previous page)

```
add_offset_err=0
ALGORITHMPACKAGEACCEPTANCEDATE=10-01-2004
ALGORITHMPACKAGEMATURITYCODE=Normal
ALGORITHMPACKAGENAME=MCDPR_15A3
ALGORITHMPACKAGEVERSION=6
ASSOCIATEDINSTRUMENTSHORTNAME.1=MODIS
ASSOCIATEDINSTRUMENTSHORTNAME.2=MODIS
ASSOCIATEDPLATFORMSHORTNAME.1=Terra
ASSOCIATEDPLATFORMSHORTNAME.2=Aqua
ASSOCIATEDSENSORSHORTNAME.1=MODIS
ASSOCIATEDSENSORSHORTNAME.2=MODIS
AUTOMATICQUALITYFLAG.1=Passed
AUTOMATICQUALITYFLAGEXPLANATION.1=No automatic quality assessment is performed in␣
↪the PGE
calibrated_nt=21
CHARACTERISTICBINANGULARSIZE500M=15.0
CHARACTERISTICBINSIZE500M=463.312716527778
DATACOLUMNS500M=2400
DATAROWS500M=2400
DAYNIGHTFLAG=Day
DESCRREVISION=6.0
EASTBOUNDINGCOORDINATE=0.016666666662491
ENGINEERING_DATA=(none-available)

EXCLUSIONGRINGFLAG.1=N
GEOANYABNORMAL=False
GEOESTMAXRMSERROR=50.0
GLOBALGRIDCOLUMNS500M=86400
GLOBALGRIDROWS500M=43200
GRANULEBEGINNINGDATETIME=2017-02-16T10:25:37.000Z
GRANULEDAYNIGHTFLAG=Day
GRANULEENDINGDATETIME=2017-02-16T10:25:37.000Z
GRINGPOINTLATITUDE.1=49.7394264948349, 59.9999999946118, 60.0089388384779, 49.
↪7424953501575
GRINGPOINTLONGITUDE.1=-15.4860189105775, -19.9999999949462, 0.0325645816155362, 0.
↪0125638874822839
GRINGPOINTSEQUENCENO.1=1, 2, 3, 4
HDFEOSVersion=HDFEOS_V2.17
HORIZONTALTILENUMBER=17
identifier_product_doi=10.5067/MODIS/MCD15A3H.006
identifier_product_doi=10.5067/MODIS/MCD15A3H.006
identifier_product_doi_authority=http://dx.doi.org
identifier_product_doi_authority=http://dx.doi.org
INPUTPOINTER=MYD15A1H.A2017048.h17v03.006.2017050060914.hdf, MYD15A1H.A2017047.
↪h17v03.006.2017049094106.hdf, MYD15A1H.A2017046.h17v03.006.2017048064901.hdf,␣
↪MYD15A1H.A2017045.h17v03.006.2017047060809.hdf, MOD15A1H.A2017048.h17v03.006.
↪2017050103059.hdf, MOD15A1H.A2017047.h17v03.006.2017053100630.hdf, MOD15A1H.
↪A2017046.h17v03.006.2017052201108.hdf, MOD15A1H.A2017045.h17v03.006.2017047102537.
↪hdf, MCD15A3_ANC_RI4.hdf
LOCALGRANULEID=MCD15A3H.A2017045.h17v03.006.2017053101326.hdf
LOCALVERSIONID=5.0.4
LONGNAME=MODIS/Terra+Aqua Leaf Area Index/FPAR 4-Day L4 Global 500m SIN Grid
long_name=MCD15A3H MODIS/Terra Gridded 500M Leaf Area Index LAI (4-day composite)
MAXIMUMOBSERVATIONS500M=1
MOD15A1_ANC_BUILD_CERT=mtAncUtil v. 1.8 Rel. 09.11.2000 17:36 API v. 2.5.6 release␣
↪09.14.2000 16:33 Rev.Index 102 (J.Glassy)
```

(continues on next page)

(continued from previous page)

```
  MOD15A2_FILLVALUE_DOC=MOD15A2 FILL VALUE LEGEND
255 = _Fillvalue, assigned when:
    * the MOD09GA suf. reflectance for channel VIS, NIR was assigned its _Fillvalue,␣
↪or
    * land cover pixel itself was assigned _Fillvalus 255 or 254.
254 = land cover assigned as perennial salt or inland fresh water.
253 = land cover assigned as barren, sparse vegetation (rock, tundra, desert.)
252 = land cover assigned as perennial snow, ice.
251 = land cover assigned as "permanent" wetlands/inundated marshlands.
250 = land cover assigned as urban/built-up.
249 = land cover assigned as "unclassified" or not able to determine.

  MOD15A2_FILLVALUE_DOC=MOD15A2 FILL VALUE LEGEND
255 = _Fillvalue, assigned when:
    * the MOD09GA suf. reflectance for channel VIS, NIR was assigned its _Fillvalue,␣
↪or
    * land cover pixel itself was assigned _Fillvalus 255 or 254.
254 = land cover assigned as perennial salt or inland fresh water.
253 = land cover assigned as barren, sparse vegetation (rock, tundra, desert.)
252 = land cover assigned as perennial snow, ice.
251 = land cover assigned as "permanent" wetlands/inundated marshlands.
250 = land cover assigned as urban/built-up.
249 = land cover assigned as "unclassified" or not able to determine.

  MOD15A2_FparExtra_QC_DOC=
FparExtra_QC 6 BITFIELDS IN 8 BITWORD
LANDSEA PASS-THROUGH START 0 END 1 VALIDS 4
LANDSEA   00 = 0 LAND       AggrQC(3,5)values{001}
LANDSEA   01 = 1 SHORE      AggrQC(3,5)values{000,010,100}
LANDSEA   10 = 2 FRESHWATER AggrQC(3,5)values{011,101}
LANDSEA   11 = 3 OCEAN      AggrQC(3,5)values{110,111}
SNOW_ICE (from Aggregate_QC bits) START 2 END 2 VALIDS 2
SNOW_ICE  0 = No snow/ice detected
SNOW_ICE  1 = Snow/ice were detected
AEROSOL START 3 END 3 VALIDS 2
AEROSOL   0 = No or low atmospheric aerosol levels detected
AEROSOL   1 = Average or high aerosol levels detected
CIRRUS (from Aggregate_QC bits {8,9} ) START 4 END 4 VALIDS 2
CIRRUS    1 = No cirrus detected
CIRRUS    1 = Cirrus was detected
INTERNAL_CLOUD_MASK START 5 END 5 VALIDS 2
INTERNAL_CLOUD_MASK 0 = No clouds
INTERNAL_CLOUD_MASK 1 = Clouds were detected
CLOUD_SHADOW START 6 END 6 VALIDS 2
CLOUD_SHADOW        0 = No cloud shadow detected
CLOUD_SHADOW        1 = Cloud shadow detected
SCF_BIOME_MASK START 7 END 7 VALIDS 2
SCF_BIOME_MASK  0 = Biome outside interval <1,4>
SCF_BIOME_MASK  1 = Biome in interval <1,4>

  MOD15A2_FparLai_QC_DOC=
FparLai_QC 5 BITFIELDS IN 8 BITWORD
MODLAND_QC START 0 END 0 VALIDS 2
MODLAND_QC   0 = Good Quality (main algorithm with or without saturation)
MODLAND_QC   1 = Other Quality (back-up algorithm or fill value)
SENSOR START 1 END 1 VALIDS 2
SENSOR       0  = Terra
```

(continues on next page)

(continued from previous page)

```
SENSOR       1  = Aqua
DEADDETECTOR START 2 END 2 VALIDS 2
DEADDETECTOR 0 = Detectors apparently fine for up to 50% of channels 1,2
DEADDETECTOR 1 = Dead detectors caused >50% adjacent detector retrieval
CLOUDSTATE START 3 END 4 VALIDS 4 (this inherited from Aggregate_QC bits {0,1} cloud␣
↪state)
CLOUDSTATE   00 = 0 Significant clouds NOT present (clear)
CLOUDSTATE   01 = 1 Significant clouds WERE present
CLOUDSTATE   10 = 2 Mixed cloud present on pixel
CLOUDSTATE   11 = 3 Cloud state not defined,assumed clear
SCF_QC START 5 END 7 VALIDS 5
SCF_QC       000=0 Main (RT) algorithm used, best result possible (no saturation)
SCF_QC       001=1 Main (RT) algorithm used, saturation occured. Good, very usable.
SCF_QC       010=2 Main algorithm failed due to bad geometry, empirical algorithm used
SCF_QC       011=3 Main algorithm faild due to problems other than geometry,␣
↪empirical algorithm used
SCF_QC       100=4 Pixel not produced at all, value coudn't be retrieved (possible␣
↪reasons: bad L1B data, unusable MOD09GA data)

  MOD15A2_StdDev_QC_DOC=MOD15A2 STANDARD DEVIATION FILL VALUE LEGEND
255 = _Fillvalue, assigned when:
    * the MOD09GA suf. reflectance for channel VIS, NIR was assigned its _Fillvalue,␣
↪or
    * land cover pixel itself was assigned _Fillvalus 255 or 254.
254 = land cover assigned as perennial salt or inland fresh water.
253 = land cover assigned as barren, sparse vegetation (rock, tundra, desert.)
252 = land cover assigned as perennial snow, ice.
251 = land cover assigned as "permanent" wetlands/inundated marshlands.
250 = land cover assigned as urban/built-up.
249 = land cover assigned as "unclassified" or not able to determine.
248 = no standard deviation available, pixel produced using backup method.

  NADIRDATARESOLUTION500M=500m
  NDAYS_COMPOSITED=8
  NORTHBOUNDINGCOORDINATE=59.9999999946118
  NUMBEROFGRANULES=1
  PARAMETERNAME.1=MCDPR15A3H
  PGEVERSION=6.0.5
  PROCESSINGCENTER=MODAPS
  PROCESSINGENVIRONMENT=Linux minion6010 2.6.32-642.6.2.el6.x86_64 #1 SMP Wed Oct 26␣
↪06:52:09 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
  PRODUCTIONDATETIME=2017-02-22T10:13:27.000Z
  QAPERCENTCLOUDCOVER.1=12
  QAPERCENTEMPIRICALMODEL=30
  QAPERCENTGOODFPAR=70
  QAPERCENTGOODLAI=70
  QAPERCENTGOODQUALITY=70
  QAPERCENTINTERPOLATEDDATA.1=0
  QAPERCENTMAINMETHOD=70
  QAPERCENTMISSINGDATA.1=77
  QAPERCENTOTHERQUALITY=100
  QAPERCENTOUTOFBOUNDSDATA.1=77
  RANGEBEGINNINGDATE=2017-02-14
  RANGEBEGINNINGTIME=00:00:00
  RANGEENDINGDATE=2017-02-17
  RANGEENDINGTIME=23:59:59
  REPROCESSINGACTUAL=reprocessed
```

(continues on next page)

(continued from previous page)

```
  REPROCESSINGPLANNED=further update is anticipated
  scale_factor=0.1
  scale_factor_err=0
  SCIENCEQUALITYFLAG.1=Not Investigated
  SCIENCEQUALITYFLAGEXPLANATION.1=See http://landweb.nascom/nasa.gov/cgi-bin/QA_WWW/
↪qaFlagPage.cgi?sat=aqua the product Science Quality status.
  SHORTNAME=MCD15A3H
  SOUTHBOUNDINGCOORDINATE=49.9999999955098
  SPSOPARAMETERS=5367, 2680
  SYSTEMFILENAME=MYD15A1H.A2017048.h17v03.006.2017050060914.hdf, MYD15A1H.A2017047.
↪h17v03.006.2017049094106.hdf, MYD15A1H.A2017046.h17v03.006.2017048064901.hdf,␣
↪MYD15A1H.A2017045.h17v03.006.2017047060809.hdf, MOD15A1H.A2017048.h17v03.006.
↪2017050103059.hdf, MOD15A1H.A2017047.h17v03.006.2017053100630.hdf, MOD15A1H.
↪A2017046.h17v03.006.2017052201108.hdf, MOD15A1H.A2017045.h17v03.006.2017047102537.
↪hdf
  TileID=51017003
  UM_VERSION=U.MONTANA MODIS PGE34 Vers 5.0.4 Rev 4 Release 10.18.2006 23:59
  units=m^2/m^2
  valid_range=0, 100
  VERSIONID=6
  VERTICALTILENUMBER=03
  WESTBOUNDINGCOORDINATE=-19.9999999949462
  _FillValue=255
Corner Coordinates:
Upper Left  (-1111950.520, 6671703.118) ( 20d 0' 0.00"W, 60d 0' 0.00"N)
Lower Left  (-1111950.520, 5559752.598) ( 15d33'26.06"W, 50d 0' 0.00"N)
Upper Right (       0.000, 6671703.118) (  0d 0' 0.01"E, 60d 0' 0.00"N)
Lower Right (       0.000, 5559752.598) (  0d 0' 0.01"E, 50d 0' 0.00"N)
Center      ( -555975.260, 6115727.858) (  8d43' 2.04"W, 55d 0' 0.00"N)
Band 1 Block=2400x416 Type=Byte, ColorInterp=Gray
  Description = MCD15A3H MODIS/Terra Gridded 500M Leaf Area Index LAI (4-day␣
↪composite)
  Minimum=0.000, Maximum=254.000, Mean=197.581, StdDev=103.876
  NoData Value=255
  Unit Type: m^2/m^2
  Offset: 0,   Scale:0.1
  Metadata:
    STATISTICS_MAXIMUM=254
    STATISTICS_MEAN=197.5806692449
    STATISTICS_MINIMUM=0
    STATISTICS_STDDEV=103.87553463306

Driver: HDF4Image/HDF4 Dataset
Files: data/MCD15A3H.A2017045.h17v03.006.2017053101326.hdf
       data/MCD15A3H.A2017045.h17v03.006.2017053101326.hdf.aux.xml
Size is 2400, 2400
Coordinate System is:
PROJCS["unnamed",
    GEOGCS["Unknown datum based upon the custom spheroid",
        DATUM["Not specified (based on custom spheroid)",
            SPHEROID["Custom spheroid",6371007.181,0]],
        PRIMEM["Greenwich",0],
        UNIT["degree",0.0174532925199433]],
    PROJECTION["Sinusoidal"],
    PARAMETER["longitude_of_center",0],
    PARAMETER["false_easting",0],
    PARAMETER["false_northing",0],
```

(continues on next page)

(continued from previous page)

```
    UNIT["Meter",1]]
Origin = (-1111950.519667000044137,6671703.117999999783933)
Pixel Size = (463.312716527916677,-463.312716527916507)
Metadata:
  ALGORITHMPACKAGEACCEPTANCEDATE=10-01-2004
  ALGORITHMPACKAGEMATURITYCODE=Normal
  ALGORITHMPACKAGENAME=MCDPR_15A3
  ALGORITHMPACKAGEVERSION=6
  ASSOCIATEDINSTRUMENTSHORTNAME.1=MODIS
  ASSOCIATEDINSTRUMENTSHORTNAME.2=MODIS
  ASSOCIATEDPLATFORMSHORTNAME.1=Terra
  ASSOCIATEDPLATFORMSHORTNAME.2=Aqua
  ASSOCIATEDSENSORSHORTNAME.1=MODIS
  ASSOCIATEDSENSORSHORTNAME.2=MODIS
  AUTOMATICQUALITYFLAG.1=Passed
  AUTOMATICQUALITYFLAGEXPLANATION.1=No automatic quality assessment is performed in␣
↪the PGE
  CHARACTERISTICBINANGULARSIZE500M=15.0
  CHARACTERISTICBINSIZE500M=463.312716527778
  DATACOLUMNS500M=2400
  DATAROWS500M=2400
  DAYNIGHTFLAG=Day
  DESCRREVISION=6.0
  EASTBOUNDINGCOORDINATE=0.016666666662491
  ENGINEERING_DATA=(none-available)

  EXCLUSIONGRINGFLAG.1=N
  FparLai_QC_DOC=
FparLai_QC 5 BITFIELDS IN 8 BITWORD
MODLAND_QC START 0 END 0 VALIDS 2
MODLAND_QC   0 = Good Quality (main algorithm with or without saturation)
MODLAND_QC   1 = Other Quality (back-up algorithm or fill value)
SENSOR START 1 END 1 VALIDS 2
SENSOR       0  = Terra
SENSOR       1  = Aqua
DEADDETECTOR START 2 END 2 VALIDS 2
DEADDETECTOR 0 = Detectors apparently fine for up to 50% of channels 1,2
DEADDETECTOR 1 = Dead detectors caused >50% adjacent detector retrieval
CLOUDSTATE START 3 END 4 VALIDS 4 (this inherited from Aggregate_QC bits {0,1} cloud␣
↪state)
CLOUDSTATE   00 = 0 Significant clouds NOT present (clear)
CLOUDSTATE   01 = 1 Significant clouds WERE present
CLOUDSTATE   10 = 2 Mixed cloud present on pixel
CLOUDSTATE   11 = 3 Cloud state not defined,assumed clear
SCF_QC START 5 END 7 VALIDS 5
SCF_QC       000=0 Main (RT) algorithm used, best result possible (no saturation)
SCF_QC       001=1 Main (RT) algorithm used, saturation occured. Good, very usable.
SCF_QC       010=2 Main algorithm failed due to bad geometry, empirical algorithm used
SCF_QC       011=3 Main algorithm faild due to problems other than geometry,␣
↪empirical algorithm used
SCF_QC       100=4 Pixel not produced at all, value coudn't be retrieved (possible␣
↪reasons: bad L1B data, unusable MOD09GA data)

  GEOANYABNORMAL=False
  GEOESTMAXRMSERROR=50.0
  GLOBALGRIDCOLUMNS500M=86400
  GLOBALGRIDROWS500M=43200
```

(continues on next page)

(continued from previous page)

```
  GRANULEBEGINNINGDATETIME=2017-02-16T10:25:37.000Z
  GRANULEDAYNIGHTFLAG=Day
  GRANULEENDINGDATETIME=2017-02-16T10:25:37.000Z
  GRINGPOINTLATITUDE.1=49.7394264948349, 59.9999999946118, 60.0089388384779, 49.
↪7424953501575
  GRINGPOINTLONGITUDE.1=-15.4860189105775, -19.9999999949462, 0.0325645816155362, 0.
↪0125638874822839
  GRINGPOINTSEQUENCENO.1=1, 2, 3, 4
  HDFEOSVersion=HDFEOS_V2.17
  HORIZONTALTILENUMBER=17
  identifier_product_doi=10.5067/MODIS/MCD15A3H.006
  identifier_product_doi=10.5067/MODIS/MCD15A3H.006
  identifier_product_doi_authority=http://dx.doi.org
  identifier_product_doi_authority=http://dx.doi.org
  INPUTPOINTER=MYD15A1H.A2017048.h17v03.006.2017050060914.hdf, MYD15A1H.A2017047.
↪h17v03.006.2017049094106.hdf, MYD15A1H.A2017046.h17v03.006.2017048064901.hdf,␣
↪MYD15A1H.A2017045.h17v03.006.2017047060809.hdf, MOD15A1H.A2017048.h17v03.006.
↪2017050103059.hdf, MOD15A1H.A2017047.h17v03.006.2017053100630.hdf, MOD15A1H.
↪A2017046.h17v03.006.2017052201108.hdf, MOD15A1H.A2017045.h17v03.006.2017047102537.
↪hdf, MCD15A3_ANC_RI4.hdf
  LOCALGRANULEID=MCD15A3H.A2017045.h17v03.006.2017053101326.hdf
  LOCALVERSIONID=5.0.4
  LONGNAME=MODIS/Terra+Aqua Leaf Area Index/FPAR 4-Day L4 Global 500m SIN Grid
  long_name=MCD15A3H MODIS/Terra+Aqua QC for FPAR and LAI (4-day composite)
  MAXIMUMOBSERVATIONS500M=1
  MOD15A1_ANC_BUILD_CERT=mtAncUtil v. 1.8 Rel. 09.11.2000 17:36 API v. 2.5.6 release␣
↪09.14.2000 16:33 Rev.Index 102 (J.Glassy)

  MOD15A2_FILLVALUE_DOC=MOD15A2 FILL VALUE LEGEND
255 = _Fillvalue, assigned when:
    * the MOD09GA suf. reflectance for channel VIS, NIR was assigned its _Fillvalue,␣
↪or
    * land cover pixel itself was assigned _Fillvalus 255 or 254.
254 = land cover assigned as perennial salt or inland fresh water.
253 = land cover assigned as barren, sparse vegetation (rock, tundra, desert.)
252 = land cover assigned as perennial snow, ice.
251 = land cover assigned as "permanent" wetlands/inundated marshlands.
250 = land cover assigned as urban/built-up.
249 = land cover assigned as "unclassified" or not able to determine.

  MOD15A2_FparExtra_QC_DOC=
FparExtra_QC 6 BITFIELDS IN 8 BITWORD
LANDSEA PASS-THROUGH START 0 END 1 VALIDS 4
LANDSEA   00 = 0 LAND       AggrQC(3,5)values{001}
LANDSEA   01 = 1 SHORE      AggrQC(3,5)values{000,010,100}
LANDSEA   10 = 2 FRESHWATER AggrQC(3,5)values{011,101}
LANDSEA   11 = 3 OCEAN      AggrQC(3,5)values{110,111}
SNOW_ICE (from Aggregate_QC bits) START 2 END 2 VALIDS 2
SNOW_ICE  0 = No snow/ice detected
SNOW_ICE  1 = Snow/ice were detected
AEROSOL START 3 END 3 VALIDS 2
AEROSOL   0 = No or low atmospheric aerosol levels detected
AEROSOL   1 = Average or high aerosol levels detected
CIRRUS (from Aggregate_QC bits {8,9} ) START 4 END 4 VALIDS 2
CIRRUS    0 = No cirrus detected
CIRRUS    1 = Cirrus was detected
INTERNAL_CLOUD_MASK START 5 END 5 VALIDS 2
```

(continues on next page)

(continued from previous page)

```
INTERNAL_CLOUD_MASK 0 = No clouds
INTERNAL_CLOUD_MASK 1 = Clouds were detected
CLOUD_SHADOW START 6 END 6 VALIDS 2
CLOUD_SHADOW        0 = No cloud shadow detected
CLOUD_SHADOW        1 = Cloud shadow detected
SCF_BIOME_MASK START 7 END 7 VALIDS 2
SCF_BIOME_MASK  0 = Biome outside interval <1,4>
SCF_BIOME_MASK  1 = Biome in interval <1,4>

  MOD15A2_FparLai_QC_DOC=
FparLai_QC 5 BITFIELDS IN 8 BITWORD
MODLAND_QC START 0 END 0 VALIDS 2
MODLAND_QC   0 = Good Quality (main algorithm with or without saturation)
MODLAND_QC   1 = Other Quality (back-up algorithm or fill value)
SENSOR START 1 END 1 VALIDS 2
SENSOR       0  = Terra
SENSOR       1  = Aqua
DEADDETECTOR START 2 END 2 VALIDS 2
DEADDETECTOR 0 = Detectors apparently fine for up to 50% of channels 1,2
DEADDETECTOR 1 = Dead detectors caused >50% adjacent detector retrieval
CLOUDSTATE START 3 END 4 VALIDS 4 (this inherited from Aggregate_QC bits {0,1} cloud␣
↪state)
CLOUDSTATE   00 = 0 Significant clouds NOT present (clear)
CLOUDSTATE   01 = 1 Significant clouds WERE present
CLOUDSTATE   10 = 2 Mixed cloud present on pixel
CLOUDSTATE   11 = 3 Cloud state not defined,assumed clear
SCF_QC START 5 END 7 VALIDS 5
SCF_QC       000=0 Main (RT) algorithm used, best result possible (no saturation)
SCF_QC       001=1 Main (RT) algorithm used, saturation occured. Good, very usable.
SCF_QC       010=2 Main algorithm failed due to bad geometry, empirical algorithm used
SCF_QC       011=3 Main algorithm faild due to problems other than geometry,␣
↪empirical algorithm used
SCF_QC       100=4 Pixel not produced at all, value coudn't be retrieved (possible␣
↪reasons: bad L1B data, unusable MOD09GA data)

  MOD15A2_StdDev_QC_DOC=MOD15A2 STANDARD DEVIATION FILL VALUE LEGEND
255 = _Fillvalue, assigned when:
    * the MOD09GA suf. reflectance for channel VIS, NIR was assigned its _Fillvalue,␣
↪or
    * land cover pixel itself was assigned _Fillvalus 255 or 254.
254 = land cover assigned as perennial salt or inland fresh water.
253 = land cover assigned as barren, sparse vegetation (rock, tundra, desert.)
252 = land cover assigned as perennial snow, ice.
251 = land cover assigned as "permanent" wetlands/inundated marshlands.
250 = land cover assigned as urban/built-up.
249 = land cover assigned as "unclassified" or not able to determine.
248 = no standard deviation available, pixel produced using backup method.

  NADIRDATARESOLUTION500M=500m
  NDAYS_COMPOSITED=8
  NORTHBOUNDINGCOORDINATE=59.9999999946118
  NUMBEROFGRANULES=1
  PARAMETERNAME.1=MCDPR15A3H
  PGEVERSION=6.0.5
  PROCESSINGCENTER=MODAPS
  PROCESSINGENVIRONMENT=Linux minion6010 2.6.32-642.6.2.el6.x86_64 #1 SMP Wed Oct 26␣
↪06:52:09 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
```

(continues on next page)

(continued from previous page)

```
  PRODUCTIONDATETIME=2017-02-22T10:13:27.000Z
  QAPERCENTCLOUDCOVER.1=12
  QAPERCENTEMPIRICALMODEL=30
  QAPERCENTGOODFPAR=70
  QAPERCENTGOODLAI=70
  QAPERCENTGOODQUALITY=70
  QAPERCENTINTERPOLATEDDATA.1=0
  QAPERCENTMAINMETHOD=70
  QAPERCENTMISSINGDATA.1=77
  QAPERCENTOTHERQUALITY=100
  QAPERCENTOUTOFBOUNDSDATA.1=77
  RANGEBEGINNINGDATE=2017-02-14
  RANGEBEGINNINGTIME=00:00:00
  RANGEENDINGDATE=2017-02-17
  RANGEENDINGTIME=23:59:59
  REPROCESSINGACTUAL=reprocessed
  REPROCESSINGPLANNED=further update is anticipated
  SCIENCEQUALITYFLAG.1=Not Investigated
  SCIENCEQUALITYFLAGEXPLANATION.1=See http://landweb.nascom/nasa.gov/cgi-bin/QA_WWW/
↪qaFlagPage.cgi?sat=aqua the product Science Quality status.
  SHORTNAME=MCD15A3H
  SOUTHBOUNDINGCOORDINATE=49.9999999955098
  SPSOPARAMETERS=5367, 2680
  SYSTEMFILENAME=MYD15A1H.A2017048.h17v03.006.2017050060914.hdf, MYD15A1H.A2017047.
↪h17v03.006.2017049094106.hdf, MYD15A1H.A2017046.h17v03.006.2017048064901.hdf,␣
↪MYD15A1H.A2017045.h17v03.006.2017047060809.hdf, MOD15A1H.A2017048.h17v03.006.
↪2017050103059.hdf, MOD15A1H.A2017047.h17v03.006.2017053100630.hdf, MOD15A1H.
↪A2017046.h17v03.006.2017052201108.hdf, MOD15A1H.A2017045.h17v03.006.2017047102537.
↪hdf
  TileID=51017003
  UM_VERSION=U.MONTANA MODIS PGE34 Vers 5.0.4 Rev 4 Release 10.18.2006 23:59
  units=class-flag
  valid_range=0, 254
  VERSIONID=6
  VERTICALTILENUMBER=03
  WESTBOUNDINGCOORDINATE=-19.9999999949462
  _FillValue=255
Corner Coordinates:
Upper Left  (-1111950.520, 6671703.118) ( 20d 0' 0.00"W, 60d 0' 0.00"N)
Lower Left  (-1111950.520, 5559752.598) ( 15d33'26.06"W, 50d 0' 0.00"N)
Upper Right (       0.000, 6671703.118) (  0d 0' 0.01"E, 60d 0' 0.00"N)
Lower Right (       0.000, 5559752.598) (  0d 0' 0.01"E, 50d 0' 0.00"N)
Center      ( -555975.260, 6115727.858) (  8d43' 2.04"W, 55d 0' 0.00"N)
Band 1 Block=2400x416 Type=Byte, ColorInterp=Gray
  Description = MCD15A3H MODIS/Terra+Aqua QC for FPAR and LAI (4-day composite)
  Minimum=0.000, Maximum=157.000, Mean=128.764, StdDev=55.617
  NoData Value=255
  Unit Type: class-flag
  Metadata:
    STATISTICS_MAXIMUM=157
    STATISTICS_MEAN=128.76351493056
    STATISTICS_MINIMUM=0
    STATISTICS_STDDEV=55.617435975907
```

Finally, we need a function that uses the previous two functions, and does the mosaicking. Here, we can consider the options for the output. In some cases, we might just want to extract the data to a numpy array. In some other cases, we might want to store the data as a file that can be shared with others. Using a bunch of VRT files as presented above creates some issues: the original HDF files need to be present, and creating a cascade of VRT files requires that all the intermediate VRT files are present. This can quickly result in unmanageable file numbers (100s-1000s). So we can think that rather than provide a VRT (and all the dependencies, we can provide a GeoTIFF file with the mosaicked and clipped data. This is a portable format that we can pass on to others. Additionally, we can choose to return a numpy array back to the caller so it can carry on processing.

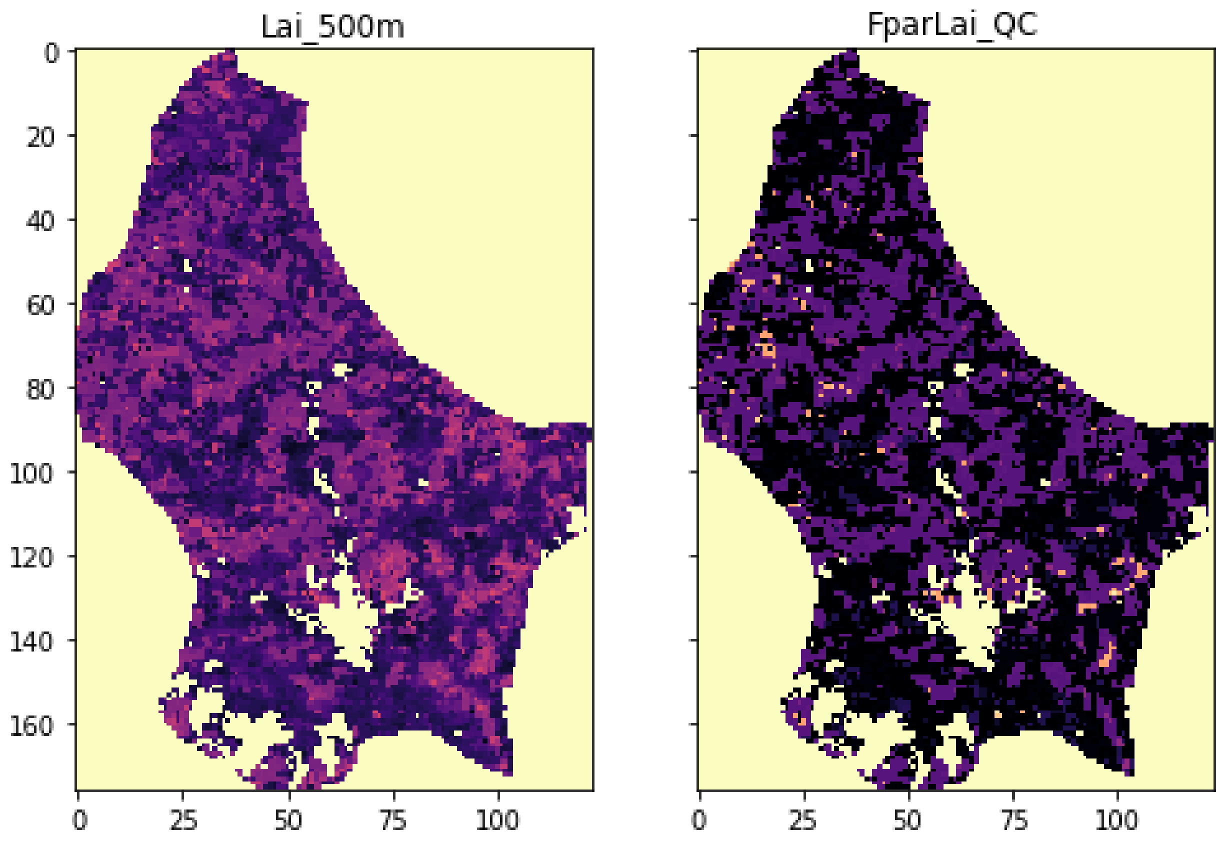
```
def mosaic_and_clip(tiles,
                    doy,
                    year,
                    folder="data/",
                    layer="Lai_500m",
                    shpfile="data/TM_WORLD_BORDERS-0.3.shp",
                    country_code="LU",
                    frmat="MEM"):
    """
    #TODO docstring missing!!!!
    """

    folder_path = Path(folder)
    # Find all files to mosaic together
    hdf_files = find_mcdfiles(year, doy, tiles, folder)

    # Create GDAL friendly-names...
    gdal_filenames = create_gdal_friendly_names(hdf_files, layer)

    if frmat == "MEM":
        g = gdal.Warp(
            "",
            gdal_filenames,
            format="MEM",
            dstNodata=255,
            cutlineDSName=shpfile,
            cutlineWhere=f"FIPS='{country_code:s}'",
            cropToCutline=True)
        data = g.ReadAsArray()
        return data
    elif frmat == "GTiff":
        geotiff_fnamex = f"{layer:s}_{year:d}_{doy:03d}_{country_code:s}.tif"
        geotiff_fname  = folder_path/geotiff_fnamex
        g = gdal.Warp(
            geotiff_fname.as_posix(),
            gdal_filenames,
            format=frmat,
            dstNodata=255,
            cutlineDSName=shpfile,
            cutlineWhere=f"FIPS='{country_code:s}'",
            cropToCutline=True)
        return geotiff_fname.as_posix()
    else:
        raise ValueError("Only MEM or GTiff formats supported!")
```

```
# Testing numpy return arrays
tiles = ["h17v03", "h17v04", "h18v03", "h18v04"]
fig, axs = plt.subplots(nrows=1, ncols=2, sharex=True, sharey=True,
                        figsize=(8, 18))

for i, the_layer in enumerate(["Lai_500m", "FparLai_QC"]):
```

(continues on next page)

(continued from previous page)

```
    data =  mosaic_and_clip(tiles,
                    149,
                    2017,
                    folder="data/",
                    layer=the_layer,
                    shpfile="data/TM_WORLD_BORDERS-0.3.shp",
                    country_code="LU",
                    frmat="MEM")
    axs[i].imshow(data, interpolation="nearest", vmin=0, vmax=120,
                 cmap=plt.cm.magma)
    axs[i].set_title(the_layer)
```



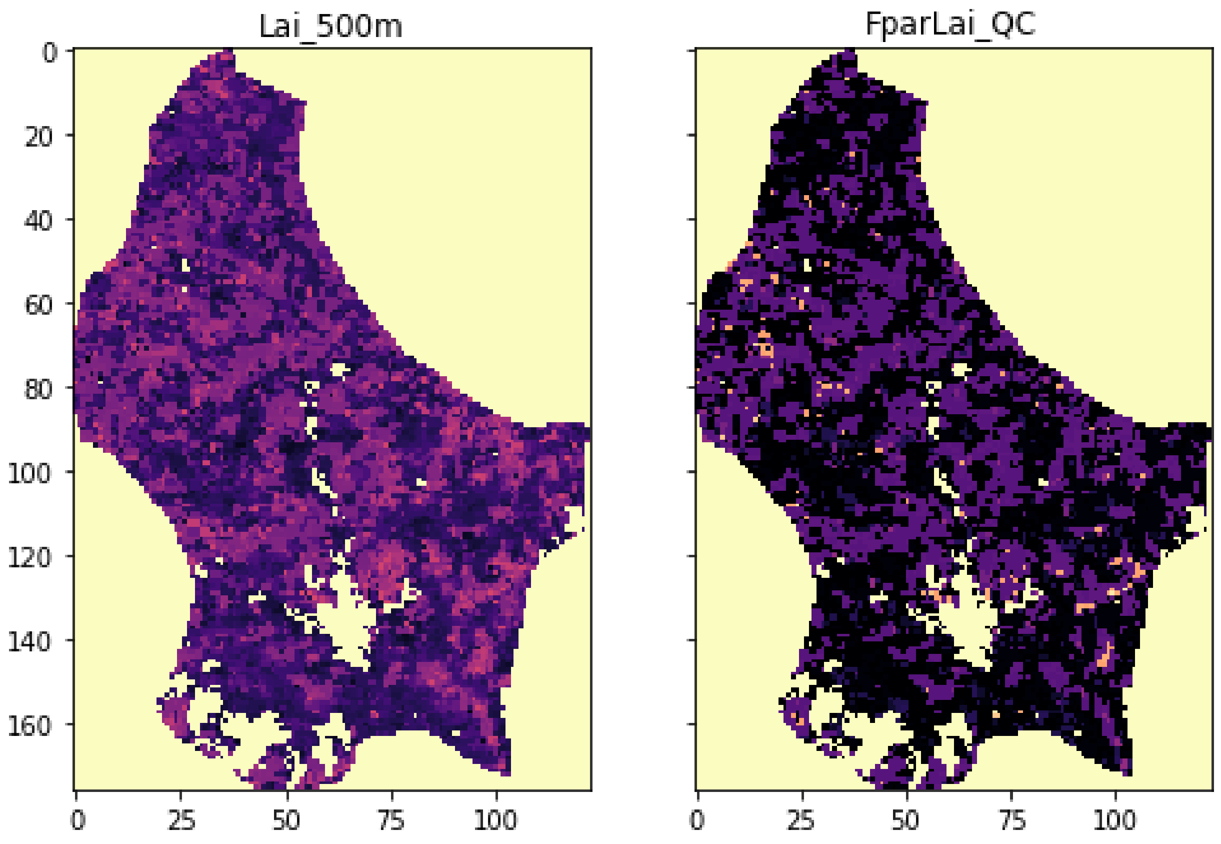
```
# Testing GeoTIFF return arrays
tiles = ["h17v03", "h17v04", "h18v03", "h18v04"]
fig, axs = plt.subplots(nrows=1, ncols=2, sharex=True, sharey=True,
                       figsize=(8, 18))

for i, the_layer in enumerate(["Lai_500m", "FparLai_QC"]):
    fname =  mosaic_and_clip(tiles,
                    149,
                    2017,
                    folder="data/",
                    layer=the_layer,
                    shpfile="data/TM_WORLD_BORDERS-0.3.shp",
                    country_code="LU",
                    frmat="GTiff")
```

(continues on next page)

(continued from previous page)

```
    print(fname)
    g = gdal.Open(fname)
    data = g.ReadAsArray()
    axs[i].imshow(data, interpolation="nearest", vmin=0, vmax=120,
                  cmap=plt.cm.magma)
    axs[i].set_title(the_layer)
```

```
data/Lai_500m_2017_149_LU.tif
data/FparLai_QC_2017_149_LU.tif
```



So this appears to have worked… **visually**. A more thorough analysis would be required, possibly opening the files and checking that you can read out individual locations.

We now have an easy to call function that does a lot of complex processing behind the scenes to provide us with the **actual** data that we might want to use.

3.4.2.3 interpreting QA

We can now get the data describing LAI and QC for any given date (and country, given the limitations of the data in the system!)

The LAI dataset is decribed on the NASA page, with the bit field information given in the file spec

```
  BITFIELDS
-------------
```

(continues on next page)

(continued from previous page)

```
0,0  MODLAND_QC bits
     '0' =  Good Quality (main algorithm with or without saturation)
     '1' =  Other Quality (back-up algorithm or fill values)

1,1 SENSOR
     '0' = Terra
     '1' = Aqua

2,2  DEADDETECTOR
     '0' = Detectors apparently fine for up to 50% of channels 1,2
     '1' = Dead detectors caused >50% adjacent detector retrieval

3,4  CLOUDSTATE (this inherited from Aggregate_QC bits {0,1} cloud state)
     '00' = 0 Significant clouds NOT present (clear)
     '01' = 1 Significant clouds WERE present
     '10' = 2 Mixed cloud present on pixel
     '11' = 3 Cloud state not defined,assumed clear

5,7  SCF_QC (3-bit, (range '000'..100') 5 level Confidence Quality score.
     '000' = 0, Main (RT) method used, best result possible (no saturation)
     '001' = 1, Main (RT) method used with saturation. Good,very usable
     '010' = 2, Main (RT) method failed due to bad geometry, empirical algorithm used
     '011' = 3, Main (RT) method failed due to problems other than geometry,
                          empirical algorithm used
     '100' = 4, Pixel not produced at all, value coudn't be retrieved
                (possible reasons: bad L1B data, unusable MOD09GA data)
```

```
qa_data =  mosaic_and_clip(tiles,
                  149,
                  2017,
                  layer="FparLai_QC")
print(f'Unique QA values found')
print(sorted(np.unique(qa_data)))
```

```
Unique QA values found
[0, 2, 8, 10, 16, 18, 32, 34, 40, 42, 48, 50, 97, 99, 105, 107, 113, 115, 157, 255]
```

We will use the bitfield `SFC_QC` as our main way to interpret quality.

The information above tells us we need to extract bits 5-7 from the QA dataset.

Let's be clear what we mean by this.

The dataset `FparLai_QC` is of data type `uint8`, unsigned 8-bit integer (i.e. unsigned byte).

so, for example, in the file above, we see the following 19 unique codes:

```
import pandas as pd
# some pretty printing code using pandas
qas = np.array([[format(q,'3d'),format(q,'08b')] \
                for q in sorted(np.unique(qa_data))]).T
pd.DataFrame({'decimal representation': qas[0],
              'binary representation': qas[1]})
```

Recall the truth table for the `and` operation:

A	B	A and B
T	T	T
T	F	F
F	T	F
F	F	F

For binary rerpresentation, we replace `True` by `1` and `False` by `0`.

We also use the bitwise `and` operator `&`:

A	B	A & B
1	1	1
1	0	0
0	1	0
0	0	0

Notice that `A & B` only lets the value of `B` come through if `A` is `1`. Setting `A` as `0` effectively 'switches off' the information in `B`.

We can see this more clearly using a combination of bits:

```
A = 0b01100
B = 0b11010

print('A         =',format(A,'#07b'))
print('B         =',format(B,'#07b'))
print('C = A & B =',format(A & B,'#07b'))
```

```
A         = 0b01100
B         = 0b11010
C = A & B = 0b01000
```

Here, the 'mask' `A` is set to `1` for bits 2 and 3 : `0b01100`.

So only the information in bits 2 and 3 of `B` is passed through to `C`. If we change the other bits in `B`, it has no effect on `C`:

```
A = 0b01100
B = 0b01001

print('A         =',format(A,'#07b'))
print('B         =',format(B,'#07b'))
print('C = A & B =',format(A & B,'#07b'))
```

```
A         = 0b01100
B         = 0b01001
C = A & B = 0b01000
```

Exercise 3.4.2

- copy the code from the block above
- change the bit values in `B` and check that only the information in the bits set as `1` in `A` is passed through to `C`
- make a new mask `A` and re-confirm your findings for this mask

Hint

We are using A as a mask, so we set the 'pass' bits in A to 1 and 'block' to 0

```
# do exercise here
```

So, to extract bits 5-7 (the 3 left-most bits) from an 8-bit number, we first perform a bitwise (binary) 'and' operation with the mask 0b11100000. This has 5 0s to the right.

Because we have trailing zeros (to the right) of the masked value, we perform a bit shift operation (>>) of length 5:

```
import pandas as pd
# some pretty printing code using pandas
mask = 0b11100000

unique_qa_data = sorted(np.unique(qa_data))
decimal = np.zeros_like(unique_qa_data, dtype=object)
binary = np.zeros_like(unique_qa_data, dtype=object)
masked = np.zeros_like(unique_qa_data, dtype=object)
shifted = np.zeros_like(unique_qa_data, dtype=object)
sfc_qc = np.zeros_like(unique_qa_data, dtype=object)
for i, q in enumerate(unique_qa_data):
    decimal[i] = format(q, '3d')
    binary[i] = format(q, '08b')
    masked[i] = format ( (q & mask), '08b')
    shifted[i] = format((q & mask)>>5, '08b')
    sfc_qc[i] = format((q & mask)>>5, '03d')



pd.DataFrame({'decimal': decimal, 'binary': binary,
              'masked': masked,'shifted': shifted,
             'SFC_QC': sfc_qc})
```

Looking back at the data interpretation table:

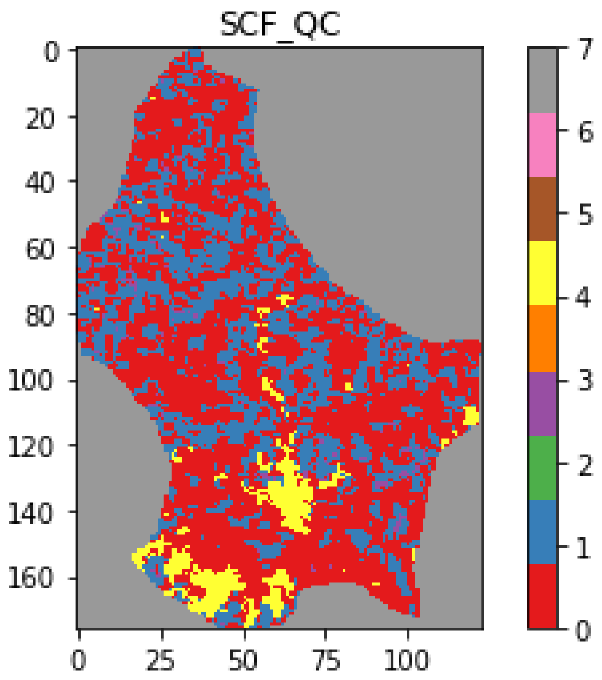
```
5,7  SCF_QC (3-bit, (range '000'..100') 5 level Confidence Quality score.
     '000' = 0, Main (RT) method used, best result possible (no saturation)
     '001' = 1, Main (RT) method used with saturation. Good,very usable
     '010' = 2, Main (RT) method failed due to bad geometry, empirical algorithm used
     '011' = 3, Main (RT) method failed due to problems other than geometry,
                empirical algorithm used
     '100' = 4, Pixel not produced at all, value coudn't be retrieved
                (possible reasons: bad L1B data, unusable MOD09GA data)
```

we can see that QA values of 0,2,,8,10,16 and 18 all correspond to `Main (RT) method used, best result possible (no saturation)`, values of 32,34, 40, 42,48, 50 to `Main (RT) method used with saturation. Good,very usable` etc.

We can apply this operation to the whole QA numpy array. While can use the standard & and >> symbols, for array operations, the functions `np.bitwise_and` and `np.right_shift` are preferred:

```
sfc_qa = np.right_shift(np.bitwise_and(qa_data, mask), 5)
plt.title('SCF_QC')
plt.imshow(sfc_qa,cmap=plt.cm.Set1)
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x12238d898>
```



Exercise 3.4.3

- For the momnent, assume that we only want data which have `SCF_QC` set to `0` (i.e. `Main (RT) method used, best result possible (no saturation)`).
- read in the LAI data associated with this dataset, and set unwanted LAI data (i.e. those with `SCF_QC` not set to `0`) to Not a Number (`np.nan`)
- plot the resultant dataset

Hint

Use the previous function to obtain the corresponding LAI dataset for the current date.

The LAI image plotted should show 'white' (no value) everywhere the `SCF_QC` image above is not red.

```
# do exercise here
```

3.4.2.4 Deriving a weight from QA

What we want to develop here is a translation from the QA information (given in `FparLai_QC`) to a *weight* that we can apply to the data.

If the quality is poor, we want a low weight. If the quality is good, a high weight. It makes sense to call the highest weight 1.0.

For LAI, we can use the QA information contained in bits 5-7 of `FparLai_QC` to achieve this.

The valid codes for `SCF_QC` here are:

```
0 : Main (RT) method used best result possible (no saturation)
1 : Main (RT) method used with saturation. Good very usable
2 : Main (RT) method failed due to bad geometry empirical algorithm used
3 : Main (RT) method failed due to problems other than geometry empirical algorithm␣
↪used
```

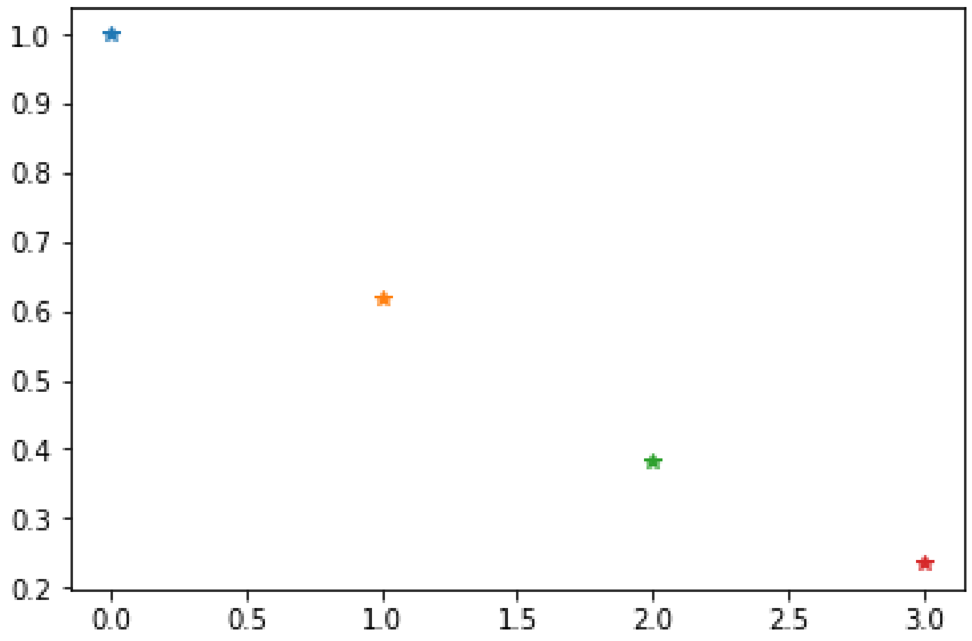
where we have translated the binary representations above to decimal.

A useful way of this to some weight is to define a real number n, where $0 <= n < 1$ and raise this to the power of SCF_QC.

So, for example is we set n = 0.61803398875 (the inverse golden ratio):

```
n = 0.61803398875

for SCF_QC in [0,1,2,3]:
    weight = n**SCF_QC
    print(f'SCF_QC: {SCF_QC} => weight {weight:.4f}')
    plt.plot(SCF_QC, weight, '*')
```

```
SCF_QC: 0 => weight 1.0000
SCF_QC: 1 => weight 0.6180
SCF_QC: 2 => weight 0.3820
SCF_QC: 3 => weight 0.2361
```



Then we have the following meaning for the weights:

```
1.0000 : Main (RT) method used best result possible (no saturation)
0.6180 : Main (RT) method used with saturation. Good very usable
0.3820 : Main (RT) method failed due to bad geometry empirical algorithm used
0.2361 : Main (RT) method failed due to problems other than geometry empirical␣
↪algorithm used
```

Altghough we could vary the value of n used and get subtle variations, this sort of weighting should produce the desired result.

Exercise 3.4.4

- write a function that converts from SCF_QC value to weight.
- apply this to the SCF_QC dataset we generated above.
- display the weight image, along side the SCF_QC and visually check you have the weighting correct

Hint

Since only `[0,1,2,3]` are valid inputs, you could use conditions such as:

```
(SCF_QC == i) * (n ** i)
```

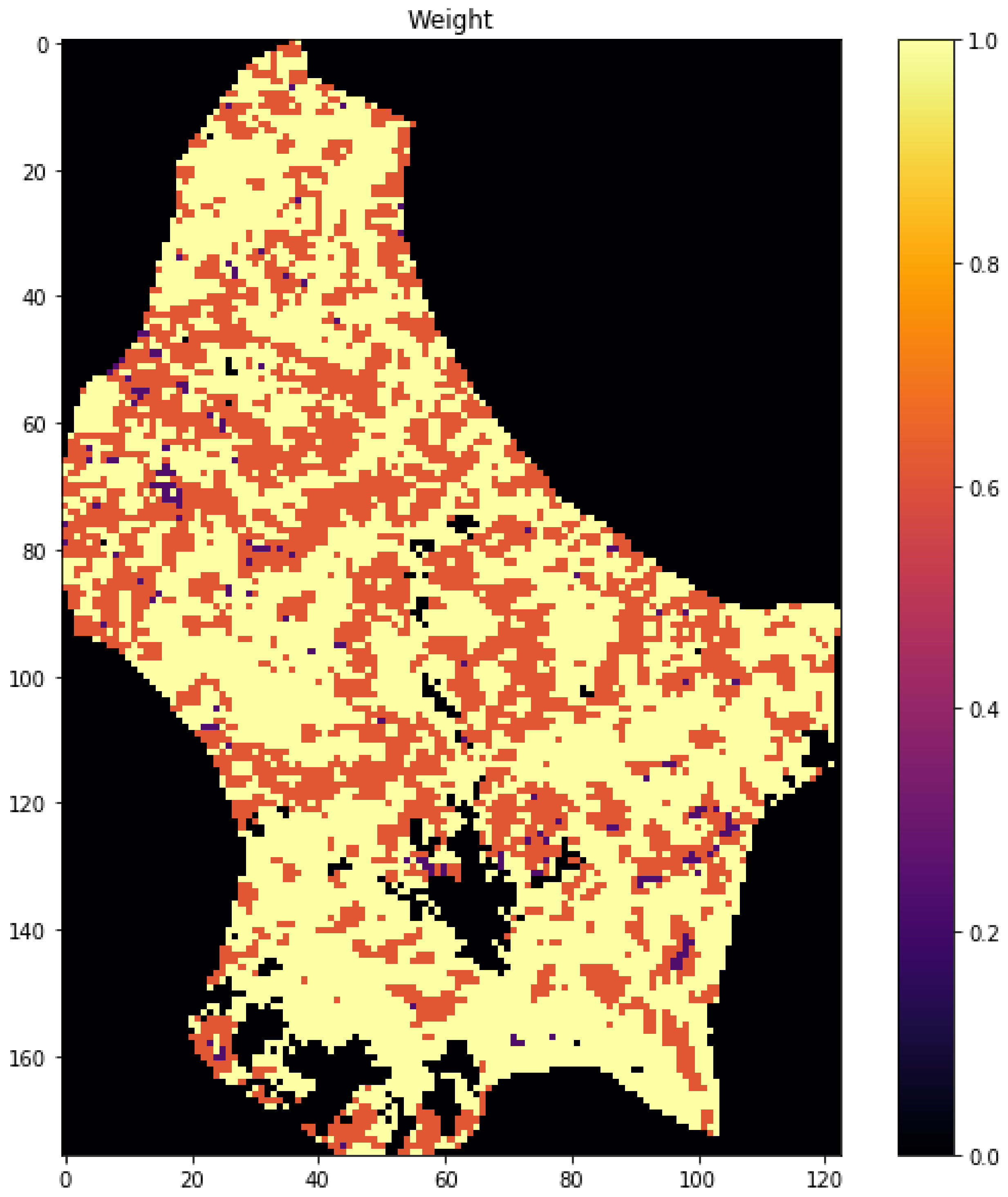
for valid values of `i`. This should then work correctly for arrays.

```
# do exercise here
```

The code you write can again be split into two very simple functions that you ought to test: One is just masking and shfting the QA data to obtain the SFC_QC flag on its own. A second part deals with calculating the weights depending on whether the SFC_QC flag has values of 0, 1, 2 or 3. For any other values, it ought to be 0.

```
def get_sfc_qc(qa_data, mask57 = 0b11100000):
    sfc_qa = np.right_shift(np.bitwise_and(qa_data, mask57), 5)
    return sfc_qa

def get_scaling(sfc_qa, golden_ratio=0.61803398875):
    weight = np.zeros_like(sfc_qa, dtype=np.float)
    for qa_val in [0, 1, 2, 3]:
        weight[sfc_qa == qa_val] = np.power(golden_ratio, float(qa_val))
    return weight


sfc_qa = get_sfc_qc(qa_data)
weights = get_scaling(sfc_qa)

plt.figure(figsize=(10,10))
plt.imshow(weights, vmin=0, vmax=1, interpolation="nearest",
           cmap=plt.cm.inferno)
plt.colorbar()
plt.title("Weight")
```

```
Text(0.5, 1.0, 'Weight')
```



3.4.3 A time series

You should now know how to access and download datasets from the NASA servers and have developed functions to do this.

You should also know how to select a dataset from a set of hdf files, and mosaic, mask and crop the data to correspond to some vector boundary. This is a very common task in geospatial processing.

You should also know how to evaluate QA information and use this to determine some quality weight. This includes an understanding of how to interpret and use binary data fields in a QA dataset.

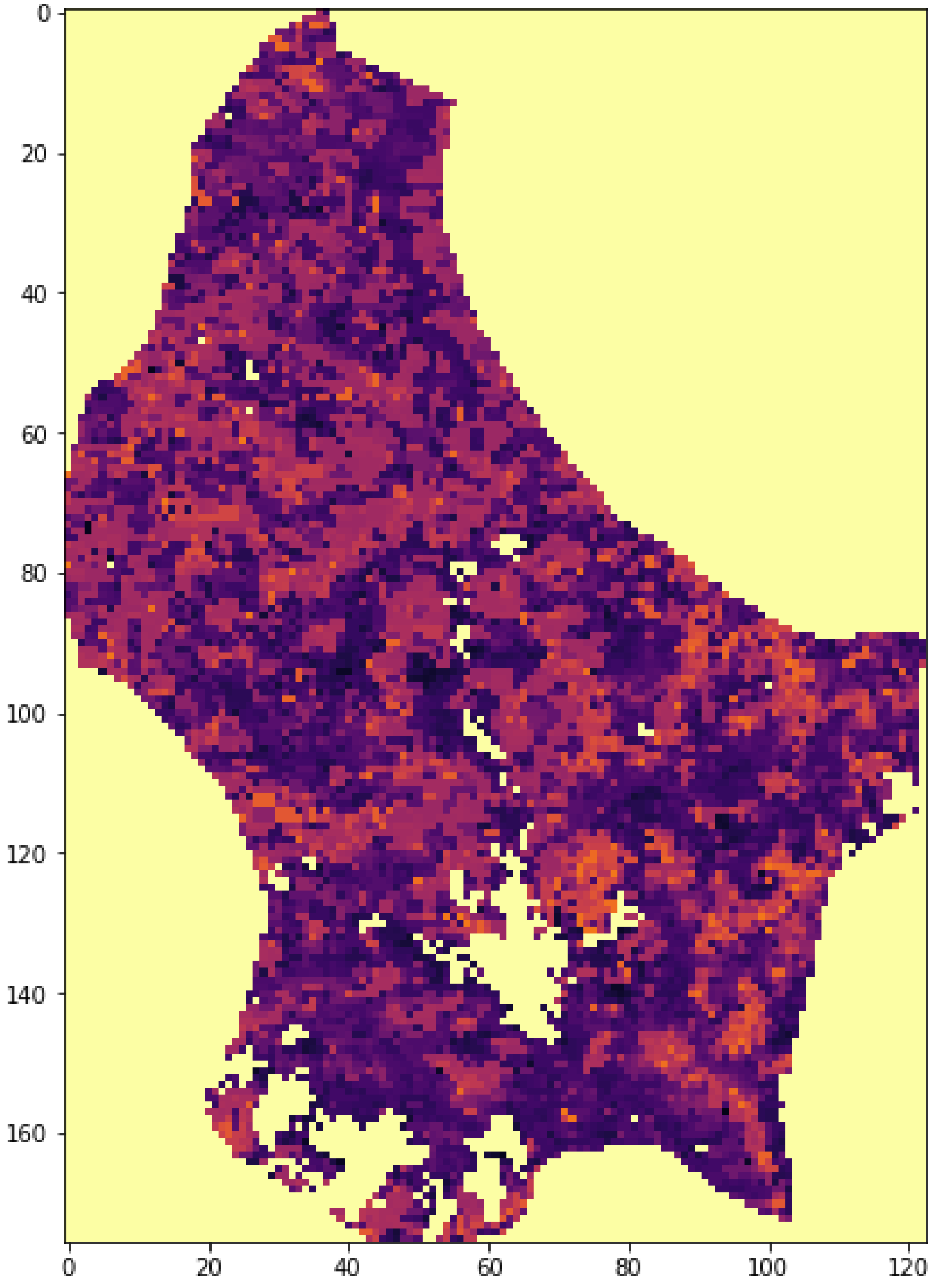
We now consider the case where we want to analyse a time series of data. We will use LAI over time to exemplify this.

We have already developed a file that returns an array (or a GeoTIFF) for a given date above, called `mosaic_and_clip`. We can loop over different dates while feeding the data into a so-called data cube. This is how you get the LAI

```
tiles = []
for h in [17, 18]:
    for v in [3, 4]:
        tiles.append(f"h{h:02d}v{v:02d}")
print(tiles)
year = 2017
doy = 149
data = mosaic_and_clip(tiles,
                    doy,
                    year,
                    folder="data/",
                    layer="Lai_500m",
                    shpfile="data/TM_WORLD_BORDERS-0.3.shp",
                    country_code="LU",
                    frmat="MEM")
plt.figure(figsize=(10,10))
plt.imshow(data/10., vmin=0, vmax=10, interpolation="nearest",
          cmap=plt.cm.inferno)
```

```
['h17v03', 'h17v04', 'h18v03', 'h18v04']
```

```
<matplotlib.image.AxesImage at 0x121e4c278>
```



Exercise 3.4.5

- Test that the code above works for reading the QA dataset as well.
- Write a function that reads both the LAI dataset and the QA data, scales the LAI data appropriately and produces a weight from the QA data.

- Return the LAI data and the weight

Hint

Scale LAI by multiplying by 0.1 as above.

```
# do exercise here
```

You should end up with something like:

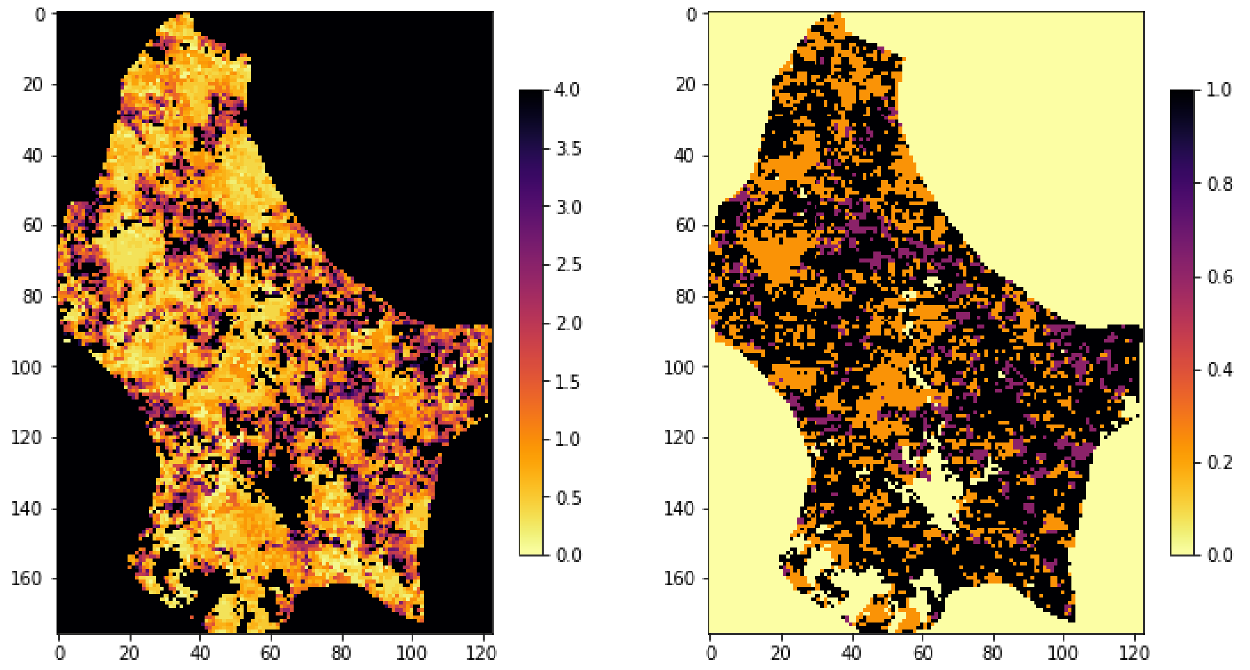
```
def process_single_date(tiles,
                        doy,
                        year,
                        folder="data/",
                        shpfile="data/TM_WORLD_BORDERS-0.3.shp",
                        country_code="LU",
                        frmat="MEM"):

    lai_data = mosaic_and_clip(tiles,
                        doy,
                        year,
                        folder=folder,
                        layer="Lai_500m",
                        shpfile=shpfile,
                        country_code=country_code,
                        frmat="MEM")*0.1
    # Note the scaling!

    qa_data = mosaic_and_clip(tiles,
                        doy,
                        year,
                        folder=folder,
                        layer="FparLai_QC",
                        shpfile=shpfile,
                        country_code=country_code,
                        frmat="MEM")
    sfc_qa = get_sfc_qc(qa_data)

    weights = get_scaling(sfc_qa)
    return lai_data, weights

tiles = []
for h in [17, 18]:
    for v in [3, 4]:
        tiles.append(f"h{h:02d}v{v:02d}")

year = 2017
doy = 273
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(12, 24))

lai, weights =  process_single_date(tiles,
                        doy,
                        year)

img1 = axs[0].imshow(lai, interpolation="nearest", vmin=0, vmax=4,
              cmap=plt.cm.inferno_r)
img2 = axs[1].imshow(weights, interpolation="nearest", vmin=0,
              cmap=plt.cm.inferno_r)
```

(continues on next page)

(continued from previous page)

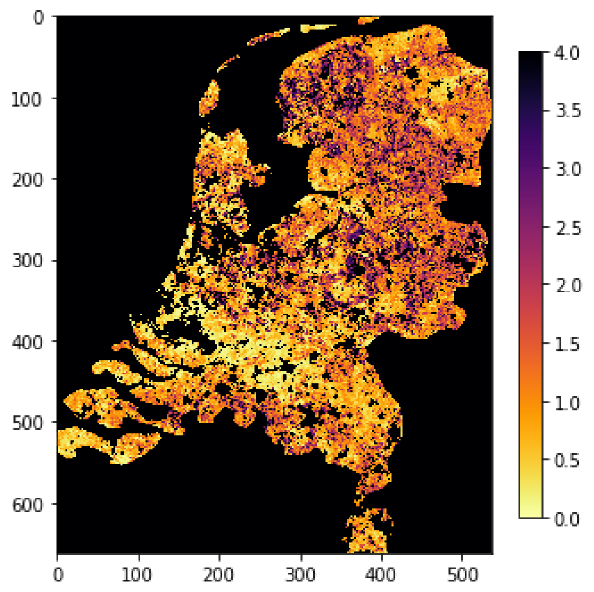
```
plt.colorbar(img1,ax=axs[0],shrink=0.2)
plt.colorbar(img2,ax=axs[1],shrink=0.2)
```

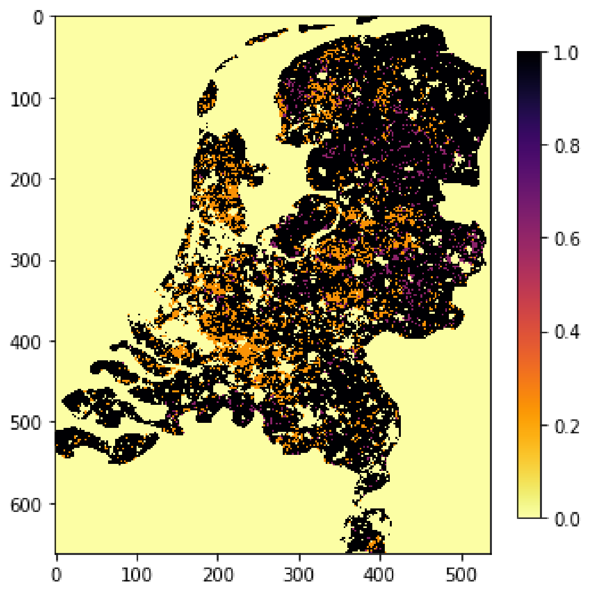
```
<matplotlib.colorbar.Colorbar at 0x1226e8c18>
```



```
tiles = []
for h in [17, 18]:
    for v in [3, 4]:
        tiles.append(f"h{h:02d}v{v:02d}")

year = 2017
doy = 273
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(12, 24))

lai, weights =  process_single_date(tiles,
                    doy,
                    year, country_code="NL")

img1 = axs[0].imshow(lai, interpolation="nearest", vmin=0, vmax=4,
              cmap=plt.cm.inferno_r)
img2 = axs[1].imshow(weights, interpolation="nearest", vmin=0,
              cmap=plt.cm.inferno_r)

plt.colorbar(img1,ax=axs[0],shrink=0.2)
plt.colorbar(img2,ax=axs[1],shrink=0.2)
```

```
<matplotlib.colorbar.Colorbar at 0x122b1ffd0>
```





Try it out:

Exercise 3.4.6

- Now we have some code to get the LAI and weight for one day, write a function that generates an annual dataset of LAI and weight
- show the dataset shapes
- Show the code works by plotting datasets for the beggining, middle and end of year

Hint

The result should be a set of two 3D numpy arrays

Really, all you need do is loop over each day, and add the new dataszet to a list.

Remember to create an initial empty list before you loop over day.

Think what might happen if the data for some day is missing.

```
# do exercise here
```

Although there are other ways of doing this, we can build up on what we had before, and just loop over days

```
from datetime import datetime, timedelta


def process_timeseries(year,
                       tiles,
                       folder,
                       shpfile,
                       country_code,
                       verbose=True):

    today = datetime(year, 1, 1)
    dates = []
    for i in range(92):
        if (i%10 == 0) and verbose:
            print(f"Doing {str(today):s}")
```

(continues on next page)

(continued from previous page)

```
        if today.year != year:
            break
        doy = int(today.strftime("%j"))

        this_lai, this_weight = process_single_date(
            tiles,
            doy,
            year,
            folder=folder,
            shpfile=shpfile,
            country_code=country_code,
            frmat="MEM")
        if doy == 1:
            # First day, create outputs!
            ny, nx = this_lai.shape
            lai_array = np.zeros((ny, nx, 92))
            weights_array = np.zeros((ny, nx, 92))
        lai_array[:, :, i] = this_lai
        weights_array[:, :, i] = this_weight
        dates.append(today)
        today = today + timedelta(days=4)
    return dates, lai_array, weights_array


tiles = []
for h in [17, 18]:
    for v in [3, 4]:
        tiles.append(f"h{h:02d}v{v:02d}")

year = 2017

dates, lai_array, weights_array = process_timeseries(
    year,
    tiles,
    folder="data/",
    shpfile="data/TM_WORLD_BORDERS-0.3.shp",
    country_code="NL")
```

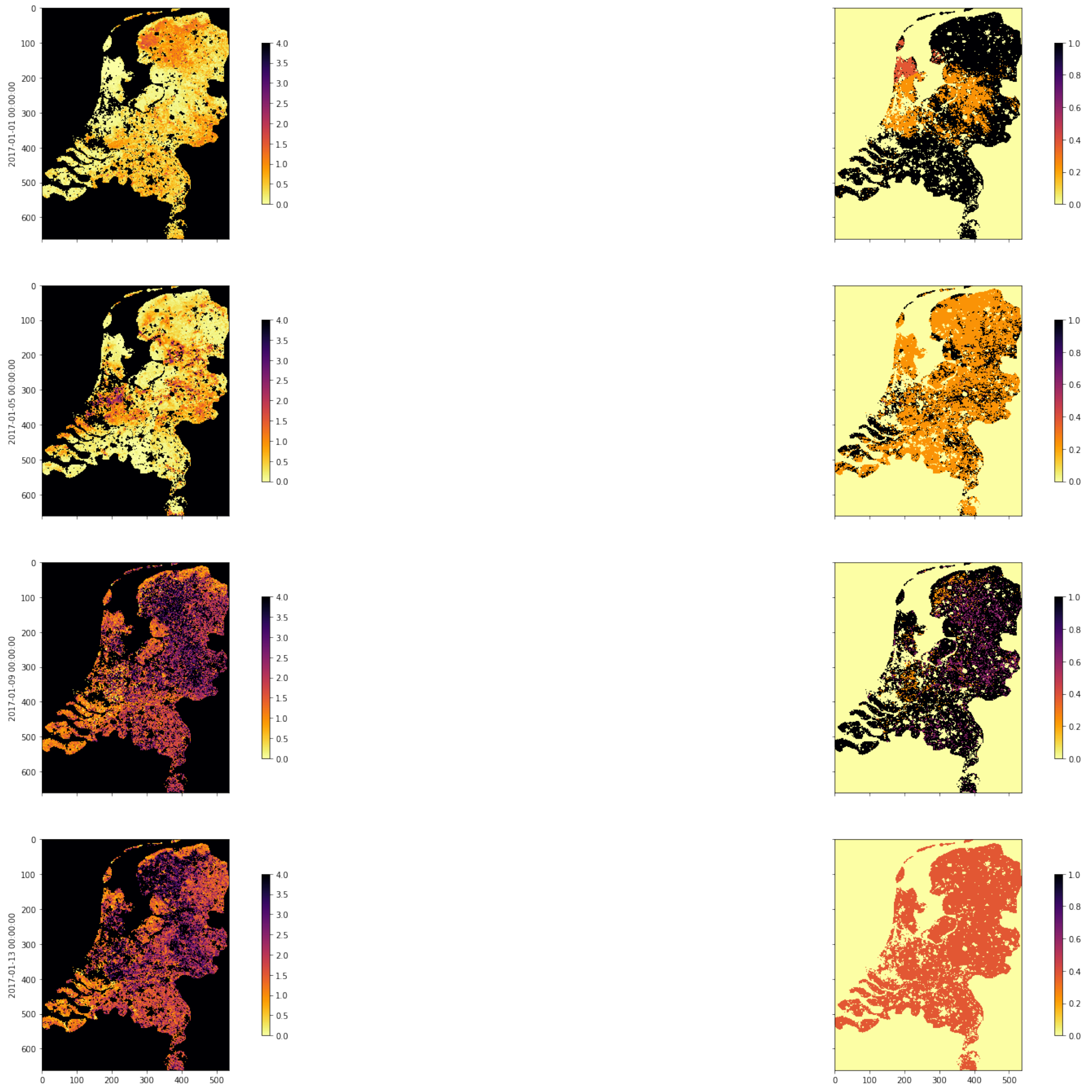
```
Doing 2017-01-01 00:00:00
Doing 2017-02-10 00:00:00
Doing 2017-03-22 00:00:00
Doing 2017-05-01 00:00:00
Doing 2017-06-10 00:00:00
Doing 2017-07-20 00:00:00
Doing 2017-08-29 00:00:00
Doing 2017-10-08 00:00:00
Doing 2017-11-17 00:00:00
Doing 2017-12-27 00:00:00
```

```
fig, axs = plt.subplots(nrows=4, ncols=2, sharex=True,
                        sharey=True, figsize=(32, 24))

for i, tstep in enumerate([10, 30, 60, 80]):
    img1 = axs[i][0].imshow(lai_array[:, :, tstep],
                    interpolation="nearest", vmin=0, vmax=4,
            cmap=plt.cm.inferno_r)
```

(continues on next page)

(continued from previous page)

```
    img2 = axs[i][1].imshow(weights_array[:, :, tstep],
                    interpolation="nearest", vmin=0, vmax=1,
            cmap=plt.cm.inferno_r)

    plt.colorbar(img1,ax=axs[i][0],shrink=0.7)
    plt.colorbar(img2,ax=axs[i][1],shrink=0.7)
    axs[i][0].set_ylabel(dates[i])
```



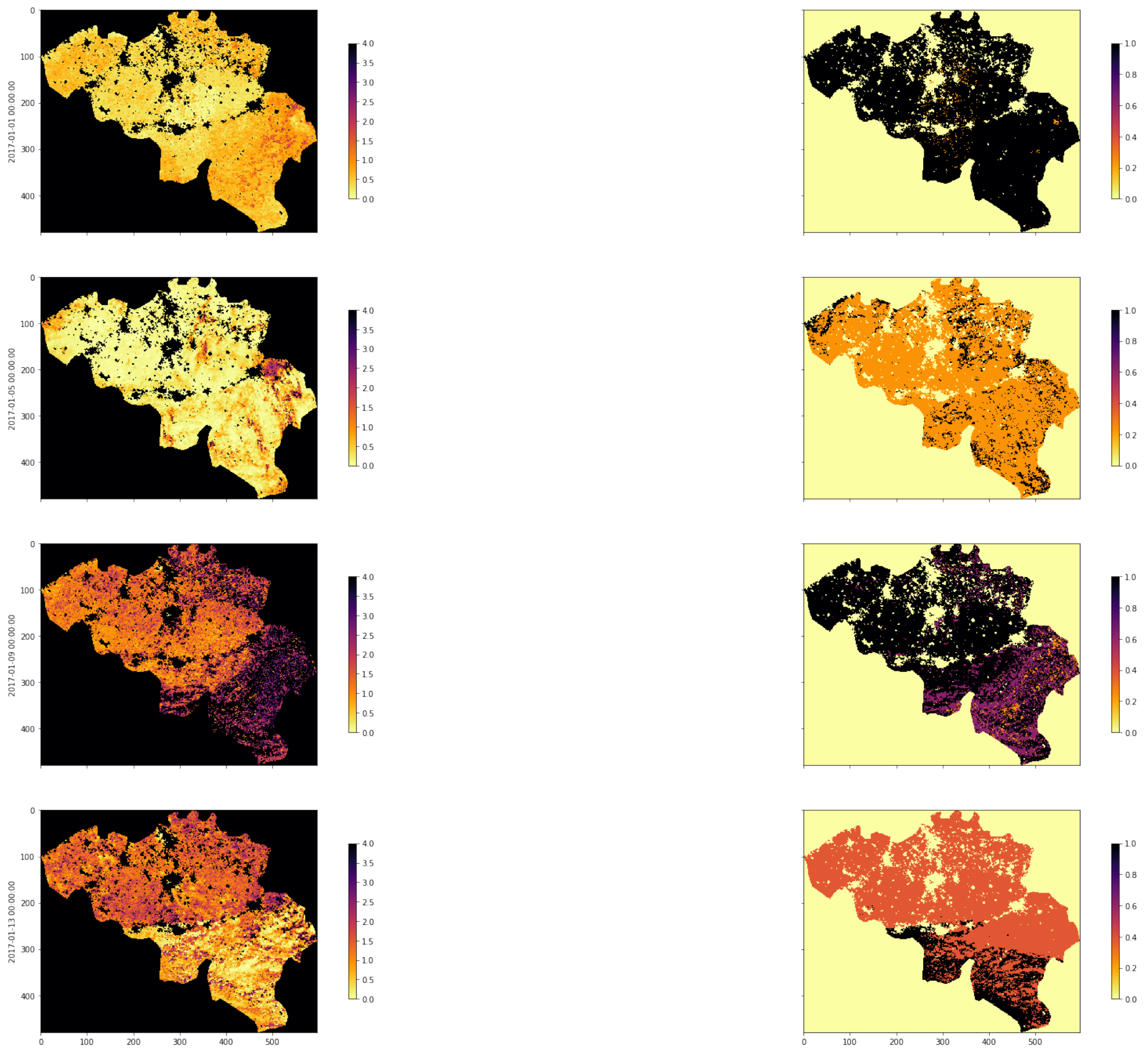
```
tiles = []
for h in [17, 18]:
    for v in [3, 4]:
        tiles.append(f"h{h:02d}v{v:02d}")
```

(continues on next page)

(continued from previous page)

```
year = 2017

dates, lai_array, weights_array = process_timeseries(
    year,
    tiles,
    folder="data/",
    shpfile="data/TM_WORLD_BORDERS-0.3.shp",
    country_code="BE")

fig, axs = plt.subplots(
    nrows=4, ncols=2, sharex=True, sharey=True, figsize=(32, 24))

for i, tstep in enumerate([10, 30, 60, 80]):
    img1 = axs[i][0].imshow(
        lai_array[:, :, tstep],
        interpolation="nearest",
        vmin=0,
        vmax=4,
        cmap=plt.cm.inferno_r)
    img2 = axs[i][1].imshow(
        weights_array[:, :, tstep],
        interpolation="nearest",
        vmin=0,
        vmax=1,
        cmap=plt.cm.inferno_r)

    plt.colorbar(img1, ax=axs[i][0], shrink=0.7)
    plt.colorbar(img2, ax=axs[i][1], shrink=0.7)
    axs[i][0].set_ylabel(dates[i])
```

```
Doing 2017-01-01 00:00:00
Doing 2017-02-10 00:00:00
Doing 2017-03-22 00:00:00
Doing 2017-05-01 00:00:00
Doing 2017-06-10 00:00:00
Doing 2017-07-20 00:00:00
Doing 2017-08-29 00:00:00
Doing 2017-10-08 00:00:00
Doing 2017-11-17 00:00:00
Doing 2017-12-27 00:00:00
```



Now let's read the data in:

3.4.4 Weighted interpolation

3.4.4.1 Smoothing

Some animations to help understand how we can use convolution to perform a weighted interpolation are given. You should got through these if you have not previously come across convolution filtering.

In convolution, we combine a *signal* y with a *filter* f to achieve a filtered signal. For example, if we have an noisy signal, we will attempt to reduce the influence of high frequency information in the signal (a 'low pass' filter, as we let the low frequency information *pass*).

We can perform a weighted interpolation by:

- numerator = smooth(signal $\times$ weight)
- denominator = smooth(weight)

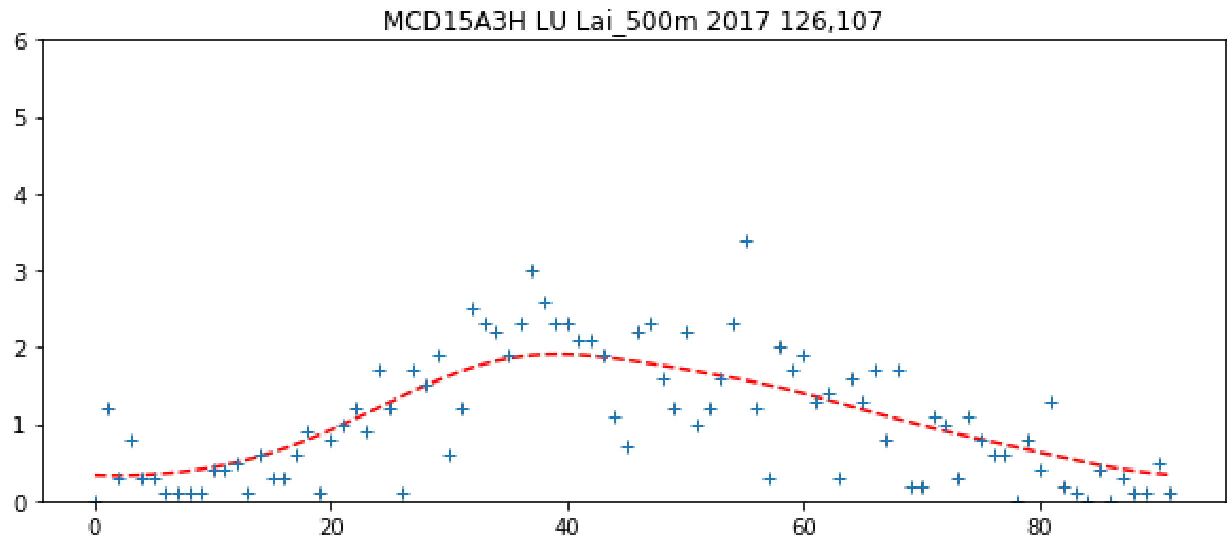
- result = numerator/denominator

```
sigma = 8
import scipy
import scipy.ndimage.filters

x = np.arange(-3*sigma,3*sigma+1)
gaussian = np.exp((-(x/sigma)**2)/2.0)

FIPS ='LU'
dates, lai_array, weights_array = process_timeseries( year, tiles, folder="data/",
                                                          shpfile="data/TM_WORLD_BORDERS-0.
↪3.shp", country_code=FIPS)
print(lai_array.shape, weights_array.shape) #Check the output array shapes

numerator = scipy.ndimage.filters.convolve1d(lai_array * weights_array, gaussian,␣
↪axis=2,mode='wrap')
denominator = scipy.ndimage.filters.convolve1d(weights_array, gaussian, axis=2,mode=
↪'wrap')

# avoid divide by 0 problems by setting zero values
# of the denominator to not a number (NaN)
denominator[denominator==0] = np.nan

interpolated_lai = numerator/denominator
print(interpolated_lai.shape)
```

```
Doing 2017-01-01 00:00:00
Doing 2017-02-10 00:00:00
Doing 2017-03-22 00:00:00
Doing 2017-05-01 00:00:00
Doing 2017-06-10 00:00:00
Doing 2017-07-20 00:00:00
Doing 2017-08-29 00:00:00
Doing 2017-10-08 00:00:00
Doing 2017-11-17 00:00:00
Doing 2017-12-27 00:00:00
(176, 123, 92) (176, 123, 92)
(176, 123, 92)
```

```
## find where the weight is highest, and lets look there!
sweight = weights_array.sum(axis=2)
r,c = np.where(sweight == np.max(sweight))
plt.figure(figsize=(10,4))
plt.title(f'{product} {FIPS} {params[0]} {year} {r[0]},{c[0]}')
ipixel = 0 # To plot the i-th pixel
plt.plot((interpolated_lai)[r[ipixel],c[ipixel],:],'r--')
plt.plot((lai_array)[r[ipixel],c[ipixel],:],'+')
plt.ylim(0,6)
```

```
(0, 6)
```



Exercise 3.4.7

- select some pixels (row, col) from the lai dataset and plot the original LAI, the interpolated LAI, and the weight

```
# do exercise here
```

3.4.5 Making movies

It is often useful to animate time series information. There are several ways of doing this.

Bear in mind that the larger the datasets, number of images and/or frames, the more time it is likely to take to generate the animations. You probably don't want more than around 100 frames to make an animation of this sort.

The two approaches we will use are:

- Javascript HTML in the notebook using `anim.to_jshtml()` from `matplotlib.animation`
- Animated gif using the `imageio` library

3.4.5.1 Javascript HTML

This approach uses javascript in html within the notebook to genrate an animation and player. The player is useful, in that we can easily stop at and explore individual frames.

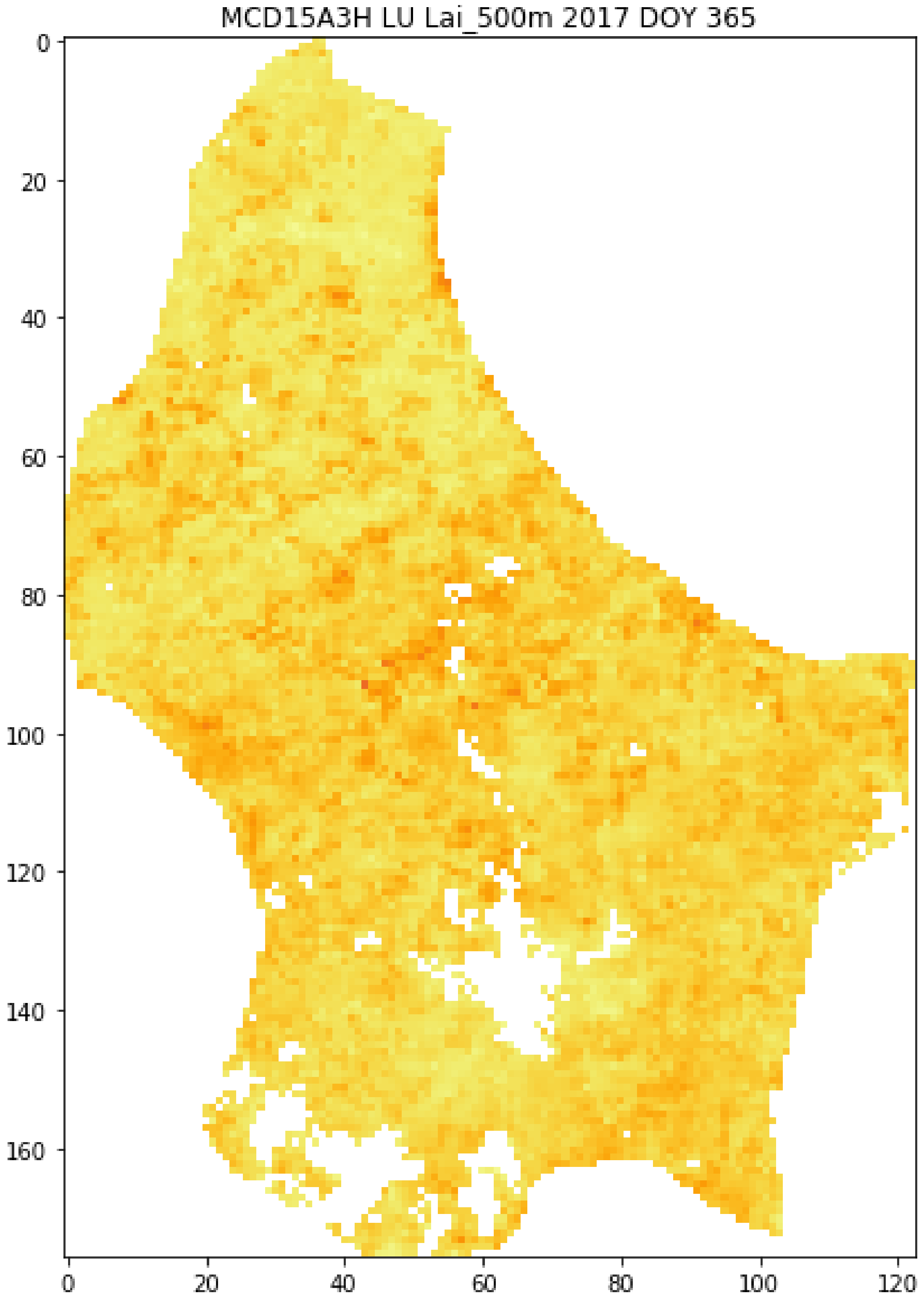
The HTML representation is written to a temporary directory (internally to anim.to_jshtml()) but deleted on exit.

```
from matplotlib import animation
import matplotlib.pylab as plt
from IPython.display import HTML

'''
lai movie javascript/html (jshtml)
'''

#fig, ax = plt.subplots(figsize=(10,10))
fig = plt.figure(0,figsize=(10,10))

```

(continues on next page)

(continued from previous page)

```
# define an animate function
# with the argument i, the frame number
def animate(i):
    # show frame i of the ilai dataset
    plt.figure(0)
    im = plt.imshow(interpolated_lai[:,:,i],vmin=0,vmax=6,cmap=plt.cm.inferno_r)
    plt.title(f'{product} {FIPS} {params[0]} {year} DOY {4*i+1:03d}')
    # make sure to return a tuple!!
    return (im,)

# set up the animation
anim = animation.FuncAnimation(fig, animate,
                               frames=interpolated_lai.shape[2], interval=40,
                               blit=True)

# display animation as HTML
HTML(anim.to_jshtml())
```


MCD15A3H LU Lai_500m 2017 DOY 365
0
20
40
60
80
100
120
140
160
0
20
40
60
80
100
120

3.4.5.2 Animated gif

In the second approach, we save individual frames of an animation, and read them in, using `imageio.imread()` into a list. We choose to write the individual frames here to a temporary directory (so they are cleaned up on exit).

This list of `imageio` datasets is then fed to `` `imageio.mimsave() `` <https://imageio.readthedocs.io/en/stable/userapi.html>`__ to save the sequence as an animated gif. This can then be displayed in a notebook cell (or otherwise). Note that the file data/MCD15A3H.A2017.h1_78_v0_34_LU.006.gif is saved in this case.

```
import imageio
import tempfile
from pathlib import Path

'''
lai movie as animated gif
'''

# switch interactive plotting off
# as we just want to save the frames,
# not plot them now
plt.ioff()

allopfile = Path('images', f'{product}_{FIPS}_{params[0]}_{year}_DOY{4*i+1:03d}')
    #get_filename(FIPS,year,doy,tile,destination_folder='images')

images = []
with tempfile.TemporaryDirectory() as tmpdirname:
    ofile = f'{tmpdirname}/tmp.png'

    for i in range(interpolated_lai.shape[2]):
        plt.figure(0,figsize=(10,10))
        # don' display the interim frames
        plt.ioff()
        plt.clf()
        plt.imshow(interpolated_lai[:,:,i],vmin=0,vmax=6,cmap=plt.cm.inferno_r)
        plt.title(f'{product} {FIPS} {params[0]} {year} DOY {4*i+1:03d}')
        plt.colorbar(shrink=0.85)
        plt.savefig(ofile)
        images.append(imageio.imread(ofile))
plt.clf()
imageio.mimsave(f'{allopfile}.gif', images)
print(f'{allopfile}.gif')
# switch interactive plotting back on
plt.ion()
```

```
images/MCD15A3H_LU_Lai_500m_2017_DOY365.gif
```

```
<Figure size 720x720 with 0 Axes>
```

Exercise 3.4.8

- Write a set of functions, with clear commenting and document strings tha:
  - develops the LAI and QA dataset for a given year to produce LAI data and weight
  - produces interpolated / smoothed LAI data as a numpy 3D array for the year
  - saves the resultant numpy dataset in a `npz` file

- Once y

Hint

Put all of the material above together.

Remember `np.savez()`!

```
# do exercise here
```

Table of Contents

2.8.16 3.5 LAI Movies

2.8.17 3.6 Reconciling projections

Table of Contents

3.6.A Introduction

This section of notes is optional to the course, and the tutor may decide *not* to go through this in class. That said, the information and obexamples contained here can be very useful for accessing and processing certain types of geospatial data.

In particular, we deal with obtaining climate data records from ECMWF that we will later use for model fitting. These data come in a netcdf format (commonly used for climate data) with a grid in latitude/longitude. To 'overlay' these data with another dataset (e.g. the MODIS LAI product that we have been using) in a different (equal area) projection, we use the `gdal` function

```
gdal.ReprojectImage(src, dst, src_proj, dst_proj, interp)
```

where:

```
src      : a source dataset that we want to process
dst      : a blank destination dataset that we set up with the
           required (output) data type, shape, and geotransform and projection
src_proj : the source dataset projection wkt
dst_proj : the destination projection wkt
interp   : the required interpolation method, e.g. gdalconst.GRA_Bilinear
```

where wkt stands for well known text and is a projection format string.

Other codes we use are ones we have developed earlier.

In these notes, we will learn:

```
* how to access an ECMWF daily climate dataset (from ERA interim)
* how to reproject the dataset to match another spatial dataset (MODIS LAI)
```

We will then save some datasets that we will use later in the notes. For this reason, it's possile to skip this section, and return to it later.

3.6.A.1 Requirements

We will need to:

- make sure we have the MODIS LAI dataset locally
- read them in for a given country.
- generate and interpolated LAI dataset
- register with ecmwf, install ecmwfapi
- get the temperature datasset from ECMWF for 2006 and 2017 for Europe
- get the country borders shapefile

```
# required general imports
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
import sys
import os
from pathlib import Path
import gdal
from datetime import datetime, timedelta
year = 2016
country_code = 'LU'
```

You can run all of the below with the script, unless you want to change any of the conditions (e.g. year or country):

```
# This does the same as the cells below but in one script
%run geog0111/Chapter3_6A_prerequisites.py $country_code $year
```

```
['geog0111/Chapter3_6A_prerequisites.py', 'LU', '2016'] 2016 LU
europe_data_2016_2017.nc exists
GEOGCS["Coordinate System imported from GRIB file",DATUM["unknown",SPHEROID["Sphere",
↪6371200,0]],PRIMEM["Greenwich",0],UNIT["degree",0.0174532925199433]]
Refreshing nc file europe_data_2016_2017.nc
data/europe_data_2016.nc
data/europe_data_2017.nc
Running outside UCL Geography. Will try to download data.
 landcover_data_2017_LU.npz
This might take a while!
trying http://www2.geog.ucl.ac.uk/~plewis/geog0111_data/lai_files/
trying http://www2.geog.ucl.ac.uk/~plewis/geog0111_data/
trying http://www2.geog.ucl.ac.uk/~ucfajlg/geog0111_data/
server may be down
```

3.6.A.2 Get the MODIS LAI datasets for 2016/2017 for W. Europe

You will probably already have this dataset, but running the code below will make sure that you do.

```
# get the MODIS LAI dataset for 2016/2017 for W. Europe
# should be 736 files
from geog0111.geog_data import procure_dataset
from pathlib import Path

num_hdf = len(list(Path('data').glob('MCD15A3H*hdf')))
if num_hdf < 736:
    _ = procure_dataset("lai_files",verbose=False)
```

3.6.A.3 Get the shapefile for country borders

Again, you should already have this, but just to make sure:

```
import requests
import shutil
from pathlib import Path

force = False
# zip file
zipfile = 'TM_WORLD_BORDERS-0.3.zip'
# URL
tm_borders_url = f"http://thematicmapping.org/downloads/{zipfile}"
# destibnation folder
destination_folder = Path('data')

# set up some filenames
zip_file = destination_folder.joinpath(zipfile)
shape_file = zip_file.with_name(zipfile.replace('zip', 'shp'))

# download zip if need to
if not Path(zip_file).exists():
    r = requests.get(tm_borders_url)
    with open(zip_file, 'wb') as fp:
        fp.write(r.content)

# extract shp from zip if need to
if force or not Path(shape_file).exists():
    shutil.unpack_archive(zip_file.as_posix(), extract_dir=destination_folder)
```

3.6.A.4 Read the LAI dataset for a given country and year

Run the code below to read in the LAI dataset for a given year and country. This uses codes we have developed in previous sections, interfaced through `process_timeseries()`.

In the code, we can save the dataset as an `npz` file, so that we can access it faster next time.

```
# read in the LAI data for given country code
from geog0111.process_timeseries import process_timeseries
import scipy
import scipy.ndimage.filters
```

(continues on next page)

(continued from previous page)

```
'''
Note, the saved npz file can be quite large
e.g. 8.1 G for France.

You can override saving it by setting save = False
but if it is saved, it will be faster to access
data the next time you need it.

If you have a slow network, you might set download=False

Does interpolation of dataset with Gaussian smoother width sigma
'''
save = True
download = True
# want sigma as low as we can deal with, whilst
# still interpolating effectively
sigma = 3


tiles = []
for h in [17, 18]:
    for v in [3, 4]:
        tiles.append(f"h{h:02d}v{v:02d}")

fname = f'lai_data_{year}_{country_code}.npz'
ofile = Path('data')/fname
done = False

if ofile.exists():
    done = True

# try to download it from server
if download:
    done = procure_dataset(fname,verbose=True)

if not done:
    # else generate it
    dates, lai_array, weights_array = process_timeseries(year,tiles,frmat='MEM',\
                                                  country_code=country_code)

    lai = {'dates':dates, 'lai':lai_array, 'weights':weights_array}

    # set up filter
    x = np.arange(-3*sigma,3*sigma+1)
    gaussian = np.exp((-(x/sigma)**2)/2.0)

    FIPS = country_code
    dates, lai_array, weights_array = lai['dates'],lai['lai'],lai['weights']
    print(lai_array.shape, weights_array.shape) #Check the output array shapes
    print('interpolating ...')
    numerator = scipy.ndimage.filters.convolve1d(lai_array * weights_array, gaussian,␣
↪axis=2,mode='wrap')
    denominator = scipy.ndimage.filters.convolve1d(weights_array, gaussian, axis=2,
↪mode='wrap')

    # avoid divide by 0 problems by setting zero values
    # of the denominator to not a number (NaN)
```

(continues on next page)

(continued from previous page)

```
    denominator[denominator==0] = np.nan

    interpolated_lai = numerator/denominator
    print(interpolated_lai.shape)

    # need to convert to dict to be able to assign
    lai['interpolated_lai'] = interpolated_lai
    print('saving ...')
    if save:
        np.savez_compressed(ofile,**lai)
```

```
# test that LAI is sensible
# a quick look at some stats to see if there are data there
# and they are sensible
lai = np.load(ofile)
print(np.nanmean(lai['lai'],axis=(0,1)))
print(np.nanmean(lai['weights'],axis=(0,1)))
# does it have the interpolated value?
if 'interpolated_lai' in list(lai.keys()):
    print(np.nanmean(lai['interpolated_lai'],axis=(0,1)))
```

```
[12.46488359 12.9707779  12.66668052 12.57172487 12.5430109  12.38457132
 12.23579545 12.27250554 12.26942905 12.43250647 12.48960643 12.60697062
 12.57322154 12.36961382 12.51265244 12.52181264 12.50411123 12.66635717
 12.75169531 12.26231061 12.86824649 12.99491408 12.97218681 13.04568551
 13.2463969  13.28130081 12.77618718 13.17412232 12.98403548 13.40995935
 12.5675813  12.53988821 13.8197755  14.04873891 13.772949   13.5904102
 14.19393016 13.86867609 13.686165   13.58825758 14.10188008 13.86718404
 13.85474409 13.53060329 13.50408814 13.1711105  13.75550166 13.87436253
 13.7763119  13.36458333 13.30586197 13.46170085 13.10407428 13.24387934
 13.86328529 13.04242424 13.41308666 12.96385347 13.70315503 13.45206486
 13.80310421 13.93165188 12.91742886 13.32721268 13.45185698 13.4931772
 13.83985588 13.49764412 13.10169993 12.66001016 12.77271803 13.33361511
 13.03124076 12.59559775 12.72172949 12.95489191 12.98522265 12.38168422
 12.22975795 12.65463784 12.55263766 13.11926275 12.82906042 12.97523097
 12.24103843 12.50066057 12.27802568 12.41912879 12.26183019 12.24083518
 12.72493071 12.60570953]
[0.19874654 0.19874654 0.19874654 0.19874654 0.283081   0.35662385
 0.26963431 0.19570155 0.21511234 0.47091527 0.50809763 0.5203252
 0.49158247 0.3148385  0.4751026  0.46047539 0.40051971 0.5203252
 0.52023698 0.14255856 0.52021934 0.51956649 0.51778441 0.49043559
 0.51111483 0.5004929  0.28549586 0.46042246 0.41188271 0.49391153
 0.21685914 0.19552705 0.47160901 0.43956678 0.4637749  0.38629833
 0.42763916 0.44436605 0.41216502 0.37258863 0.43220906 0.44254868
 0.44187819 0.42624525 0.44708329 0.35459134 0.45583491 0.43577323
 0.44803609 0.3884333  0.37978754 0.35983172 0.35277396 0.28014961
 0.43859633 0.26876897 0.40346633 0.30571634 0.45274714 0.47630242
 0.45258834 0.42075784 0.28426075 0.39557928 0.42928009 0.46751551
 0.44337797 0.46513351 0.3877981  0.23727371 0.31272117 0.50028116
 0.51651401 0.36189612 0.45934615 0.50984443 0.51780205 0.36099151
 0.17427367 0.46317168 0.33834976 0.20008836 0.19874654 0.19874654
 0.19874654 0.19874654 0.19874654 0.19874654 0.19874654 0.19874654
 0.19874654 0.19874654]
[0.65610413 0.65800351 0.62776398 0.57380242 0.51343854 0.4643343
 0.4385093  0.43811953 0.45779979 0.48876735 0.52140917 0.55237027
 0.58166253 0.61216015 0.64970439 0.70132282 0.77204754 0.86270811
```

(continues on next page)

(continued from previous page)

```
 0.97294838 1.09576577 1.22270548 1.3418187  1.44579277 1.53184345
 1.59929471 1.65154397 1.6967618  1.75069782 1.82931007 1.94638937
 2.10579342 2.29535476 2.48920358 2.66158148 2.79585874 2.88889225
 2.94499235 2.97291018 2.97854192 2.96459096 2.93244254 2.88443324
 2.82565677 2.76352022 2.70607503 2.65675994 2.61066125 2.56344996
 2.50817913 2.44477369 2.37730315 2.31850576 2.27771294 2.25864496
 2.26110334 2.28101681 2.31142542 2.34757848 2.38048346 2.40196121
 2.40629592 2.39235872 2.36532757 2.32891928 2.28460233 2.22955228
 2.15469741 2.05423776 1.92709977 1.78558512 1.64338691 1.51373368
 1.40292473 1.30638887 1.22091165 1.14252163 1.07085676 1.00815965
 0.95794105 0.92136489 0.89487309 0.86823828 0.82945063 0.76814057
 0.68530332 0.59536944 0.5209187  0.4812206  0.48085913 0.51419636
 0.56841624 0.62179996]
```

3.6.A.5 register with ECMWF and install the API

Follow the ECMWF instructions

First, you should register as a user with ECMWF.

The first time:

- register and follow the emailed instructions.
- read and acknowledge the conditions of access and related information.
- follow the instructions on how to access datasets to receive and set up an ECMWF key

Help is available online.

```
# install ecmwf api -- do this once only
ECMWF = 'https://software.ecmwf.int/wiki/download/attachments/56664858/ecmwf-api-
↪client-python.tgz'
try:
    from ecmwfapi import ECMWFDataServer
except:
    import os
    if os.name == 'nt':
        # on windows
        !pip install $ECMWF
    else:
        # on Unix/Linux
        !pip install --user $ECMWF
```

3.6.2.4 Get the 2t dataset from ECMWF for Europe

Run the code below to request and download the daily 2m temperature dataset for 2016 and 2017 (0.25 degree resolution) from the ECMWF ERA interim data.

If the file already exists locally, the request will be ignored.

If you do need to run the request, it may take several hours, depending on the ECMWF queue at the time.

```
from ecmwfapi import ECMWFDataServer
from pathlib import Path
from geog0111.geog_data import procure_dataset
```

(continues on next page)

(continued from previous page)

```
ecmwf_file = 'europe_data_2016_2017.nc'

if not (Path('data')/ecmwf_file).exists():
    # try to get it from UCL servers
    done = procure_dataset(ofile,verbose=True)
    if not done:
        server = ECMWFDataServer()
        print('requesting data ... may take some time')
        server.retrieve({
            "class": "ei",
            "dataset": "interim",
            "date": "2016-01-01/to/2017-12-31", # Time period
            "expver": "1",
            "levtype": "sfc",
            "param": "2t",            # Parameters. Here we use 2m Temperature (2t) ␣
↪See the ECMWF parameter database, at http://apps.ecmwf.int/codes/grib/param-db
            "stream": "oper",
            "type": "an",
            "time": "12",
            "step": "0",
            "area": "75/-20/10/60",     # Subset or clip to an area, here to Europe.␣
↪Specify as North/West/South/East in Geographic lat/long degrees. Southern latitudes␣
↪and Western longitudes must be given as negative numbers.
            "grid": "0.25/0.25",        # Regrid from the default grid to a regular␣
↪lat/lon with specified resolution. The first number is east-west resolution␣
↪(longitude) and the second is north-south (latitude).
            "format": "netcdf",         # Convert the output file from the default␣
↪GRIB format to NetCDF format. Requires "grid" to be set to a regular lat/lon grid.
            "target": f"data/{ecmwf_file}",  # The output file name. Set this to␣
↪whatever you like.
        })
else: print(f'{ecmwf_file} exists')
```

```
europe_data_2016_2017.nc exists
```

3.6.2.5 Generate dataset wkt and correct ECMWF file

The coordinate system of the ECMWF is peculiar in that it has a non-standard Earth spheroid.

According to ECMWF, the Earth model assumed for GRIB data (and inherited for other data products) uses a sphere with radius 6367.47 km, as defined in the WMO GRIB Edition 1 specifications, Table 7, GDS Octet 17.

We will mostly be dealing with netcdf files, as above. But it is not straightforward to get this projection information from the netcdf file.

Instead, we access an example grib format file and save it as `data/Pacific.wind.7days.grb`. We then generate a wkt (well known text) format file `` `data/grb.wkt `` <data/grb.wkt>'__ and insert this into the file using `gdal.Translate()`.

The code below is quite complicated, buit needed to fix file format problems. It also converts the integer data into physical units (temperature in K) using the `-unscale` option.

```
from osgeo import gdal
import requests
```

(continues on next page)

(continued from previous page)

```
from pathlib import Path
import numpy as np
from datetime import datetime,timedelta

# download example grib file from
url = "http://gribs2.gmn-usa.com/cgi-bin/" +\
        "weather_fetch.pl?parameter=wind&days=7&region=Pacific&dataset=nww3"
ofile = 'data/Pacific.wind.7days.grb'
overwrite = False

# get the example grib datafile
# see
# https://gis.stackexchange.com/questions/
# 289314/using-gdal-to-read-data-from-grib-file-in-python
output_fname = Path(ofile)
with requests.Session() as session:
    r1 = session.request('get',url)
    if r1.url:
        r2 = session.get(r1.url)
        data = r2.content
        d = 0
        if overwrite or (not output_fname.exists()):
            with open(output_fname, 'wb') as fp:
                d = fp.write(data)

dataset = gdal.Open(ofile)
wkt = dataset.GetProjection()
with open('data/grb.wkt', 'w') as fp:
    # write wkt to file
    d = fp.write(wkt)

# use this to fix the downloaded file
# which is ecmwf_file
ifile = f"data/{ecmwf_file}"

# need to sort the metadata
meta = gdal.Open(ifile).GetMetadata()
# get time info
timer = np.array([(datetime(1900,1,1) + timedelta(days=float(i)/24.)) \
for i in meta['NETCDF_DIM_time_VALUES'][1:-1].split(',')])

# pull the years info from ifile
# if the file is multiple years eg europe_data_2010_2011.nc
# then split it into multiple files
years = np.array(Path(ifile).stem.split('_'))[2:].astype(int)

# filter data for required year

for year in years:
    ofile = f'data/europe_data_{year}.nc'
    mask = np.logical_and(timer >= datetime(year,1,1),timer <= datetime(year+1,1,1))
    timer2 = timer[mask]
    bands = ' '.join([f'-b {i}' for i in (np.where(mask)[0]+1)])
    timer3 = '{'+','.join(np.array(meta['NETCDF_DIM_time_VALUES'][1:-1].split(',
↪'))[mask])+'}'
    timer4 = '{'+str(mask.sum())+',4}'
    options = f"-of netcdf -unscale -ot Float32 {bands} -mo NETCDF_DIM_time_VALUES=
↪{timer3}" + \
```

(continues on next page)

(continued from previous page)

```
                f" -mo NETCDF_DIM_time_DEF={timer4} -a_srs data/grb.wkt"
    gdal.Translate(ofile+'tmp',ifile,options=options)
    Path(ofile+'tmp').replace(ofile)
    print(ofile)
```

```
data/europe_data_2016.nc
data/europe_data_2017.nc
```

```
from osgeo import gdal
import requests
from pathlib import Path
import numpy as np

'''
Get the SRS 6974 for MODIS in case we want to use it
'''

url = 'http://spatialreference.org/ref/sr-org/6974/ogcwkt/'
ofile = 'data/modis_6974.wkt'
overwrite = False

# http://spatialreference.org/ref/sr-org/6974
output_fname = Path(ofile)
with requests.Session() as session:
    r1 = session.request('get',url)
    if r1.url:
        r2 = session.get(r1.url)
        data = r2.text
        d = 0
        if overwrite or (not output_fname.exists()):
            with open(output_fname, 'w') as fp:
                d = fp.write(data)

# test opening it
wkt2 = open(ofile,'r').readlines()

print(wkt2)
```

```
['PROJCS["MODIS Sinusoidal",GEOGCS["WGS 84",DATUM["WGS_1984",SPHEROID["WGS 84",
↪6378137,298.257223563,AUTHORITY["EPSG","7030"]],AUTHORITY["EPSG","6326"]],PRIMEM[
↪"Greenwich",0,AUTHORITY["EPSG","8901"]],UNIT["degree",0.01745329251994328,AUTHORITY[
↪"EPSG","9122"]],AUTHORITY["EPSG","4326"]],PROJECTION["Sinusoidal"],PARAMETER["false_
↪easting",0.0],PARAMETER["false_northing",0.0],PARAMETER["central_meridian",0.0],
↪PARAMETER["semi_major",6371007.181],PARAMETER["semi_minor",6371007.181],UNIT["m",1.
↪0],AUTHORITY["SR-ORG","6974"]]']
```

3.6.2.6 Get land cover data

The MODIS land cover product MCD12Q1 we need is on the server:

```
https://e4ftl01.cr.usgs.gov//MODV6_Cmp_C/MOTA/
```

This is an annual dataset, with access date `YYYY.01.01`.

We will use

```
Land Cover Type 3: Annual Leaf Area Index (LAI) classification
```

referenced as "`LC_Type3`" (valid range 0 to 10).

This is interpreted as:

Name	Value	Description
Water Bodies	0	At least 60% of area is covered by permanent water bodies.
Grasslands	1	Dominated by herbaceous annuals (<2m) includ- ing cereal croplands.
Shrublands	2	Shrub (1-2m) cover >10%.
Broadleaf Croplands	3	Dominated by herbaceous annuals (<2m) that are cultivated with broadleaf crops.
Savannas	4	Between 10-60% tree cover (>2m).
Evergreen Broadleaf Forests	5	Dominated by evergreen broadleaf and palmate trees (>2m). Tree cover >60%.
Deciduous Broadleaf Forests	6	Dominated by deciduous broadleaf trees (>2m). Tree cover >60%.
Evergreen Needle-leaf Forests	7	Dominated by evergreen conifer trees (>2m). Tree cover >60%.
Deciduous Needle-leaf Forests	8	Dominated by deciduous needleleaf (larch) trees (>2m). Tree cover >60%.
Non-Vegetated Lands	9	At least 60% of area is non-vegetated barren (sand, rock, soil) or permanent snow and ice with less than 10% vegetation.
Urban and Built-up Lands	10	At least 30% impervious surface area including building materials, asphalt, and vehicles.
Unclassified	255	Has not received a map label because of missing inputs.

```
from geog0111.get_modis_files import get_modis_files
'''
Get the MODIS LC files from the server
to store in data
'''
try:
    url = 'https://e4ftl01.cr.usgs.gov//MODV6_Cmp_C/MOTA/'
    filename = get_modis_files(1,year,[tiles],base_url=url,\
                                                  version=6,verbose=True,\
                                                  destination_folder='data',\
                                                  product='MCD12Q1')[0]
    print(filename)
except:
    print('server may be down')
```

```
server may be down
```

```
from geog0111.process_timeseries import mosaic_and_clip

'''
Extract and clip the dataset
'''
lc_data = mosaic_and_clip(tiles,
                    1,
                    year,
                    folder='data',
                    layer="LC_Type3",
                    shpfile='data/TM_WORLD_BORDERS-0.3.shp',
```

(continues on next page)

(continued from previous page)

```
                        country_code=country_code,
                        product='MCD12Q1',
                        frmat="MEM")
```

```
'''
Define LC table from userguide
https://lpdaac.usgs.gov/sites/default/\
          files/public/product_documentation/\
          mcd12_user_guide_v6.pdf
'''

table = '''
|Water Bodies|0|At least 60% of area is covered by permanent water bodies.|
|Grasslands|1|Dominated by herbaceous annuals (<2m) includ- ing cereal croplands.|
|Shrublands|2|Shrub (1-2m) cover >10%.|
|Broadleaf Croplands|3|Dominated by herbaceous annuals (<2m) that are cultivated with␣
↪broadleaf crops.|
|Savannas|4|Between 10-60% tree cover (>2m).|
|Evergreen Broadleaf Forests|5|Dominated by evergreen broadleaf and palmate trees (>
↪2m). Tree cover >60%.|
|Deciduous Broadleaf Forests|6|Dominated by deciduous broadleaf trees (>2m). Tree␣
↪cover >60%.|
|Evergreen Needleleaf Forests|7|Dominated by evergreen conifer trees (>2m). Tree␣
↪cover >60%.|
|Deciduous Needleleaf Forests|8|Dominated by deciduous needleleaf (larch) trees (>2m).
↪ Tree cover >60%.|
|Non-Vegetated Lands|9|At least 60% of area is non-vegetated barren (sand, rock,␣
↪soil) or permanent snow and ice with less than 10% vegetation.|
|Urban and Built-up Lands|10|At least 30% impervious surface area including building␣
↪materials, asphalt, and vehicles.|
|Unclassified|255|Has not received a map label because of missing inputs.|
'''

LC_Type3 = np.array([s.split('|')[1:-1] for s in table.split('\n')[1:-1]]).T


np.savez_compressed(f'data/landcover_{year}_{country_code}.npz',
                        LC_Type3=LC_Type3,lc_data=lc_data)
```

```
# Unique values for "LC_Type3"
def plot_land_cover(lc_data,year,country_code,cmap='tab20b'):
    '''
    Define LC table from userguide
    https://lpdaac.usgs.gov/sites/default/\
              files/public/product_documentation/\
              mcd12_user_guide_v6.pdf
    '''

    table = '''
    |Water Bodies|0|At least 60% of area is covered by permanent water bodies.|
    |Grasslands|1|Dominated by herbaceous annuals (<2m) includ- ing cereal croplands.|
    |Shrublands|2|Shrub (1-2m) cover >10%.|
    |Broadleaf Croplands|3|Dominated by herbaceous annuals (<2m) that are cultivated␣
↪with broadleaf crops.|
    |Savannas|4|Between 10-60% tree cover (>2m).|
    |Evergreen Broadleaf Forests|5|Dominated by evergreen broadleaf and palmate trees␣
↪(>2m). Tree cover >60%.|
```

(continues on next page)

(continued from previous page)

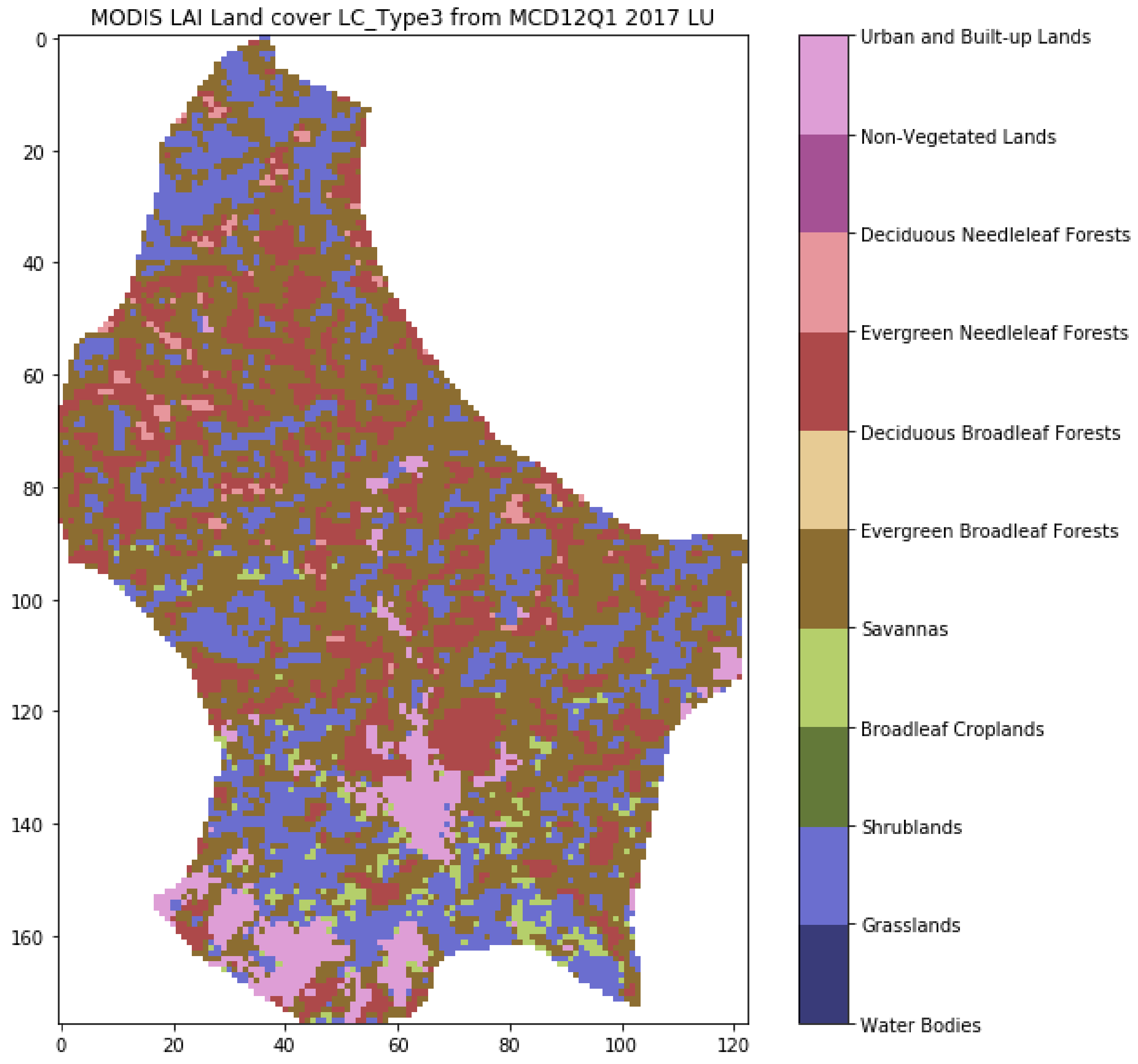
```
    |Deciduous Broadleaf Forests|6|Dominated by deciduous broadleaf trees (>2m). Tree␣
↪cover >60%.|
    |Evergreen Needleleaf Forests|7|Dominated by evergreen conifer trees (>2m). Tree␣
↪cover >60%.|
    |Deciduous Needleleaf Forests|8|Dominated by deciduous needleleaf (larch) trees (>
↪2m). Tree cover >60%.|
    |Non-Vegetated Lands|9|At least 60% of area is non-vegetated barren (sand, rock,␣
↪soil) or permanent snow and ice with less than 10% vegetation.|
    |Urban and Built-up Lands|10|At least 30% impervious surface area including␣
↪building materials, asphalt, and vehicles.|
    |Unclassified|255|Has not received a map label because of missing inputs.|
    '''

    LC_Type3 = np.array([s.split('|')[1:-1] for s in table.split('\n')[1:-1]]).T


    '''
    First, lets get the codes and names of the LCs used
    '''
    flc_data = lc_data.astype(float)
    flc_data[lc_data == 255] = np.nan
    land_covers_present = np.unique(lc_data[lc_data!=255])
    land_cover_names = LC_Type3[0]
    '''
    For categorical data we want a quantitative colormap

    The core options are:
    https://matplotlib.org/tutorials/colors/colormaps.html

    qcmaps = ['Pastel1', 'Pastel2', 'Paired', 'Accent',
              'Dark2', 'Set1', 'Set2', 'Set3',
               'tab10', 'tab20', 'tab20b', 'tab20c']
    '''

    '''
    Now learn how to plot with categorical labels
    following example in
    https://gist.github.com/jakevdp/8a992f606899ac24b711
    FuncFormatter to put labels
    '''

    ncov = land_covers_present.max()
    # This function formatter will replace integers with target names
    formatter = plt.FuncFormatter(lambda val, loc: land_cover_names[val])
    plt.figure(figsize=(10,10))
    plt.title(f'MODIS LAI Land cover LC_Type3 from MCD12Q1 {year} {country_code}')
    plt.imshow(flc_data,vmax=ncov,vmin=0,\
               cmap=plt.cm.get_cmap(cmap,ncov))
    plt.colorbar(ticks=np.arange(ncov+2).astype(int), \
                 format=formatter)
    return(land_cover_names[land_covers_present])

from geog0111.plot_landcover import plot_land_cover
print(plot_land_cover(lc_data,year,country_code))
```

```
['Grasslands' 'Broadleaf Croplands' 'Savannas'
```

(continues on next page)

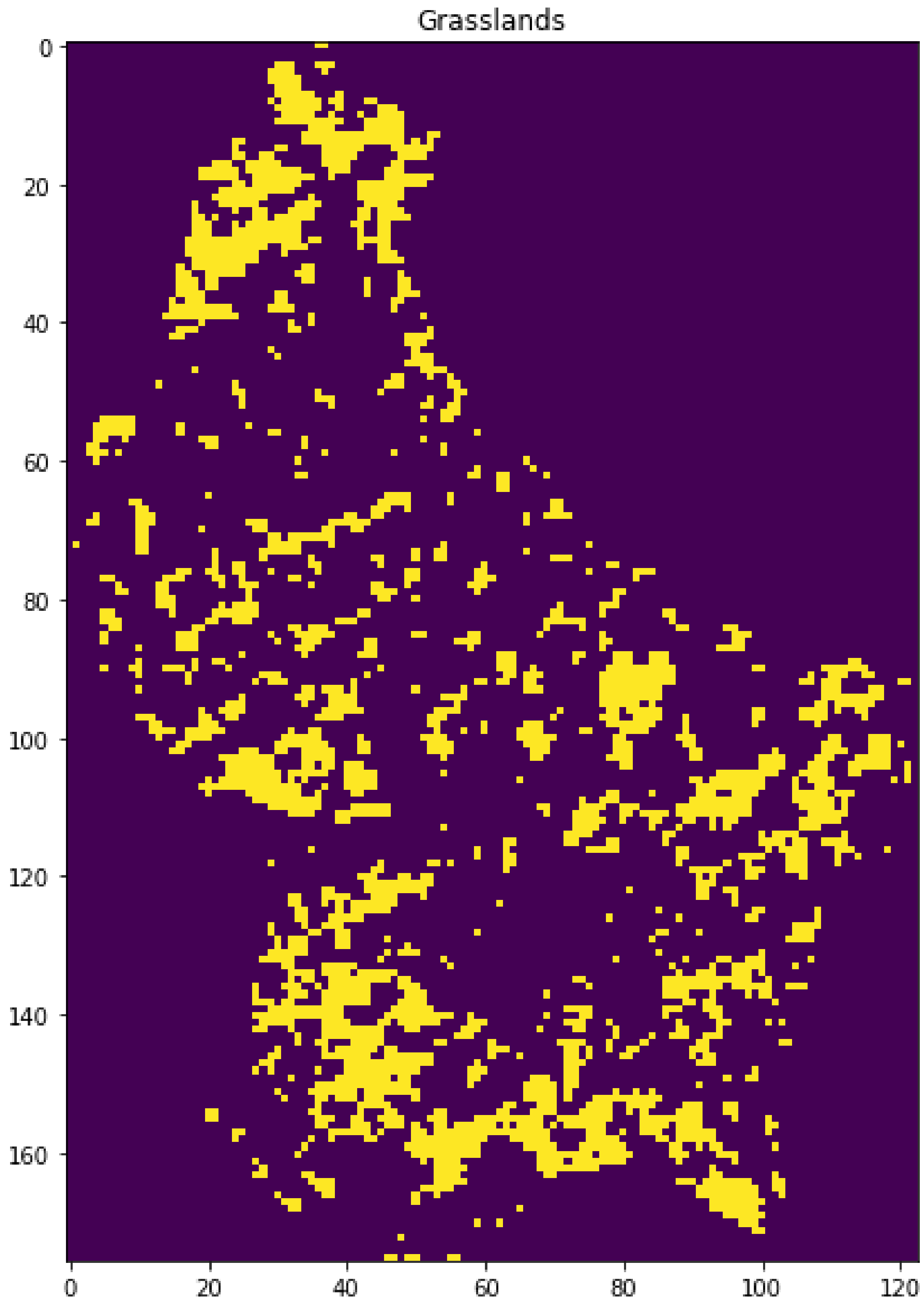
(continued from previous page)

```
'Evergreen Broadleaf Forests' 'Deciduous Broadleaf Forests'
'Evergreen Needleleaf Forests' 'Urban and Built-up Lands']
```


MODIS LAI Land cover LC_Type3 from MCD12Q1 2017 LU
0
20
40
60
80
100
120
140
160
0
20
40
60
80
100
120
Urban and Built-up Lands
Non-Vegetated Lands
Deciduous Needleleaf Forests
Evergreen Needleleaf Forests
Deciduous Broadleaf Forests
Evergreen Broadleaf Forests
Savannas
Broadleaf Croplands
Shrublands
Grasslands
Water Bodies

```
lc = 1

flc_data = lc_data.astype(float)
flc_data[lc_data != lc] = np.nan
plt.figure(figsize=(10,10))
plt.imshow(flc_data.astype(int))
plt.title(LC_Type3[0,lc])
print(LC_Type3[2,lc])
```

```
Dominated by herbaceous annuals (<2m) includ- ing cereal croplands.
```



```
# uncomment to run all LCs
_='''for year in [2016,2017]:
    for country_code in ['BE','DA','EI','FR','GM','IM','LU','NL','SP','SZ','UK']:
```

(continues on next page)

(continued from previous page)

```
        %run geog0111/get_landcover.py $country_code $year'''
```

2.8.18 3.6 Reconciling projections

Table of Contents

3.6.1 Introduction

This section of notes is optional to the course, and the tutor may decide *not* to go through this in class.

That said, the information and examples contained here can be very useful for accessing and processing certain types of geospatial data.

In particular, we deal with obtaining climate data records from ECMWF that we will later use for model fitting. These data come in a netcdf format (commonly used for climate data) with a grid in latitude/longitude. To 'overlay' these data with another dataset (e.g. the MODIS LAI product that we have been using) in a different (equal area) projection, we use the `gdal` function

```
gdal.ReprojectImage(src, dst, src_proj, dst_proj, interp)
```

where:

```
src      : a source dataset that we want to process
dst      : a blank destination dataset that we set up with the
           required (output) data type, shape, and geotransform and projection
src_proj : the source dataset projection wkt
dst_proj : the destination projection wkt
interp   : the required interpolation method, e.g. gdalconst.GRA_Bilinear
```

where wkt stands for well known text and is a projection format string.

Other codes we use are ones we have developed earlier.

In these notes, we will learn:

```
* how to access an ECMWF daily climate dataset (from ERA interim)
* how to reproject the dataset to match another spatial dataset (MODIS LAI)
```

We will then save some datasets that we will use later in the notes. For this reason, it's possile to skip this section, and return to it later.

3.6.1.1 Projections

For various reasons, different geospatial datasets will come in different projections.

Considering for example, satellite-derived data from Low Earth Orbit LEO, the satellite sensor will typically obtain image data in a swath as it passes over the Earth surface. Projected onto the Earth surface, this appears as a strip of data:

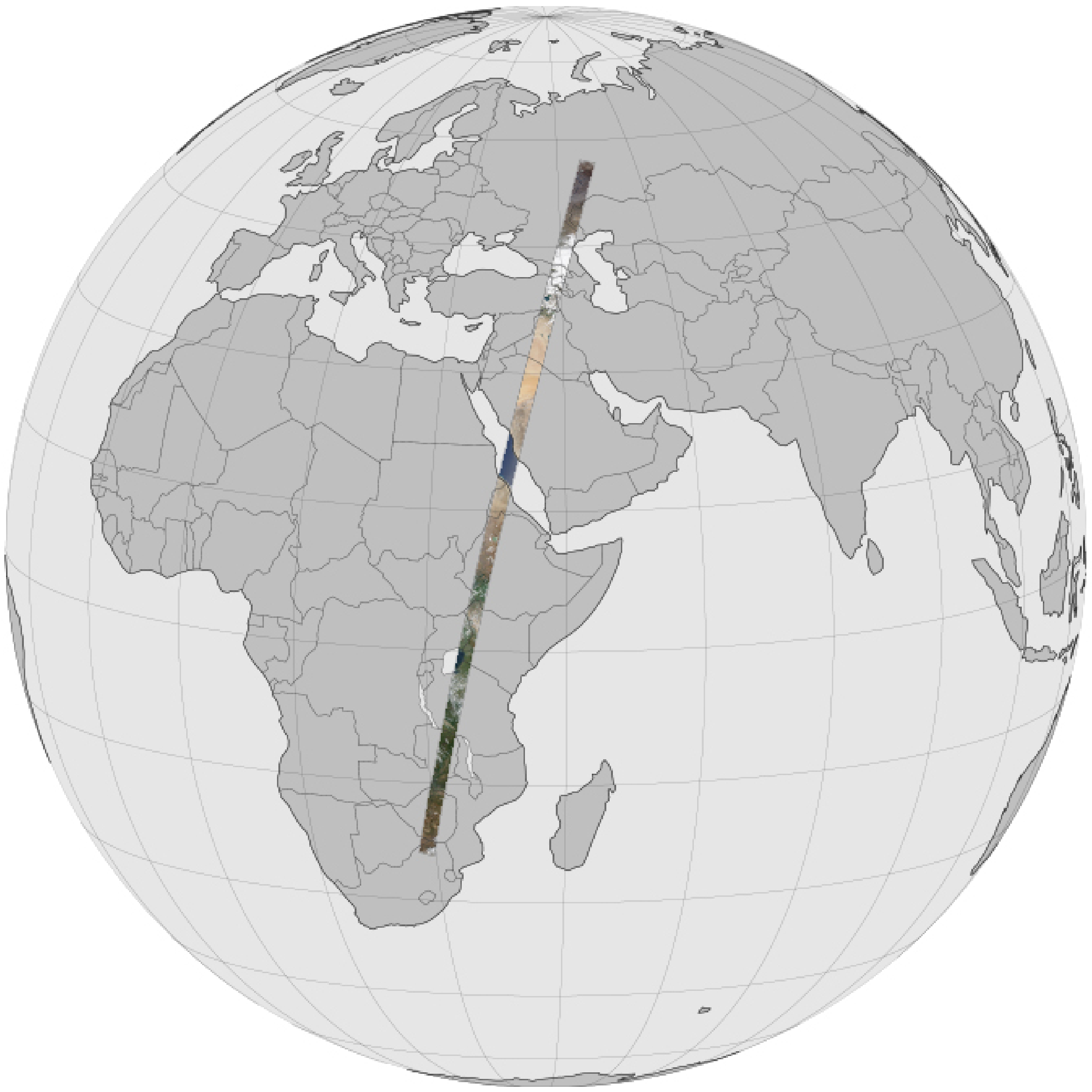


Fig. 12: https://earthobservatory.nasa.gov/Features/LDCMLongSwath

but in the satellite data recording system, the data are stored as a regular array. We call such satellite data ‘swath’ (or ‘swath-like’) data (in the satellite imager coordinate system) and we may obtain data products in anything up to Level 2 in such a form.

These data are often difficult for data scientists to deal with. They generally prefer to have a dataset mapped to a uniform space-time grid, even though this may involve some re-sampling, which can sometimes result in loss of information. The convenience of a uniform space-time grid means that you can. for example, look at dynamic features (information over time).

The properties of the ‘uniform space-time grid’ will depend on user requirements. For some, it is important to have an equal area projection, one where the ‘pixel size’ is consistent throughout the dataset.

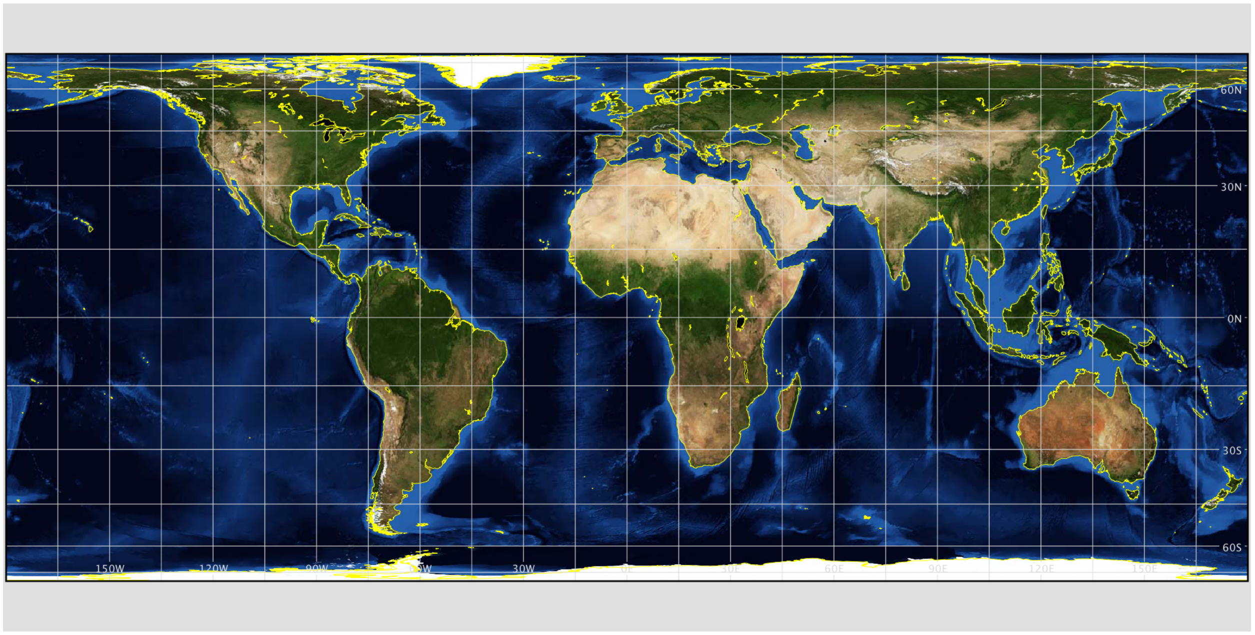


Fig. 13: https://www.giss.nasa.gov/tools/gprojector/hehttps://www.giss.nasa.gov/tools/gprojector/help/projections/CylindricalEqualArea.png

even if this is not convenient for viewing some areas of the Earth (map projections are very political!).

Or other factors may be more important, such as user familiarity with a simple latitude/longitude grid typically used by climate scientists.

For others, a conformal projection (preserving angles, as a cost of distance distortion) may be vital.

We have see that MODIS data products, for example, come described in an equal area sinusoidal grid:

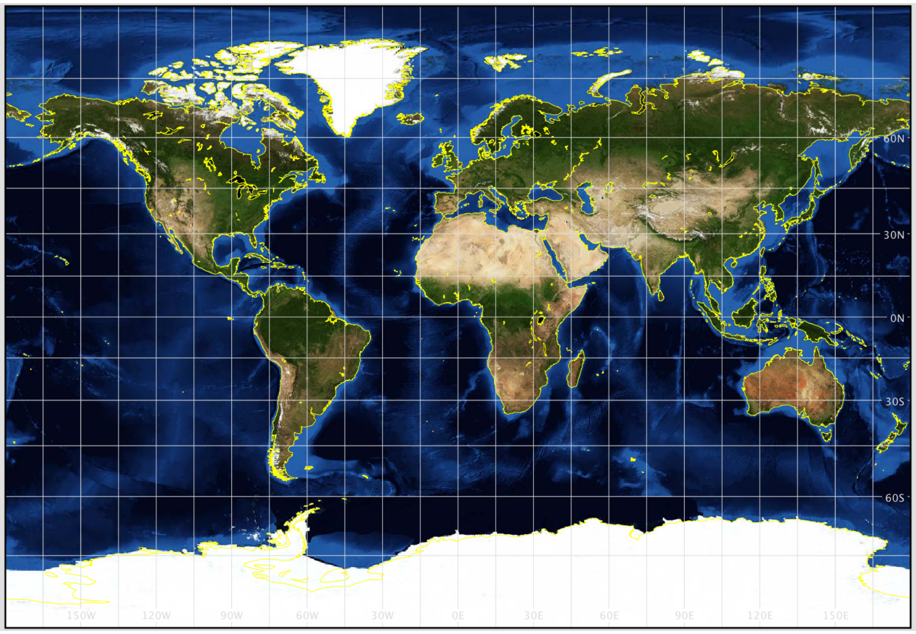


Fig. 14: https://www.giss.nasa.gov/tools/gprojector/help/projections/CylindricalStereographic.png

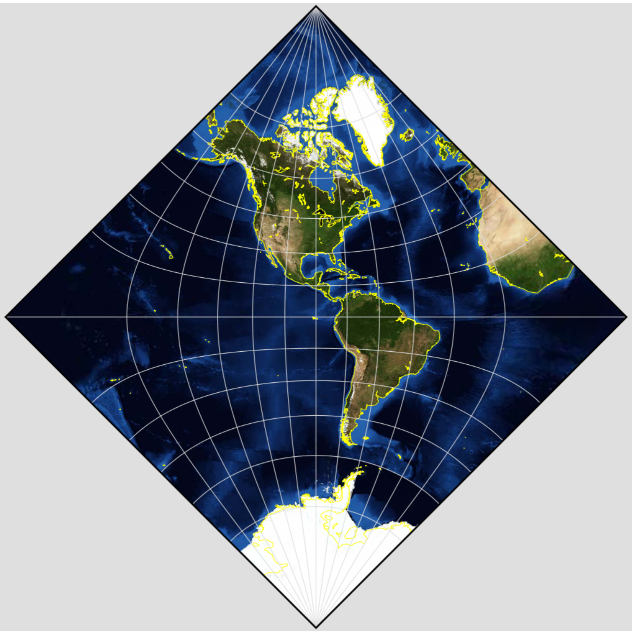
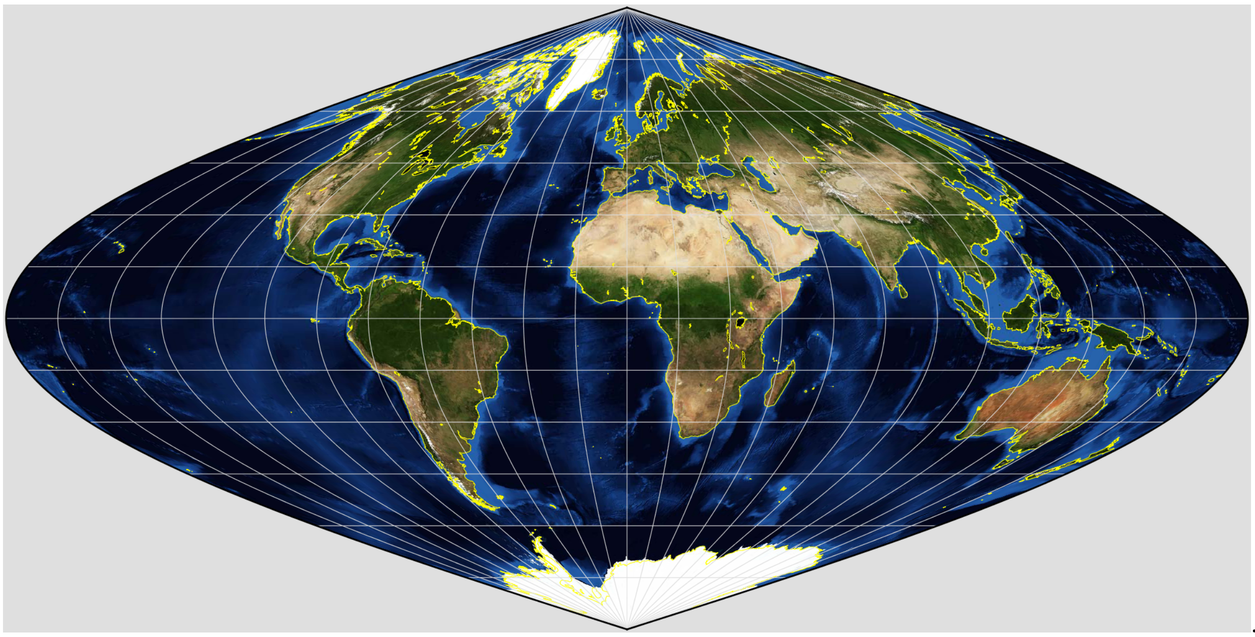


Fig. 15: https://www.giss.nasa.gov/tools/gprojector/help/projections/AdamsHemisphereInASquare.png


.

but the data for high latitudes and longitudes appears very distorted.

We must accept then, that dealing with geospatial data must involve some understanding of projections, as well as practically, how to convert datasets between different projections.

Earth shape

One factor that can make life even more complicated than using just different projections is the use of different assumptions about the Earth shape (e.g. sphere, spheroid, radius variations). Often, the particular assumptions used by a group of users is just a result of history: it is what has 'traditionally' used for that purpose. It can be seen as too bothersome or expensive to change this.

Since we can convert between different projections though, we can also deal with different Earth shape assumptions. We just have to be very clear about what was assumed. If at all possible, the geospatial datasets themselves should contain a full description of the projection and Earth shape assumed, but this is not always the case.

The datasets we will mostly be dealing are in the following projections:

- MODIS Sinusoidal (tested), which assumes a custom spherical Earth of radius 6371007.181 m. In `cartopy` this is given as Sinusoidal.MODIS:

  ```
  # MODIS data products use a Sinusoidal projection of a spherical Earth
  # http://modis-land.gsfc.nasa.gov/GCTP.html
  Sinusoidal.MODIS = Sinusoidal(globe=Globe(ellipse=None,
                                            semimajor_axis=6371007.181,
                                            semiminor_axis=6371007.181))
  ```

  In the MODIS data hdf products, the projection information is stored directly. Extracted as a wkt, this is:

  ```
  [[PROJCS["unnamed",
      GEOGCS["Unknown datum based upon the custom spheroid",
          DATUM["Not_specified_based_on_custom_spheroid",
              SPHEROID["Custom spheroid",6371007.181,0]],
      PRIMEM["Greenwich",0],
      UNIT["degree",0.0174532925199433]],
  PROJECTION["Sinusoidal"],
  PARAMETER["longitude_of_center",0],
  ```

(continues on next page)

(continued from previous page)

```
PARAMETER["false_easting",0],
PARAMETER["false_northing",0],
UNIT["metre",1,AUTHORITY["EPSG","9001"]]]
```

According to SR-ORG, the MODIS projection uses a spherical projection ellipsoid but a WGS84 datum ellipsoid. This is not quite the same as the definition in the wkt above.

It is also defined by SR-ORG with the EPSG code 6974 for software that can use `semi_major` and `semi_minor` projection definitions.

Some software may use the simpler 6965 definition (or the older 6842).

The MODIS projection 6974 is given as:

```
PROJCS["MODIS Sinusoidal",
    GEOGCS["WGS 84",
        DATUM["WGS_1984",
            SPHEROID["WGS 84",6378137,298.257223563,
                AUTHORITY["EPSG","7030"]],
        AUTHORITY["EPSG","6326"]],
    PRIMEM["Greenwich",0,
        AUTHORITY["EPSG","8901"]],
    UNIT["degree",0.01745329251994328,
        AUTHORITY["EPSG","9122"]],
    AUTHORITY["EPSG","4326"]],
PROJECTION["Sinusoidal"],
PARAMETER["false_easting",0.0],
PARAMETER["false_northing",0.0],
PARAMETER["central_meridian",0.0],
PARAMETER["semi_major",6371007.181],
PARAMETER["semi_minor",6371007.181],
UNIT["m",1.0],
AUTHORITY["SR-ORG","6974"]]
```

None of these codes are defined in `gdal` (see files in $GDAL_DATA/*.wkt for details), so to use them, we have to take the file from SR-ORG.

For the datasets we are using, it makes no real difference whether the projection information from the file is used instead of MODIS projection 6974, so we will use that from the file. For other areas and especially for any higher spatial resolution datasets, it is worth investigating which is more appropriate.

- ECMWF netcdf format (derived from GRIB) ERA Interim climate datasets (1979-Present). These are geographic coordinates (latitude/longitude) in a custom spheroid with a radius 6371200 m.

This information can be obtained from any example of a GRIB file, as we shall see below. As a wkt, this is:

```
['GEOGCS["Coordinate System imported from GRIB file",
DATUM["unknown",SPHEROID["Sphere",6371200,0]],
PRIMEM["Greenwich",0],UNIT["degree",0.0174532925199433]]']
```

- A more common spheroid to use is WGS84, although even in that case there are multiple 'realisations' available (used mainly by the DoD). Users should generally implement that given in EPSG code 4326 used by the GPS system, for example.

```
[GEOGCS["WGS 84",
    DATUM["WGS_1984",
        SPHEROID["WGS 84",6378137,298.257223563,
            AUTHORITY["EPSG","7030"]],
```

(continues on next page)

(continued from previous page)

```
        AUTHORITY["EPSG","6326"]],
    PRIMEM["Greenwich",0,
        AUTHORITY["EPSG","8901"]],
    UNIT["degree",0.0174532925199433,
        AUTHORITY["EPSG","9122"]],
    AUTHORITY["EPSG","4326"]]]
```

3.6.1.2 Changing Projections

We can conveniently use the Python `` `cartopy <https://scitools.org.uk/cartopy/docs/v0.16/>`__ `` package to explore projections.

We download an image taken from the satellite sensor (SEVIRI):

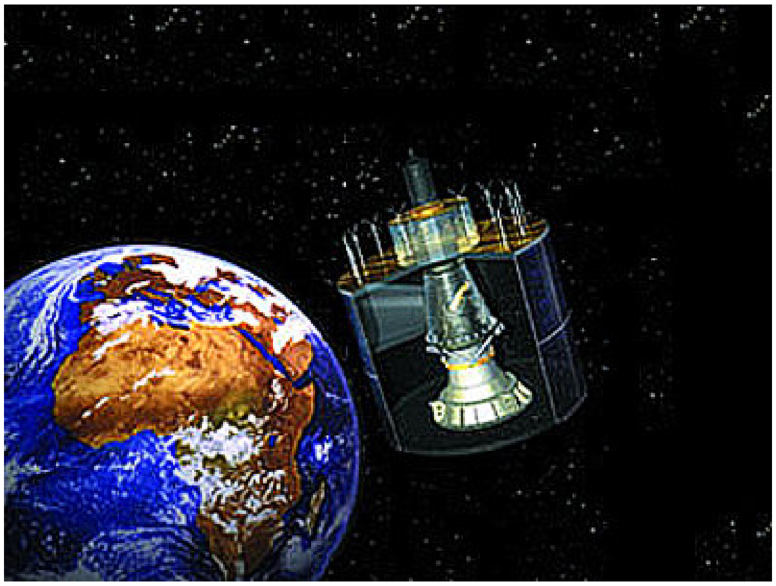


Fig. 16: http://www.esa.int/spaceinimages/Images/2005/12/Artist_s_view_of_SEVIRI_in_orbit

The sensor builds up images of the Earth disc from geostationarty orbit, actioned by the platform spin.

In the code below, we plot the dataset in the 'earth disk' (Orthographic) projection, then re-map it to the equal area Sinusoidal projection.

```
try:
    from urllib2 import urlopen
except ImportError:
    from urllib.request import urlopen
from io import BytesIO
%matplotlib inline

import cartopy.crs as ccrs
import matplotlib.pyplot as plt
are_you_sure = False

'''
=====================================================
Don't run this cell in class as it will take too long!
```

(continues on next page)

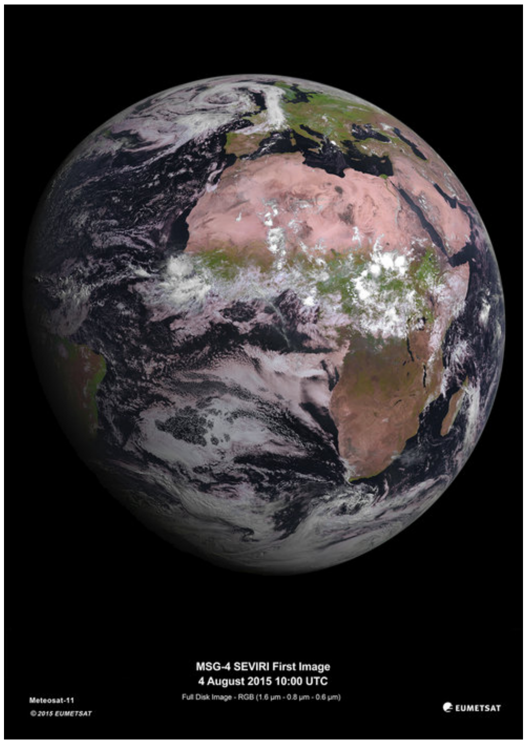


Fig. 17: http://www.esa.int/spaceinimages/Images/2015/08/MSG-4_Europe_s_latest_weather_satellite_delivers_first_image

(continued from previous page)

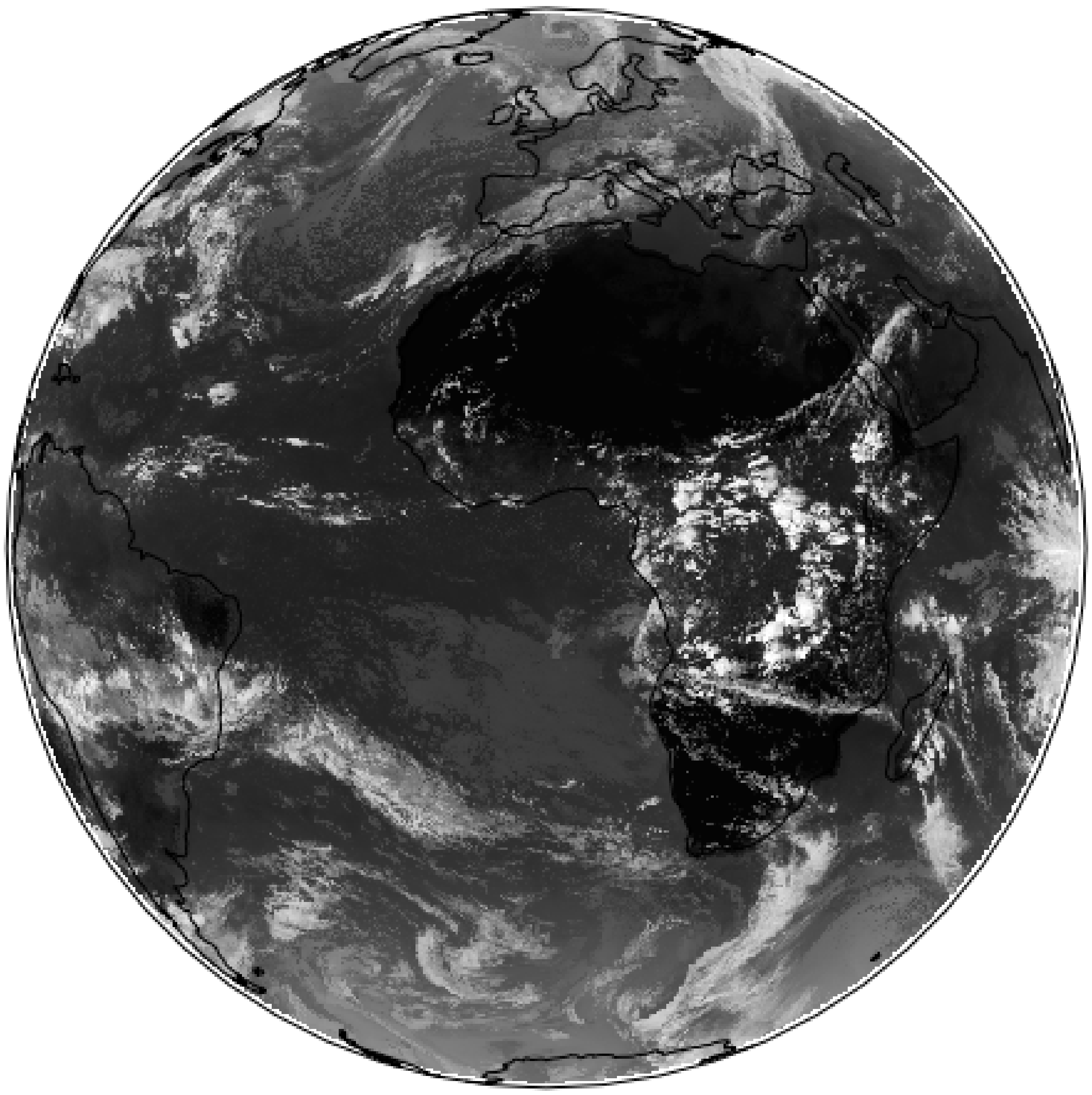
```
                 Use it for homework
                 set are_you_sure = True
=====================================================
'''

'''
from https://scitools.org.uk/cartopy/docs/v0.16/\
            gallery/geostationary.html#sphx-glr-gallery-geostationary-py
'''
def geos_image():
    """
    Return a specific SEVIRI image by retrieving it from a github gist URL.

    Returns
    -------
    img : numpy array
        The pixels of the image in a numpy array.
    img_proj : cartopy CRS
        The rectangular coordinate system of the image.
    img_extent : tuple of floats
        The extent of the image ``(x0, y0, x1, y1)`` referenced in
        the ``img_proj`` coordinate system.
    origin : str
        The origin of the image to be passed through to matplotlib's imshow.
```

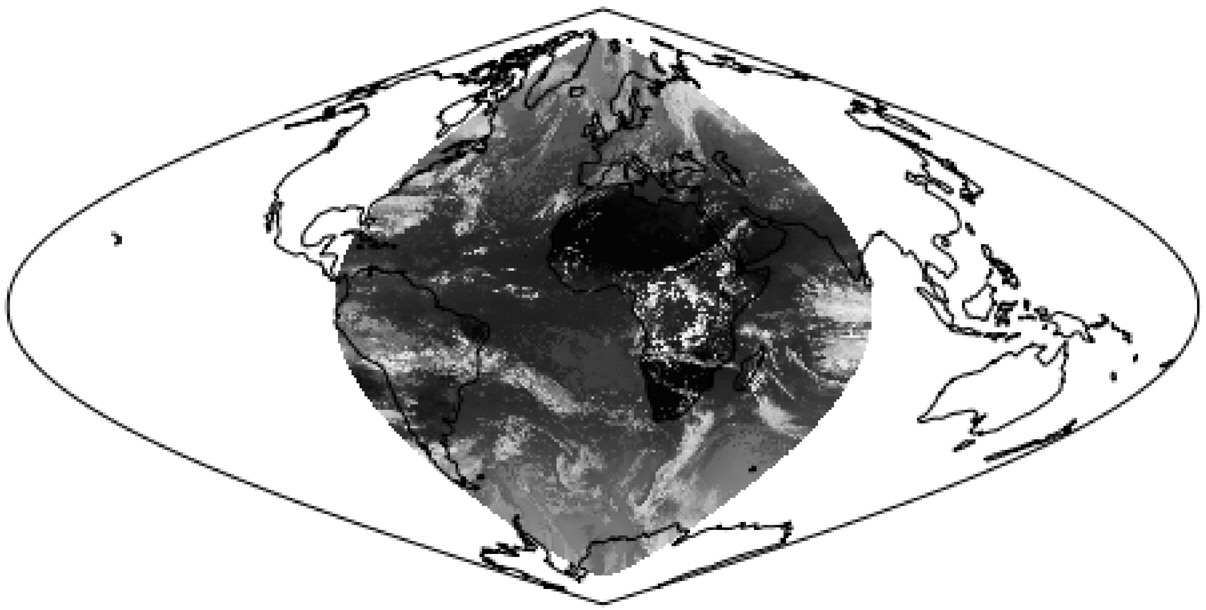
(continues on next page)

(continued from previous page)

```
    """
    url = ('https://gist.github.com/pelson/5871263/raw/'
           'EIDA50_201211061300_clip2.png')
    img_handle = BytesIO(urlopen(url).read())
    img = plt.imread(img_handle)
    img_proj = ccrs.Geostationary(satellite_height=35786000)
    img_extent = [-5500000, 5500000, -5500000, 5500000]
    return img, img_proj, img_extent, 'upper'

if are_you_sure:
    print('Retrieving image...')
    img, crs, extent, origin = geos_image()

    fig = plt.figure(figsize=(8,8))
    ax = fig.add_subplot(1, 1, 1,projection=\
                         ccrs.Orthographic(central_longitude=0.0, central_latitude=0.
↪0))
    ax.coastlines()
    ax.set_global()
    ax.imshow(img, transform=crs, extent=extent, origin=origin, cmap='gray')

    fig = plt.figure(figsize=(8,8))
    ax = fig.add_subplot(1, 1, 1, projection=\
                         ccrs.Sinusoidal(central_longitude=0.0, \
                            false_easting=0.0, false_northing=0.0))
    ax.coastlines()
    ax.set_global()
    print('Projecting and plotting image (this may take a while)...')
    ax.imshow(img, transform=crs, extent=extent, origin=origin, cmap='gray')
```

```
Retrieving image...
Projecting and plotting image (this may take a while)...
```





The full list of `` `cartopy `` projections <https://scitools.org.uk/cartopy/docs/v0.16/crs/projections.html>'__ is quite entensive.

Exercise 3.6.1 Extra Homework

- Explore some different types of projection using `cartopy` and make a note of their features.
- Read up (follow the links in the text above) on projections.

```
#do exercise here
```

3.6.2 Requirements

We will need to:

- make sure we have the MODIS LAI dataset locally
- read them in for a given country.
- register with ecmwf, install ecmwfapi
- get the temperature datasset from ECMWF for 2006 and 2017 for Europe
- get the country borders shapefile

Set up the conditions

```
# required general imports
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
import sys
import os
from pathlib import Path
import gdal
from datetime import datetime, timedelta
import cartopy.crs as ccrs
```

```
'''
Set the country code and year to be used here
'''
country_code = 'UK'
year = 2017
shpfile = "data/TM_WORLD_BORDERS-0.3.shp"
```

3.6.2.1 Run the pre-requisite scripts

Make sure you register with ECMWF * register with ECMWF and install the API

```
Follow the [ECMWF instructions](https://confluence.ecmwf.int/display/WEBAPI/
↪Access+ECMWF+Public+Datasets)
```

Sort data prerequisitions * Run the codes in the prerequisites section

```
OR
```

- Run the [prerequisites script]:

```
# install ecmwf api -- do this once only
ECMWF = 'https://software.ecmwf.int/wiki/download/attachments/56664858/ecmwf-api-
↪client-python.tgz'
try:
    from ecmwfapi import ECMWFDataServer
except:
    try:
        !pip install $ECMWF
    except:
        # on Unix/Linux
        !pip install --user $ECMWF
```

```
# just make sure the pre-requisites are run
%run geog0111/Chapter3_6A_prerequisites.py $country_code $year
```

```
['geog0111/Chapter3_6A_prerequisites.py', 'UK', '2017'] 2017 UK
Looking for match to sample  2017-01-01 00:00:00
Looking for match to sample  2017-02-10 00:00:00
Looking for match to sample  2017-03-22 00:00:00
Looking for match to sample  2017-05-01 00:00:00
Looking for match to sample  2017-06-10 00:00:00
Looking for match to sample  2017-07-20 00:00:00
Looking for match to sample  2017-08-29 00:00:00
Looking for match to sample  2017-10-08 00:00:00
Looking for match to sample  2017-11-17 00:00:00
Looking for match to sample  2017-12-27 00:00:00
18.45137418797783
0.19862203366558234
(2624, 1396, 92) (2624, 1396, 92)
interpolating ...
(2624, 1396, 92)
saving ...
europe_data_2016_2017.nc exists
GEOGCS["Coordinate System imported from GRIB file",DATUM["unknown",SPHEROID["Sphere",
↪6371200,0]],PRIMEM["Greenwich",0],UNIT["degree",0.0174532925199433]]
```

(continues on next page)

(continued from previous page)

```
Refreshing nc file europe_data_2016_2017.nc
data/europe_data_2016.nc
data/europe_data_2017.nc
```

```
# read in the LAI data for given country code
tiles = []
for h in [17, 18]:
    for v in [3, 4]:
        tiles.append(f"h{h:02d}v{v:02d}")

fname = f'lai_data_{year}_{country_code}.npz'
ofile = Path('data')/fname
try:
    # read data from npz file
    lai = np.load(ofile)
    print(lai['lai'].shape)
except:
    print(f"{ofile} doesn't exist: sort the pre-requisites")
```

```
(2624, 1396, 92)
```

```
import numpy as np
# a quick look at some stats to see if there are data there
# and they are sensible
lai = np.load(ofile)
print(np.array(lai['lai'][1000,700]),\
      np.array(lai['weights'][1000,700]))
# does it have the interpolated value?
if 'interpolated_lai' in list(lai.keys()):
    print(np.array(lai['interpolated_lai'][1000,700]))
```

```
[1.1 1.1 1.2 0.9 0.9 1.2 1.  0.8 1.4 0.3 0.  0.5 0.3 0.9 0.6 1.  0.7 0.5
 0.7 0.6 1.3 0.7 0.1 1.  1.  0.6 1.1 0.5 1.1 1.  1.1 1.1 1.1 0.  0.6 1.4
 1.3 1.6 1.7 1.7 1.1 0.3 1.3 1.7 1.5 1.2 0.5 0.6 1.6 3.1 0.3 2.  1.6 0.5
 2.6 2.5 0.4 0.4 0.4 0.4 0.4 1.1 1.7 0.5 1.5 1.4 0.1 1.4 1.  1.  0.3 1.1
 0.2 1.1 0.1 0.7 2.6 1.7 2.  1.4 1.4 0.3 0.4 0.7 1.1 1.1 0.8 0.8 0.9 1.2
 1.2 1.2] [0.38196601 0.38196601 0.38196601 0.38196601 0.38196601 0.38196601
 0.38196601 0.38196601 0.38196601 0.38196601 0.38196601 1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         0.23606798 1.         1.
 1.         1.         1.         1.         0.23606798 0.23606798
 1.         1.         1.         1.         0.23606798 1.
 1.         1.         0.23606798 1.         1.         0.23606798
 1.         1.         0.23606798 0.23606798 0.23606798 0.23606798
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 0.23606798 1.         1.         0.38196601 0.38196601 0.38196601
 0.38196601 0.38196601 0.38196601 0.38196601 0.38196601 0.38196601
 0.38196601 0.38196601 0.38196601 0.38196601 0.38196601 0.38196601
 0.38196601 0.38196601]
[1.08683704 1.07935938 1.06092468 1.03161555 0.99133678 0.93493706
 0.86317607 0.78136836 0.70352293 0.64487202 0.61266704 0.60678279
 0.61883651 0.63991235 0.66383767 0.68530728 0.7020839  0.71506691
```

(continues on next page)

(continued from previous page)

```
 0.72433462 0.73247513 0.74002342 0.7487489  0.76081558 0.77753903
 0.79985155 0.82674875 0.8580399  0.89089931 0.92313233 0.95320821
 0.98065863 1.00873477 1.04290792 1.08866497 1.14943392 1.22080857
 1.2929335  1.35676154 1.4021202  1.42632177 1.43105239 1.41880135
 1.40112126 1.38702625 1.38535581 1.40496862 1.45132144 1.52190587
 1.6097794  1.70184005 1.78498301 1.84800873 1.88579434 1.89137779
 1.85216521 1.7573269  1.60194669 1.4068558  1.2226864  1.09378123
 1.03387865 1.01734516 1.02167784 1.02682062 1.02221966 1.00590369
 0.98109385 0.95002526 0.91607184 0.8832294  0.85542066 0.83730438
 0.83548611 0.85858254 0.90888448 0.98602178 1.07794995 1.15678679
 1.19786983 1.18422965 1.12426637 1.04138037 0.9671476  0.91852027
 0.90214004 0.90955621 0.93302657 0.96662729 1.00304032 1.03742083
 1.06445991 1.08219511]
```

3.6.3 Reconcile the datasets

In this section, we will use `gdal` to transform two datasets into the same coordinate system.

To do this, we identify one dataset with the projection and geographic extent that we want for our data (a MODIS sub-dataset here, the 'exemplar').

We then download a climate dataset in a latitude/longitude grid (netcdf format) and transform this to be consistent with the MODIS dataset.

3.6.3.1 load an exemplar dataset

Since we want to match up datasets, we need to produce an example of the dataset we want to match up to.

We save the exemplar as a GeoTiff format file here.

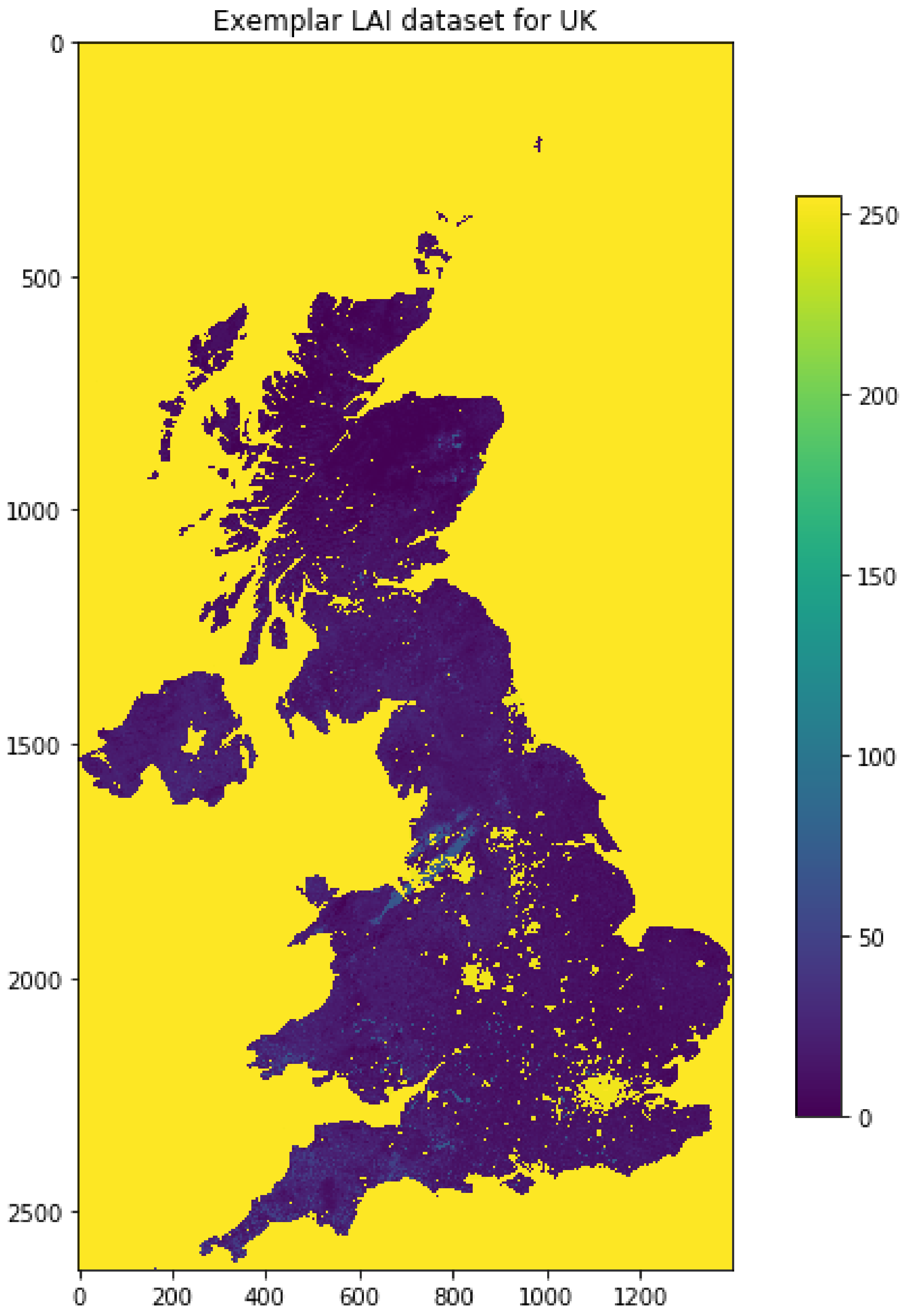
```
from osgeo import gdal, gdalconst,osr
import numpy as np
from geog0111.process_timeseries import mosaic_and_clip

# set to True if you want to override
# the MODIS projection (see above)
use_6974 = False

'''
https://stackoverflow.com/questions/10454316/
how-to-project-and-resample-a-grid-to-match-another-grid-with-gdal-python
'''

# first get an exemplar LAI file, clipped to
# the required limits. We will use this to match
# the t2 dataset to
match_filename = mosaic_and_clip(tiles,1,year,ofolder='tmp',\
                    country_code=country_code,shpfile=shpfile,frmat='GTiff')

print(match_filename)

'''
Now get the projection, geotransform and dataset
size that we want to match to
'''
```

(continues on next page)

(continued from previous page)

```
match_ds = gdal.Open(match_filename, gdalconst.GA_ReadOnly)
match_proj = match_ds.GetProjection()
match_geotrans = match_ds.GetGeoTransform()
wide = match_ds.RasterXSize
high = match_ds.RasterYSize

print('\nProjection from file:')
print(match_proj,'\n')

'''
set Projection 6974 from SR-OR
by setting use_6974 = True
'''
if use_6974:
    print('\nProjection 6974 from SR-ORG:')
    modis_wkt = 'data/modis_6974.wkt'
    match_proj = open(modis_wkt,'r').readline()
    match_ds.SetProjection(match_proj)
    print(match_proj,'\n')

'''
Visualise
'''
plt.figure(figsize=(10,10))
plt.title(f'Exemplar LAI dataset for {country_code}')
plt.imshow(match_ds.ReadAsArray())
plt.colorbar(shrink=0.75)
# close the file -- we dont need it any more
del match_ds
```

```
tmp/Lai_500m_2017_001_UK.tif

Projection from file:
PROJCS["unnamed",GEOGCS["Unknown datum based upon the custom spheroid",DATUM["Not_
↪specified_based_on_custom_spheroid",SPHEROID["Custom spheroid",6371007.181,0]],
↪PRIMEM["Greenwich",0],UNIT["degree",0.0174532925199433]],PROJECTION["Sinusoidal"],
↪PARAMETER["longitude_of_center",0],PARAMETER["false_easting",0],PARAMETER["false_
↪northing",0],UNIT["metre",1,AUTHORITY["EPSG","9001"]]]
```


Exemplar LAI dataset for UK
0
500
1000
1500
2000
2500
0
200
400
600
800
1000
1200
250
200
150
100
50
0

3.6.3.2 get information from source file

Now, we pull the information we need from the source file (the netcdf format t2 dataset).

We need to know:

- the data type
- the number of bands (time samples in this case)
- the geotransform of the dataset (the fact that it's 0.25 degree resolution over Europe)

and access these from the source dataset.

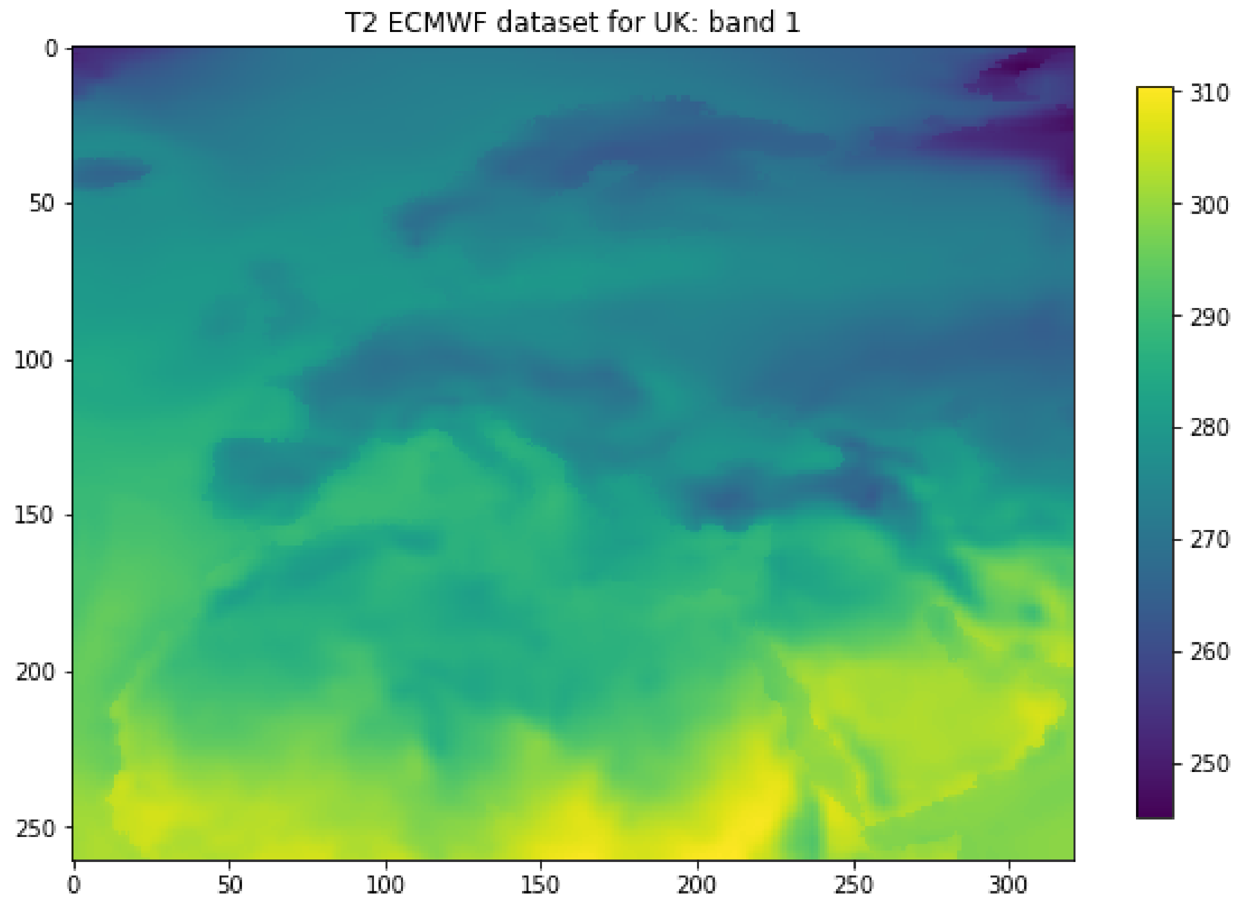
```
from osgeo import gdal, gdalconst,osr
import numpy as np

# set up conditions
src_filename = f'data/europe_data_{year}.nc'
'''
access information from source
'''
src_dataname = 'NETCDF:"'+src_filename+'":t2m'
src      = gdal.Open(src_filename, gdalconst.GA_ReadOnly)

'''
Get geotrans, data type and number of bands
from source dataset
'''
band1 = src.GetRasterBand(1)
src_proj = src.GetProjection()
src_geotrans = src.GetGeoTransform()
nbands = src.RasterCount
src_format = band1.DataType
nx = band1.XSize
ny = band1.YSize

print('Information found')
print('GeoTransform:    ',src_geotrans)
print('Projection:      ',src_proj)
print('number of bands:',nbands)
print('format:          ',src_format)
print('nx,ny:           ',nx,ny)

# read data
t2m = band1.ReadAsArray()
plt.figure(figsize=(10,10))
ax = plt.subplot ( 1, 1, 1)
ax.set_title(f'T2 ECMWF dataset for {country_code}: band 1')

im = plt.imshow(t2m)
_ = plt.colorbar(im,shrink=0.6)
```

```
Information found
GeoTransform:     (-20.125, 0.25, 0.0, 75.125, 0.0, -0.25)
Projection:      GEOGCS["Coordinate System imported from GRIB file",DATUM["unknown",
↪SPHEROID["Sphere",6371200,0]],PRIMEM["Greenwich",0],UNIT["degree",0.
↪0174532925199433]]
number of bands: 365
```

(continues on next page)

(continued from previous page)

```
format:             6
nx,ny:              321 261
```



3.6.3.4 reprojection

Now, set up a blank gdal dataset (in memory) with the size, data type, projection etc. that we want, the reproject the temperature dataset into this.

The processing may take some time if the LAI dataset is large (e.g. France).

The result will be of the same size, projection etc as the cropped LAI dataset.

```
dst_filename = src_filename.replace('.nc',f'_{country_code}.tif')
force = False


if (not Path(dst_filename).exists()) or force:

    dst = gdal.GetDriverByName('MEM').Create('', wide, high, nbands, src_format)

    dst.SetGeoTransform( match_geotrans )
    dst.SetProjection( match_proj)
```

(continues on next page)

(continued from previous page)

```
    print('Information found')
    print('wide:       ',wide)
    print('high:       ',high)
    print('geotrans:   ',match_geotrans)
    print('projection:',match_proj)

    # Do the work: reproject the dataset
    # This will take a few minutes, depending on dataset size
    _ = gdal.ReprojectImage(src, dst, src_proj, match_proj, gdalconst.GRA_Bilinear)
```

```
xOrigin = match_geotrans[0]
yOrigin = match_geotrans[3]
pixelWidth = match_geotrans[1]
pixelHeight = match_geotrans[5]

extent = (xOrigin,xOrigin+pixelWidth*wide,\
         yOrigin+pixelHeight*(high),yOrigin+pixelHeight)

print(extent)

if (not Path(dst_filename).exists()) or force:



    '''
    Visualise: takes some time to plot
               due to reprojections
    '''
    t2m = dst.GetRasterBand(1).ReadAsArray()
    match_ds = gdal.Open(match_filename, gdalconst.GA_ReadOnly).ReadAsArray()

    # visualise
    plt.figure(figsize=(15,10))
    ax = plt.subplot ( 1, 2, 1 ,projection=ccrs.Sinusoidal.MODIS)
    ax.coastlines('10m')
    ax.set_title(f'T2m ECMWF dataset for {country_code}: band 1')
    im = ax.imshow(t2m[::-1],extent=extent)
    plt.colorbar(im,shrink=0.75)


    ax = plt.subplot ( 1, 2, 2 ,projection=ccrs.Sinusoidal.MODIS)
    ax.coastlines('10m')
    ax.set_title(f'MODIS LAI {country_code}')
    im = plt.imshow(match_ds,extent=extent)
    _ = plt.colorbar(im,shrink=0.75)
```

```
(-528121.3116353625, 118541.0173548233, 5549929.5167459585, 6765137.823590214)
```

3.6.3.5 crop

Finally, we crop the temperature dataset using `gdal.Warp()` and save it to a (GeoTiff) file:

```
 # Output / destination
dst_filename = src_filename.replace('.nc',f'_{country_code}.tif')
```

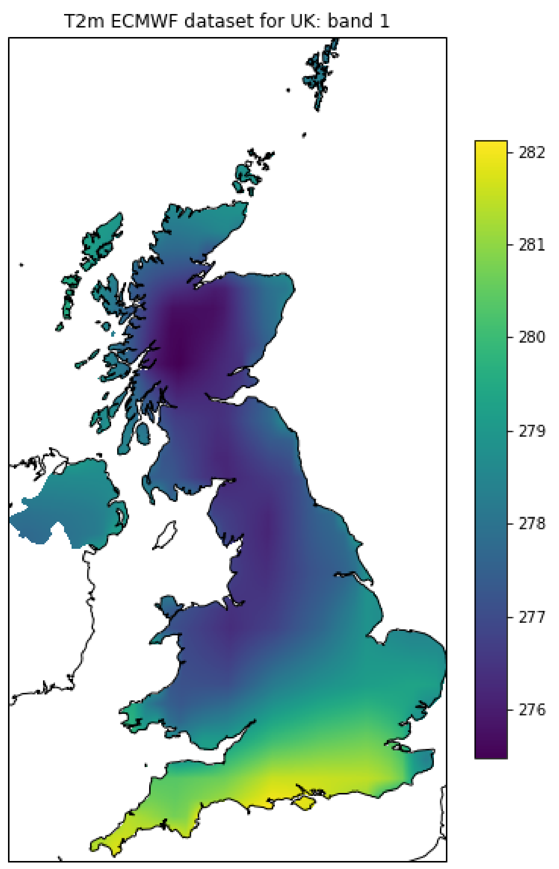
(continues on next page)

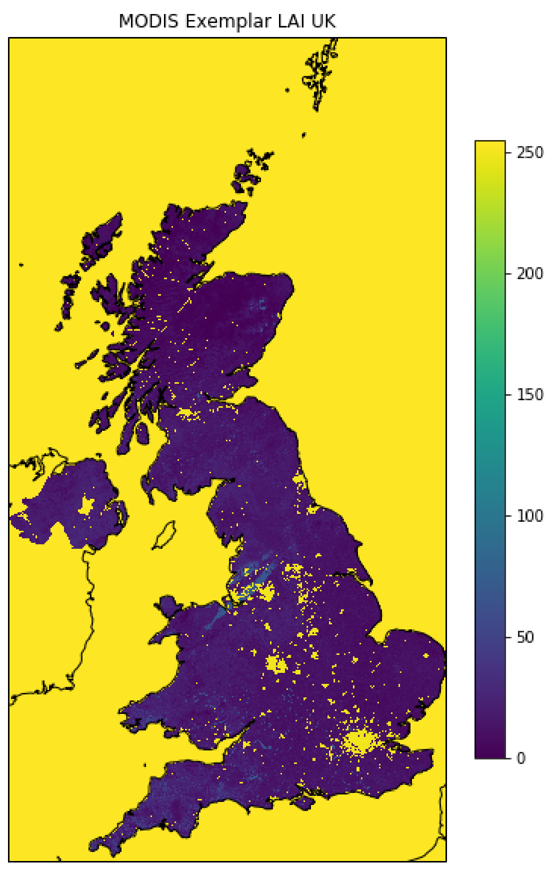
(continued from previous page)

```
force = False


if (not Path(dst_filename).exists()) or force:
    '''
    Only run this if file doesnt exist
    '''
    frmat = 'GTiff'
    g = gdal.Warp(dst_filename,
                  dst,
                  format=frmat,
                  dstNodata=-300,
                  cutlineDSName=shpfile,
                  cutlineWhere=f"FIPS='{country_code:s}'",
                  cropToCutline=True)
    del dst # Flush
    del g
```

```
# visualise
print(dst_filename)
t2m = gdal.Open(dst_filename, gdalconst.GA_ReadOnly)
t2m = t2m.GetRasterBand(1).ReadAsArray()
t2m[t2m==-300] = np.nan
match_ds = gdal.Open(match_filename, gdalconst.GA_ReadOnly).ReadAsArray()

# visualise
plt.figure(figsize=(15,10))
ax = plt.subplot ( 1, 2, 1 ,projection=ccrs.Sinusoidal.MODIS)
ax.coastlines('10m')
ax.set_title(f'T2m ECMWF dataset for {country_code}: band 1')
im = ax.imshow(t2m[::-1],extent=extent)
plt.colorbar(im,shrink=0.75)

ax = plt.subplot ( 1, 2, 2 ,projection=ccrs.Sinusoidal.MODIS)
ax.coastlines('10m')
ax.set_title(f'MODIS Exemplar LAI {country_code}')
im = plt.imshow(match_ds,extent=extent)
_ = plt.colorbar(im,shrink=0.75)
```

```
data/europe_data_2017_UK.tif
```





Now let's look at the time information in the metadata:

```
meta = gdal.Open(src_filename).GetMetadata()

print(meta['time#units'])
```

```
hours since 1900-01-01 00:00:00.0
```

The time information is in hours since `1900-01-01 00:00:00.0`. This is not such a convenient unit for plotting, so we can use `datetime` to fix that:

```
timer = meta['NETCDF_DIM_time_VALUES']
print(timer[:100])
```

```
{1025628,1025652,1025676,1025700,1025724,1025748,1025772,1025796,1025820,1025844,
↪1025868,1025892,102
```

```
# split the string into integers
timer = [int(i) for i in meta['NETCDF_DIM_time_VALUES'][1:-1].split(',')]

print (timer[:20])
```

```
[1025628, 1025652, 1025676, 1025700, 1025724, 1025748, 1025772, 1025796, 1025820,␣
↪1025844, 1025868, 1025892, 1025916, 1025940, 1025964, 1025988, 1026012, 1026036,␣
↪1026060, 1026084]
```

```
# split the string into integers
# convert to days
timer = [float(i)/24. for i in meta['NETCDF_DIM_time_VALUES'][1:-1].split(',')]

print (timer[:20])
```

```
[42734.5, 42735.5, 42736.5, 42737.5, 42738.5, 42739.5, 42740.5, 42741.5, 42742.5,␣
↪42743.5, 42744.5, 42745.5, 42746.5, 42747.5, 42748.5, 42749.5, 42750.5, 42751.5,␣
↪42752.5, 42753.5]
```

```
from datetime import datetime,timedelta

# add base date
# split the string into integers
# convert to days
timer = [(datetime(1900,1,1) + timedelta(days=float(i)/24.)) \
         for i in meta['NETCDF_DIM_time_VALUES'][1:-1].split(',')]

print (timer[:20])
```

```
[datetime.datetime(2017, 1, 1, 12, 0), datetime.datetime(2017, 1, 2, 12, 0), datetime.
↪datetime(2017, 1, 3, 12, 0), datetime.datetime(2017, 1, 4, 12, 0), datetime.
↪datetime(2017, 1, 5, 12, 0), datetime.datetime(2017, 1, 6, 12, 0), datetime.
↪datetime(2017, 1, 7, 12, 0), datetime.datetime(2017, 1, 8, 12, 0), datetime.
↪datetime(2017, 1, 9, 12, 0), datetime.datetime(2017, 1, 10, 12, 0), datetime.
↪datetime(2017, 1, 11, 12, 0), datetime.datetime(2017, 1, 12, 12, 0), datetime.
↪datetime(2017, 1, 13, 12, 0), datetime.datetime(2017, 1, 14, 12, 0), datetime.
↪datetime(2017, 1, 15, 12, 0), datetime.datetime(2017, 1, 16, 12, 0), datetime.
↪datetime(2017, 1, 17, 12, 0), datetime.datetime(2017, 1, 18, 12, 0), datetime.
↪datetime(2017, 1, 19, 12, 0), datetime.datetime(2017, 1, 20, 12, 0)]
```

3.6.3.6 Putting this together

We can now put these codes together to make a function `match_netcdf_to_data()`:

```
from osgeo import gdal, gdalconst,osr
import numpy as np
from geog0111.process_timeseries import mosaic_and_clip
from datetime import datetime

def match_netcdf_to_data(src_filename,match_filename,dst_filename,year,\
                         country_code=None,shpfile=None,force=False,\
                         nodata=-300,frmat='GTiff',verbose=False):

    '''
    see :
    https://stackoverflow.com/questions/10454316/
    how-to-project-and-resample-a-grid-to-match-another-grid-with-gdal-python
    '''

    '''
    Get the projection, geotransform and dataset
    size that we want to match to
    '''
    if verbose: print(f'getting info from match file {match_filename}')
```

(continues on next page)

(continued from previous page)

```
    match_ds = gdal.Open(match_filename, gdalconst.GA_ReadOnly)

    match_proj = match_ds.GetProjection()
    match_geotrans = match_ds.GetGeoTransform()
    wide = match_ds.RasterXSize
    high = match_ds.RasterYSize
    # close the file -- we dont need it any more
    del match_ds

    '''
    access information from source
    '''
    if verbose: print(f'getting info from source netcdf file {src_filename}')
    try:
        src_dataname = 'NETCDF:"'+src_filename+'":t2m'
        src = gdal.Open(src_dataname, gdalconst.GA_ReadOnly)
    except:
        if verbose: print('failed')
        return(None)

    # get meta data
    meta = gdal.Open(src_filename, gdalconst.GA_ReadOnly).GetMetadata()

    extent = [match_geotrans[0],match_geotrans[0]+match_geotrans[1]*wide,\
              match_geotrans[3]+match_geotrans[5]*high,match_geotrans[3]]
    # get time info
    timer = np.array([(datetime(1900,1,1) + timedelta(days=float(i)/24.)) \
         for i in meta['NETCDF_DIM_time_VALUES'][1:-1].split(',')])

    if (not Path(dst_filename).exists()) or force:

        '''
        Get geotrans, proj, data type and number of bands
        from source dataset
        '''
        band1 = src.GetRasterBand(1)
        src_geotrans = src.GetGeoTransform()
        src_proj = src.GetProjection()

        nbands = src.RasterCount
        src_format = band1.DataType

        dst = gdal.GetDriverByName('MEM').Create(\
                                        '', wide, high, \
                                        nbands, src_format)
        dst.SetGeoTransform( match_geotrans )
        dst.SetProjection( match_proj)

        if verbose: print(f'reprojecting ...')
            # Output / destination
        _ = gdal.ReprojectImage(src, dst, \
                                    src_proj, \
                                    match_proj,\
                                    gdalconst.GRA_Bilinear )
        if verbose: print(f'cropping to {country_code:s} ...')
        done = gdal.Warp(dst_filename,
                        dst,
```

(continues on next page)

(continued from previous page)

```
                        format=frmat,
                        dstNodata=nodata,
                        cutlineDSName=shpfile,
                        cutlineWhere=f"FIPS='{country_code:s}'",
                        cropToCutline=True)
        del dst

    return(timer,dst_filename,extent)
```

```
from osgeo import gdal, gdalconst,osr
import numpy as np
from geog0111.process_timeseries import mosaic_and_clip
from datetime import datetime,timedelta
from geog0111.match_netcdf_to_data import match_netcdf_to_data
from geog0111.geog_data import procure_dataset
from pathlib import Path

# set conditions

country_code = 'UK'
year = 2017
shpfile = "data/TM_WORLD_BORDERS-0.3.shp"
src_filename = f'data/europe_data_{year}.nc'
dst_filename = f'data/europe_data_{year}_{country_code}.tif'
t2_filename = f'data/europe_data_{year}_{country_code}.npz'
# read in the LAI data for given country code
tiles = []
for h in [17, 18]:
    for v in [3, 4]:
        tiles.append(f"h{h:02d}v{v:02d}")


#read LAI
fname = f'lai_data_{year}_{country_code}.npz'
ofile = Path('data')/fname
lai = np.load(ofile)

if not Path(t2_filename).exists():
    print(f'calculating dataset match in {t2_filename}')
    # first get an exemplar LAI file, clipped to
    # the required limits. We will use this to match
    # the t2 dataset to
    match_filename = mosaic_and_clip(tiles,1,year,\
                        country_code=country_code,\
                        shpfile=shpfile,frmat='GTiff')
    '''
    Match the datasets using the function
    we have developed
    '''
    meta = gdal.Open(src_filename, gdalconst.GA_ReadOnly).GetMetadata()

    timer,dst_filename,extent = match_netcdf_to_data(\
                                    src_filename,match_filename,\
                                    dst_filename,year,\
                                    country_code=country_code,\
                                    shpfile=shpfile,\
```

(continues on next page)

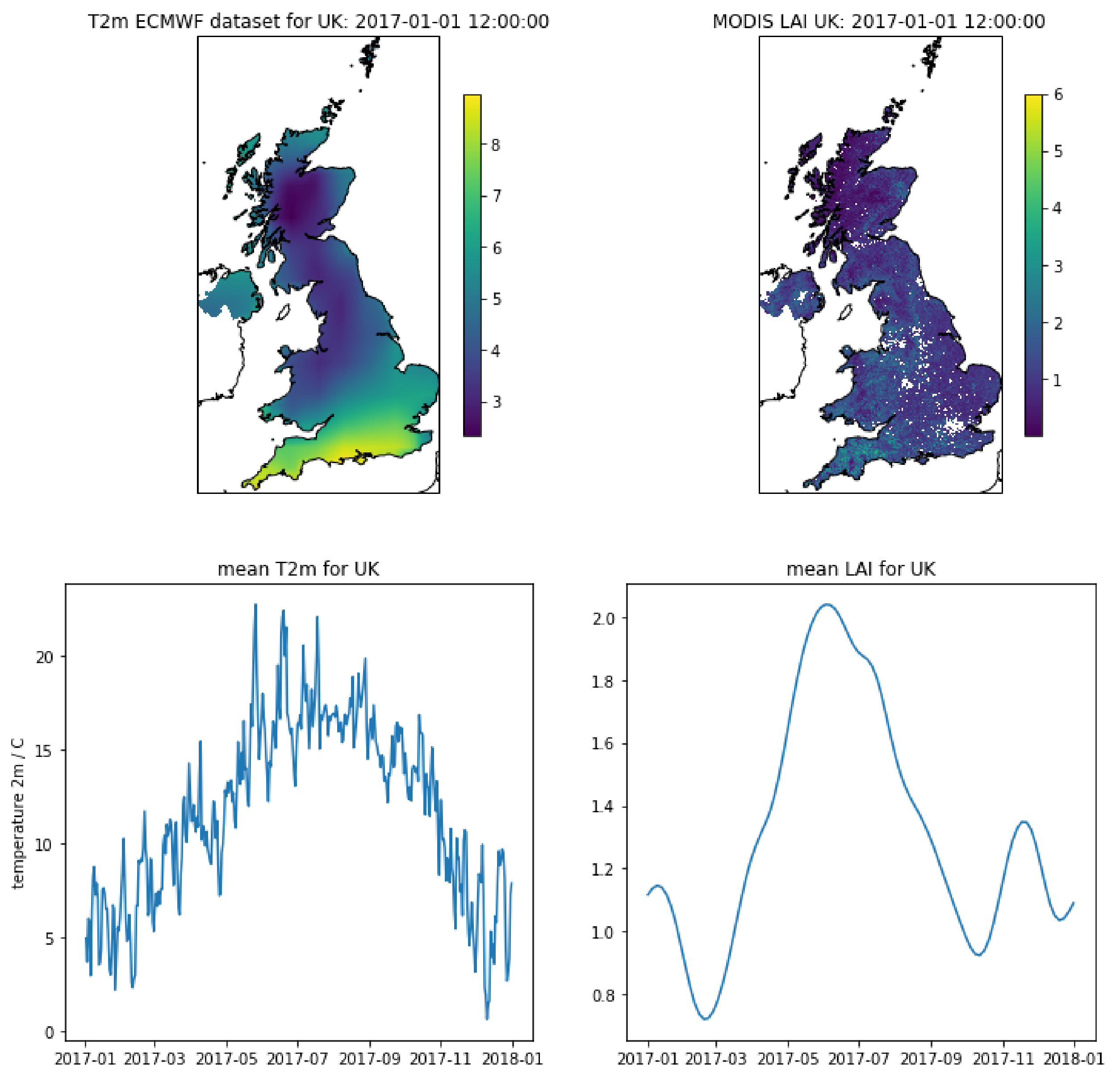
(continued from previous page)

```
                                    nodata=-300,frmat='GTiff',\
                                    verbose=True)

    # read and interpret the t2 data and flip
    temp2 = gdal.Open(dst_filename).ReadAsArray()[:,::-1]
    temp2[temp2==-300] = np.nan
    temp2 -= 273.15
    # save these
    print(f'saving data to {t2_filename}')
    np.savez_compressed(t2_filename,timer=timer,temp2=temp2,extent=extent)

else:
    print(f'dataset in {t2_filename} exists')

print('done')
t2data = np.load(t2_filename)
timer,temp2,extent = t2data['timer'],t2data['temp2'],t2data['extent']
```

```
calculating dataset match in data/europe_data_2017_UK.npz
getting info from match file data/Lai_500m_2017_001_UK.tif
getting info from source netcdf file data/europe_data_2017.nc
saving data to data/europe_data_2017_UK.npz
done
```

```
# visualise the interpolated dataset
import matplotlib.pylab as plt
import cartopy.crs as ccrs
%matplotlib inline

plt.figure(figsize=(12,12))
ax = plt.subplot ( 2, 2, 1 ,projection=ccrs.Sinusoidal.MODIS)
ax.coastlines('10m')
ax.set_title(f'T2m ECMWF dataset for {country_code}: {str(timer[0])}')
im = ax.imshow(temp2[0],extent=extent)
plt.colorbar(im,shrink=0.75)

ax = plt.subplot ( 2, 2, 2 ,projection=ccrs.Sinusoidal.MODIS)
ax.coastlines('10m')
ax.set_title(f'MODIS LAI {country_code}: {str(timer[0])}')
im = plt.imshow(interpolated_lai[:,:,0],vmax=6,extent=extent)
_ = plt.colorbar(im,shrink=0.75)

plt.subplot ( 2, 2, 3 )
plt.title(f'mean T2m for {country_code}')
plt.plot(timer,np.nanmean(temp2,axis=(1,2)))
plt.ylabel('temperature 2m / C')
plt.subplot ( 2, 2, 4 )
plt.title(f'mean LAI for {country_code}')
mean = np.nanmean(interpolated_lai,axis=(0,1))
plt.plot(timer[::4],mean)
```

```
[<matplotlib.lines.Line2D at 0x11dc20b38>]
```



3.6.6 Summary

In this section, we have learned about projections, and have reconciled two datasets that were originally in different projections. NThey also were defined with geoids with different Earth radius assumptions.

These issues are typical when dealing with geospatial data.

This part of the notes is non compulsory, as the codes and ideas are quite complicated for people just begining to learn coding. We have included it here to allow students to revisit this later. It is also included because we want to develop some interesting datasets for modelling, so we need to deal with reconciling datasets from different providers in different projections.

In this section, we have developed the following datasets:

```
from geog0111.geog_data import procure_dataset
import numpy as np
from pathlib import Path

year = 2017
country_code = 'UK'
'''
LAI data
'''
# read in the LAI data for given country code
lai_filename = f'data/lai_data_{year}_{country_code}.npz'
# get the dataset in case its not here
procure_dataset(Path(lai_filename).name,verbose=False)

lai = np.load(lai_filename)
print(lai_filename,list(lai.keys()))

'''
T 2m data
'''
t2_filename = f'data/europe_data_{year}_{country_code}.npz'
# get the dataset in case its not here
procure_dataset(Path(t2_filename).name,verbose=False)
t2data = np.load(t2_filename)
print(t2_filename,list(t2data.keys()))
```

```
data/lai_data_2017_UK.npz ['dates', 'lai', 'weights', 'interpolated_lai']
data/europe_data_2017_UK.npz ['timer', 'temp2', 'extent']
```

```
import numpy as np
# a quick look at some stats to see if there are data there
# and they are sensible
lai = np.load(ofile)
print(np.array(lai['lai'][1000,700]),\
      np.array(lai['weights'][1000,700]))
# does it have the interpolated value?
if 'interpolated_lai' in list(lai.keys()):
    print(np.array(lai['interpolated_lai'][1000,700]))
```

```
[1.1 1.1 1.2 0.9 0.9 1.2 1.  0.8 1.4 0.3 0.  0.5 0.3 0.9 0.6 1.  0.7 0.5
 0.7 0.6 1.3 0.7 0.1 1.  1.  0.6 1.1 0.5 1.1 1.  1.1 1.1 1.1 0.  0.6 1.4
 1.3 1.6 1.7 1.7 1.1 0.3 1.3 1.7 1.5 1.2 0.5 0.6 1.6 3.1 0.3 2.  1.6 0.5
 2.6 2.5 0.4 0.4 0.4 0.4 0.4 1.1 1.7 0.5 1.5 1.4 0.1 1.4 1.  1.  0.3 1.1
 0.2 1.1 0.1 0.7 2.6 1.7 2.  1.4 1.4 0.3 0.4 0.7 1.1 1.1 0.8 0.8 0.9 1.2
 1.2 1.2] [0.38196601 0.38196601 0.38196601 0.38196601 0.38196601 0.38196601
 0.38196601 0.38196601 0.38196601 0.38196601 0.38196601 1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         0.23606798 1.         1.
 1.         1.         1.         1.         0.23606798 0.23606798
 1.         1.         1.         1.         0.23606798 1.
 1.         1.         0.23606798 1.         1.         0.23606798
 1.         1.         0.23606798 0.23606798 0.23606798 0.23606798
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
```

(continues on next page)

(continued from previous page)

```
 0.23606798 1.         1.         0.38196601 0.38196601 0.38196601
 0.38196601 0.38196601 0.38196601 0.38196601 0.38196601 0.38196601
 0.38196601 0.38196601 0.38196601 0.38196601 0.38196601 0.38196601
 0.38196601 0.38196601]
[1.08683704 1.07935938 1.06092468 1.03161555 0.99133678 0.93493706
 0.86317607 0.78136836 0.70352293 0.64487202 0.61266704 0.60678279
 0.61883651 0.63991235 0.66383767 0.68530728 0.7020839  0.71506691
 0.72433462 0.73247513 0.74002342 0.7487489  0.76081558 0.77753903
 0.79985155 0.82674875 0.8580399  0.89089931 0.92313233 0.95320821
 0.98065863 1.00873477 1.04290792 1.08866497 1.14943392 1.22080857
 1.2929335  1.35676154 1.4021202  1.42632177 1.43105239 1.41880135
 1.40112126 1.38702625 1.38535581 1.40496862 1.45132144 1.52190587
 1.6097794  1.70184005 1.78498301 1.84800873 1.88579434 1.89137779
 1.85216521 1.7573269  1.60194669 1.4068558  1.2226864  1.09378123
 1.03387865 1.01734516 1.02167784 1.02682062 1.02221966 1.00590369
 0.98109385 0.95002526 0.91607184 0.8832294  0.85542066 0.83730438
 0.83548611 0.85858254 0.90888448 0.98602178 1.07794995 1.15678679
 1.19786983 1.18422965 1.12426637 1.04138037 0.9671476  0.91852027
 0.90214004 0.90955621 0.93302657 0.96662729 1.00304032 1.03742083
 1.06445991 1.08219511]
```

Exercise 3.6.2 Extra Homework

Go carefully through these notes and make notes of the processes we have to go through to reconcile datasets such as these.

Learn what issues to look out for when coming across a new dataset, and how to use Python code to deal with it. Try to stick to one geospatial package as far as possible (`gdal` here) as you can make problems for yourself by mixing them.

```
# do exercise here
```

Table of Contents

2.8.19 4. Assessed Practical

4.1 Introduction

4.1.1 Task overview

These notes describe the practical work you must submit for assessment in this course.

The practical comes in two parts: (1) data preparation (50%); (2) modelling (50%).

It is important that you complete both parts of this exercise.

The submission for Part 1 of the coursework (worth 50% of the marks) is the Monday after Reading week (12:00 Noon). That is Monday 12^{th} November 2018.

The submission for Part 2 will be 7^{th} January 2019 (12:00 Noon). Submission is through the usual Turnitin link on the course Moodle page.

- **Part 1: Data Preparation**

  The first task you must complete is to produce a dataset of the proportion of HUC catchment 13010001 (Rio Grande headwaters in Colorado, USA) that is covered by snow for **two years** (not necessarily consecutive), along with associated datasets on temperature (in C) and river discharge at the Del Norte monitoring station. You will use these data in the modelling work in Part 2 of the coursework.

  You **may not** use data from the years 2005 or 2006, as this will be given to you in illustrations of the material.

  The dataset you produce must have a value for **the mean snow cover, temperature and discharge in the catchment for every day over each year.**

  Your write up **must** include fully labelled graph(s) of snow cover, temperature and discharge for the catchment for each year (with units as appropriate), along with some summary statistics (e.g. mean or median, minimum, maximum, and the timing of these).

  You **must** provide evidence of how you got these data (i.e. the code and commands you ran to produce the data).

Checklist:

```
* provide fully commented/documented code for all operations.
* provide two years of **daily** data (not 2005 or 2006)
* Generate datasets of:
    * mean snow cover (0.0 to 1.0) for the catchment for each day of the year
    * temperature (C) at the Del Norte monitoring station for each day of the year
    * river discharge at the Del Norte monitoring station for each day of the year
```

(continues on next page)

(continued from previous page)

```
* Produce a table of summary statistics for each of the 3 datasets (one for easch␣
↪year)
* produce graphs of the 3 datasets for each year (as function of day of year)
* produce an `npz` file containing the 3 datasets, one for each year.
* produce images of snow cover spatial data for the catchment for **13 samples**␣
↪spaced equally through the year, one set of images for each year. You need to do␣
↪this for the data pre-interpolation and aftyer you have done the interpolation.
```

- **Part 2: Modelling**

  You will have prepared two years of data in Part 1 of the work.

  If, for some reason, you have failed to generate an appropriate dataset, you may use datasets that will be provided for you for the years 2005 and 2006. There will be no penalty for that in your Part 2 submission: failure to gernerate the datasets will be accounted for in marks allocated for Part 1.

  You will be given a simple hydrological model of snowmelt.

  Use one of these years to calibrate the (snowmelt) hydrological model and one year to test it.

  The model parameter estimate *must* be objective (i.e. you can't just arbitrarily choose a set) and optimal in some way you must define (you *must* state the equation of the cost function you will try to minimise and explain the approach used).

  You **must** state the values of the model parameters that you have estimated and show evidence for how you went about calculating them. Ideally, you should also state the uncertainty in these parameter estimates (not critical to pass this section though).

  You **must** quantify the goodness of fit between your measured flow data and that produced by your model, both for the calibration exercise and the validation.

Checklist:

```
* Provide a site intoduction and an introduction to the purpose of the exercise (
↪'Introduction')
* Provide an introduction to the modelling and calibration/validation ('Method')
* provide code that reads in the datasets and performs the model calibration and␣
↪validation ('Code')
* Provide a table of results on model parameter calibration (and ideally,␣
↪uncertainty) ('Results')
* Provide graphs of the observed and modelled river discharge data for the␣
↪calibration year ('Results')
* Provide graphs of the observed and modelled river discharge data for the validation␣
↪year ('Results')
* Assess the accuracy of the calibration and validation ('Results')
* Discuss the results in the light of the introduction ('Discussion')
* Draw conclusions about issues associated with modelling of this sort ('Conclusion')
```

You **must** work individually on this task. If you do not, it will be treated as plagiarism. By reading these instructions for this exercise, we assume that you are aware of the UCL rules on plagiarism. You can find more information on this matter in your student handbook. If in doubt about what might constitute plagiarism, ask one of the course convenors.

4.1.2 Purpose of the work

The hydrology of the Rio Grande Headwaters in Colorado, USA is snowmelt dominated. It varies considerably from year to year and may very further under a changing climate.

We can build a mathemetical ('environmental') model to describe the main physical processes affecting hydrology in the catchment. Such a model could help understand current behaviour and allow some prediction about possible future scenarios.

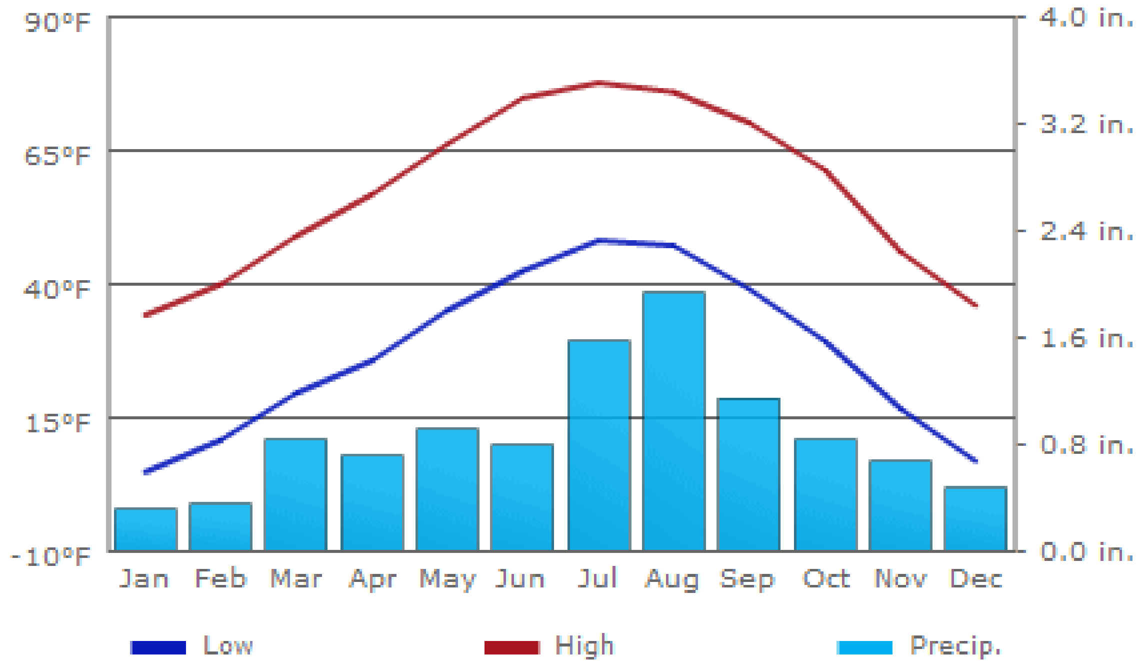
What you are going to do is to build, calibrate and test a (snowmelt) hydrological model, driven by observations in the Rio Grande Headwaters in Colorado, USA

files/images/temperature.png

The purpose of the model will be to describe the streamflow at the Del Norte measurement station, just on the edge of the catchment. You will use environmental (temperature) data and snow cover observations to drive the model. You will perform calibration and testing by comparing model output with observed streamflow data.

4.2.1.1 Del Norte

The average climate for Del Norte is:



Further general information is available from various websites, including NOAA.

Fig. 18: www.coloradofishing.net

You can visualise the site Del Norte 2E here.

4.2.1.2 Discharge dataset

First we should look at the streamflow/discharge data.

The link for downloading data from 2001 onwards (but not the current year until the end of the year): http://waterdata.usgs.gov.

files/images/temperature.png

You should examine all of the streamflow data and use that to make a decision on which years data you want to use in your experiments. You must justify your decision. You choice coul;d be based on the fact that some particular year shows ‘typical’ behaviour, or that some set of years seems to encompass the bounds of behaviour. The choies you make here will impact your ability to generalise from your results, so make sure you comment on this.

write some code to download the data, select one of the years that you want, and load the data into numpy arrays of time and streamflow (discharge).

run the code for the two years of data that you have selected and save the datasets to a file.

4.2.1.3 Temperature dataset

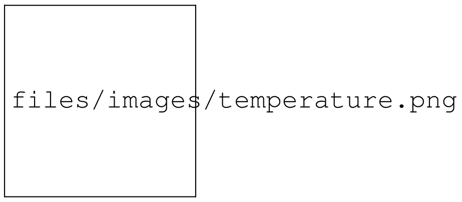
Go to the Colorado State Climate data site, select the Del Norte 2E site, and the period for which you want your data (typically, Jan 1^{st} to Dec 31^{st} for the years you want.

You can obtain data on maximum and mimimum temperatures, as well as snowfall and precipitation.

As the site doesn't provide a data download mechanism, you will need to save the data, most easily by copying and pasting the data columns into a file. Make a note of what you did here, to include in your write up.

You only really need the maximum temperature dataset, but you may find the others helpful for interpretation or to improve your model.

4.2.1.4 Snow cover dataset



You need to calculate the **daily snow cover** (values between 0 and 1 for the proportion of the catchment covered in snow) for the catchment (**HUC feature 2** (catchment 13010001)).

You would want to use a **daily** snow product for this task, such as that available from MODIS, so make sure you know what that is and explore the characteristics of the dataset.

You will notice from the figure above (the figure should give you some clue as to a suitable data product) that there will be areas of each image for which you have no information (described in the dataset QC). You will need to decide what to do about 'missing data'. For instance, you might consider interpolating over missing values.

The simplest thing you might think of might be to produce a mean snow cover over what samples are available (ignoring the missing values). But that would be rather poor. Really, you should apply some sort of interpolation (in time).

However you decide to process the data, you must give a rationale for why you have taken the approach you have done.

You will notice that if you use MODIS data, you have access to both data from Terra (MOD10A) and Aqua (MYD10A), which potentially gives you two samples per day. Think about how to take that into account. Again, the simplest thing to do might be to just use one of these. That is likely to be sufficient, but it would be much better to include both datasets.

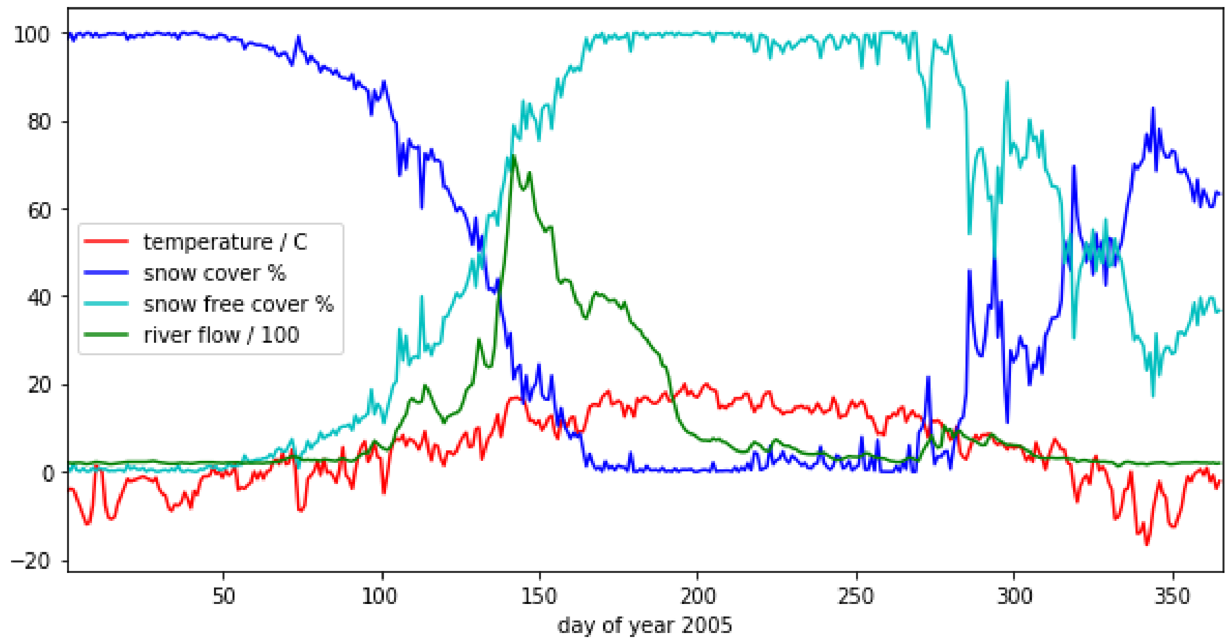
```
# load a pre-cooked version of the data for 2005 (NB -- Dont use this year!!!
# except perhaps for testing)

# load the data from a pickle file
import pickle
import pylab as plt
import numpy as np
%matplotlib inline
```

(continues on next page)

(continued from previous page)

```
with open('data/data.pkl', 'rb') as f:
    data = pickle.load(f, encoding='latin1')

# set up plot
plt.figure(figsize=(10,5))
plt.xlim(data['doy'][0],data['doy'][-1]+1)
plt.xlabel('day of year 2005')

# plot data
plt.plot(data['doy'],data['temp'],'r',label='temperature / C')
plt.plot(data['doy'],data['snowprop']*100,'b',label='snow cover %')
plt.plot(data['doy'],100-data['snowprop']*100,'c',label='snow free cover %')
plt.plot(data['doy'],data['flow']/100.,'g',label='river flow / 100')
plt.legend(loc='best')
```

```
<matplotlib.legend.Legend at 0x10725edd8>
```



we have plotted the streamflow (scaled) in green, the snow cover in blue, and the non snow cover in cyan and the temperature in red. It should be apparent that thge hydrology is snow melt dominated, and to describe this (i.e. to build the simplest possible model) we can probably just apply some time lag function to the snow cover.

4.2.2 Data Advice

4.2.2.1 MODIS snow cover data

For MODIS data, you will need to work out which data product you want and how to download it. To help you with this, the 'pattern' of the URLs is:

```
info = {'YEAR':2007,'MONTH':1,'DAY':1}
url = 'https://n5eil01u.ecs.nsidc.org/MOST/MOD10A1.006/' + \
    '{YEAR}.{MONTH:02d}.DAY:02d}/MYD*h09v05.006*hdf'.format(**info)
```

You should have codes from previous practicals sessions that allow you to download such data and you should then store the daily datasets in you local file system.

We can use the usual tools to explore the MODIS hdf files:

```
import gdal
modis_file = 'data/MOD10A1.A2016001.h09v05.006.2016183123533.hdf'
g = gdal.Open(modis_file)
if g is None:
    print ('error opening file: HDF4 problem with GDAL?')
else:
    # note this has changed in collection 6
    data_layer = 'MOD_Grid_Snow_500m:NDSI_Snow_Cover'

    subdatasets = g.GetSubDatasets()
    for fname, name in subdatasets:
        print (name)
        print ("\t", fname)

    fname = 'HDF4_EOS:EOS_GRID:"%s":%s'%(modis_file,data_layer)
    raster = gdal.Open(fname)
```

```
[2400x2400] NDSI_Snow_Cover MOD_Grid_Snow_500m (8-bit unsigned integer)
     HDF4_EOS:EOS_GRID:"data/MOD10A1.A2016001.h09v05.006.2016183123533.hdf":MOD_Grid_
↪Snow_500m:NDSI_Snow_Cover
[2400x2400] NDSI_Snow_Cover_Basic_QA MOD_Grid_Snow_500m (8-bit unsigned integer)
     HDF4_EOS:EOS_GRID:"data/MOD10A1.A2016001.h09v05.006.2016183123533.hdf":MOD_Grid_
↪Snow_500m:NDSI_Snow_Cover_Basic_QA
[2400x2400] NDSI_Snow_Cover_Algorithm_Flags_QA MOD_Grid_Snow_500m (8-bit unsigned␣
↪integer)
     HDF4_EOS:EOS_GRID:"data/MOD10A1.A2016001.h09v05.006.2016183123533.hdf":MOD_Grid_
↪Snow_500m:NDSI_Snow_Cover_Algorithm_Flags_QA
[2400x2400] NDSI MOD_Grid_Snow_500m (16-bit integer)
     HDF4_EOS:EOS_GRID:"data/MOD10A1.A2016001.h09v05.006.2016183123533.hdf":MOD_Grid_
↪Snow_500m:NDSI
[2400x2400] Snow_Albedo_Daily_Tile MOD_Grid_Snow_500m (8-bit unsigned integer)
     HDF4_EOS:EOS_GRID:"data/MOD10A1.A2016001.h09v05.006.2016183123533.hdf":MOD_Grid_
↪Snow_500m:Snow_Albedo_Daily_Tile
[2400x2400] orbit_pnt MOD_Grid_Snow_500m (8-bit integer)
     HDF4_EOS:EOS_GRID:"data/MOD10A1.A2016001.h09v05.006.2016183123533.hdf":MOD_Grid_
↪Snow_500m:orbit_pnt
[2400x2400] granule_pnt MOD_Grid_Snow_500m (8-bit unsigned integer)
     HDF4_EOS:EOS_GRID:"data/MOD10A1.A2016001.h09v05.006.2016183123533.hdf":MOD_Grid_
↪Snow_500m:granule_pnt
```

4.2.3.2 Boundary Data

Boundary data, such as catchments, might typically come as ESRI shapefiles or may be in other vector formats. There tends to be variable quality among different databases, but a reliable source for catchment data the USA is the USGS. One set of catchments in the tile we have is the Rio Grande headwaters, which we can see has a HUC 8-digit code of 13010001. The full dataset is easily found from the USGS or locally. Literature and associated data concerning this area can be found here. Associated GIS data are here, including the watershed boundary data.

Data more specific to our particular catchment of interest can be found on the Rio Grande Data Project pages.

You should use the file Hydrologic_Units.zip. You may need to `unzip` this file to get at the shapefile `` `Hydrologic_Units/HUC_Polygons.shp `` <data/Hydrologic_Units/HUC_Polygons.shp>'__ within it.

You can explore the shape file with the following:

```
!ogrinfo data/Hydrological_Units/HUC_Polygons.shp HUC_Polygons -nomd -geom=NO -where
↪"HUC=13010001"
```

```
INFO: Open of `data/Hydrological_Units/HUC_Polygons.shp'
      using driver `ESRI Shapefile' successful.

Layer name: HUC_Polygons
Geometry: Polygon
Feature Count: 1
Extent: (-1207861.193700, -1295788.385400) - (-115932.919500, 152769.254400)
Layer SRS WKT:
PROJCS["USA_Contiguous_Albers_Equal_Area_Conic",
    GEOGCS["GCS_North_American_1983",
        DATUM["North_American_Datum_1983",
            SPHEROID["GRS_1980",6378137.0,298.257222101]],
        PRIMEM["Greenwich",0.0],
        UNIT["Degree",0.0174532925199433],
        AUTHORITY["EPSG","4269"]],
    PROJECTION["Albers_Conic_Equal_Area"],
    PARAMETER["False_Easting",0.0],
    PARAMETER["False_Northing",0.0],
    PARAMETER["longitude_of_center",-96.0],
    PARAMETER["Standard_Parallel_1",29.5],
    PARAMETER["Standard_Parallel_2",45.5],
    PARAMETER["latitude_of_center",37.5],
    UNIT["Meter",1.0]]
HUC: Integer (9.0)
REG_NAME: String (50.0)
SUB_NAME: String (51.0)
ACC_NAME: String (36.0)
CAT_NAME: String (60.0)
HUC2: Integer (4.0)
HUC4: Integer (4.0)
HUC6: Integer (9.0)
REG: Integer (4.0)
SUB: Integer (4.0)
ACC: Integer (9.0)
CAT: Integer (9.0)
CAT_NUM: String (8.0)
Shape_Leng: Real (19.11)
Shape_Area: Real (19.11)
OGRFeature(HUC_Polygons):2
  HUC (Integer) = 13010001
  REG_NAME (String) = Rio Grande Region
  SUB_NAME (String) = Rio Grande Headwaters
  ACC_NAME (String) = Rio Grande Headwaters
  CAT_NAME (String) = Rio Grande Headwaters. Colorado.
  HUC2 (Integer) = 13
  HUC4 (Integer) = 1301
  HUC6 (Integer) = 130100
  REG (Integer) = 13
```

(continues on next page)

(continued from previous page)

```
  SUB (Integer) = 1301
  ACC (Integer) = 130100
  CAT (Integer) = 13010001
  CAT_NUM (String) = 13010001
  Shape_Leng (Real) = 313605.66409400001
  Shape_Area (Real) = 3458016895.23000001907
```

The catchment is only a small proportion of the total dataset, so make sure you apply masking and cropping appropriately.

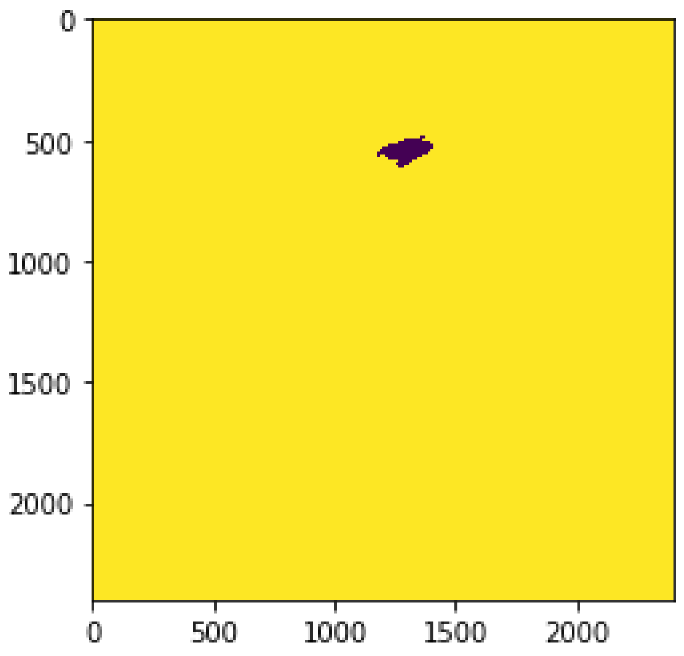
You will need to develop code to load the time series of snow cover data into a 3D numpy array, cropped to the required catchment

You will then need to develop code to interpolate over any missing data, so that there is an estimate of snow cover for every pixel in the catchment for all days

You will need to produce images (i.e. fully labelled plots) of snow cover spatial data for the catchment for 13 samples spaced equally through the year, one set of images for each year. This will be used to demonstrate that you have achieved this interpolation, so you should show the datyasets for both pre- and post-interpolation.

```
from geog0111.raster_mask import raster_mask2
import pylab as plt
%matplotlib inline

m = raster_mask2(fname,\
                target_vector_file="data/Hydrological_Units/HUC_Polygons.shp",\
                attribute_filter=2)
plt.imshow(m)
```

```
<matplotlib.image.AxesImage at 0x12646fe80>
```



4.2.3.3 Discharge Data

A sample of the river discharge data are in the file `data/delnorte.dat <data/delnorte.dat>`__.

If you examine the file:

```
f = open('data/delnorte.dat').readlines()
for i in range(40):
    print(f[i],end='')
```

```
# ---------------------------------- WARNING ----------------------------------------
# Provisional data are subject to revision. Go to
# http://help.waterdata.usgs.gov/policies/provisional-data-statement for more␣
↪information.
#
# File-format description:  http://help.waterdata.usgs.gov/faq/about-tab-delimited-
↪output
# Automated-retrieval info: http://help.waterdata.usgs.gov/faq/automated-retrievals
#
# Contact:   gs-w_support_nwisweb@usgs.gov
# retrieved: 2016-11-02 04:17:13 -04:00     (natwebsdas01)
#
# Data for the following 1 site(s) are contained in this file
#    USGS 08220000 RIO GRANDE NEAR DEL NORTE, CO
# -----------------------------------------------------------------------------------
#
# TS_ID - An internal number representing a time series.
# IV_TS_ID - An internal number representing the Instantaneous Value time series from␣
↪which the daily statistic is calculated.
#
# Data provided for site 08220000
#    TS_ID       Parameter    Statistic   IV_TS_ID       Description
#    18268       00060        00003       -1             Discharge, cubic feet per␣
↪second (Mean)
#
# Data-value qualification codes included in this output:
#     A  Approved for publication -- Processing and review completed.
#     e  Value has been edited or estimated by USGS personnel and is write protected.
#
agency_cd   site_no datetime        18268_00060_00003       18268_00060_00003_cd
5s  15s     20d     14n     10s
USGS        08220000        2005-01-01      210     A:e
USGS        08220000        2005-01-02      190     A:e
USGS        08220000        2005-01-03      190     A:e
USGS        08220000        2005-01-04      200     A:e
USGS        08220000        2005-01-05      200     A:e
USGS        08220000        2005-01-06      190     A:e
USGS        08220000        2005-01-07      170     A:e
USGS        08220000        2005-01-08      180     A:e
USGS        08220000        2005-01-09      200     A:e
USGS        08220000        2005-01-10      220     A:e
USGS        08220000        2005-01-11      210     A:e
USGS        08220000        2005-01-12      200     A:e
USGS        08220000        2005-01-13      190     A:e
```

you will see comment lines that start with `#`, followed by data lines.

The easiest way to read these data would be to use:

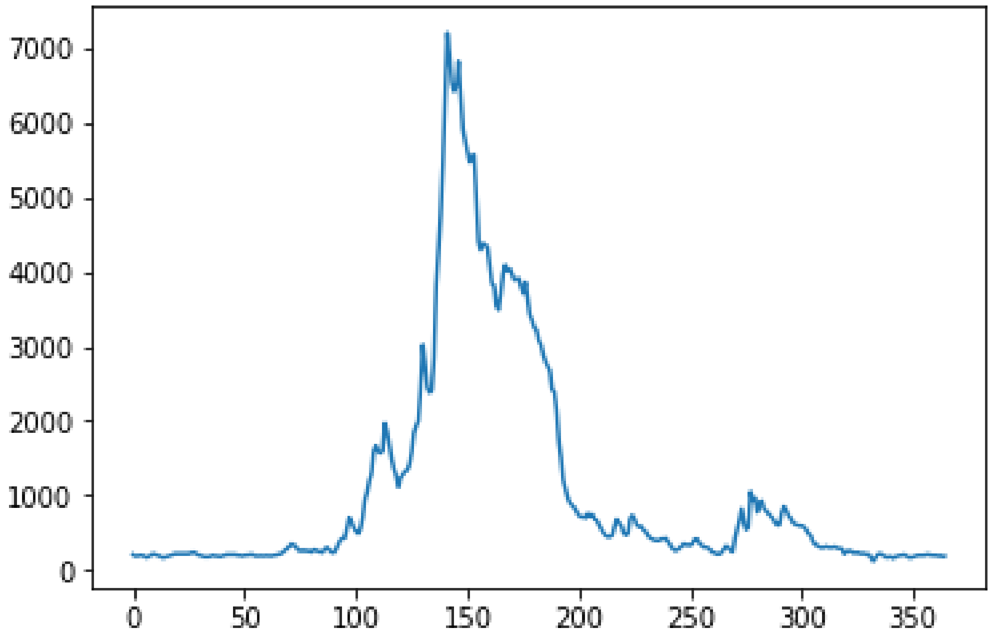
```
import numpy as np
file = 'data/delnorte.dat'
data = np.loadtxt(file,usecols=(2,3),unpack=True,dtype=str)
```

```
# so you have the dates in
# print the first 100
print (data[0][:100])
# and data in
print (data[1][:100])
# but the data start in column 3 so use [2:]
```

```
['datetime' '20d' '2005-01-01' '2005-01-02' '2005-01-03' '2005-01-04'
 '2005-01-05' '2005-01-06' '2005-01-07' '2005-01-08' '2005-01-09'
 '2005-01-10' '2005-01-11' '2005-01-12' '2005-01-13' '2005-01-14'
 '2005-01-15' '2005-01-16' '2005-01-17' '2005-01-18' '2005-01-19'
 '2005-01-20' '2005-01-21' '2005-01-22' '2005-01-23' '2005-01-24'
 '2005-01-25' '2005-01-26' '2005-01-27' '2005-01-28' '2005-01-29'
 '2005-01-30' '2005-01-31' '2005-02-01' '2005-02-02' '2005-02-03'
 '2005-02-04' '2005-02-05' '2005-02-06' '2005-02-07' '2005-02-08'
 '2005-02-09' '2005-02-10' '2005-02-11' '2005-02-12' '2005-02-13'
 '2005-02-14' '2005-02-15' '2005-02-16' '2005-02-17' '2005-02-18'
 '2005-02-19' '2005-02-20' '2005-02-21' '2005-02-22' '2005-02-23'
 '2005-02-24' '2005-02-25' '2005-02-26' '2005-02-27' '2005-02-28'
 '2005-03-01' '2005-03-02' '2005-03-03' '2005-03-04' '2005-03-05'
 '2005-03-06' '2005-03-07' '2005-03-08' '2005-03-09' '2005-03-10'
 '2005-03-11' '2005-03-12' '2005-03-13' '2005-03-14' '2005-03-15'
 '2005-03-16' '2005-03-17' '2005-03-18' '2005-03-19' '2005-03-20'
 '2005-03-21' '2005-03-22' '2005-03-23' '2005-03-24' '2005-03-25'
 '2005-03-26' '2005-03-27' '2005-03-28' '2005-03-29' '2005-03-30'
 '2005-03-31' '2005-04-01' '2005-04-02' '2005-04-03' '2005-04-04'
 '2005-04-05' '2005-04-06' '2005-04-07' '2005-04-08']
['18268_00060_00003' '14n' '210' '190' '190' '200' '200' '190' '170' '180'
 '200' '220' '210' '200' '190' '170' '170' '180' '190' '200' '210' '220'
 '220' '220' '220' '220' '220' '220' '230' '240' '230' '210' '200' '190'
 '180' '180' '180' '190' '200' '190' '190' '180' '190' '200' '210' '210'
 '210' '210' '210' '200' '200' '190' '200' '203' '205' '213' '207' '191'
 '190' '196' '190' '195' '196' '194' '189' '201' '201' '208' '218' '233'
 '262' '285' '320' '350' '340' '320' '272' '257' '272' '255' '266' '255'
 '245' '277' '270' '255' '248' '239' '284' '305' '270' '243' '235' '275'
 '344' '401' '424' '430' '576' '705']
```

```
# and the stream flow in data[1][2:]
plt.plot(data[1][2:].astype(float))
```

```
[<matplotlib.lines.Line2D at 0x1270929b0>]
```



You will need to convert the date field (i.e. the data in `data[0]`) into the day of year.

This is readily accomplished using `datetime`:

```
import datetime
# transform the first one, which is data[0][2:]

ds = np.array(data[0][2].split('-')).astype(int)
print (ds)
year,doy = datetime.datetime(ds[0],ds[1],ds[2]).strftime('%Y %j').split()
print (year,doy)
```

```
[2005    1    1]
2005 001
```

4.2.3.4 Temperature data

We can directly access temperature data from here.

The format of `` ` ``delNorteT.dat <files/data/delNorteT.dat>`__ is given here.

The first three fields are date fields (`YEAR`, `MONTH` and `DAY`), followed by `TMAX`, `TMIN`, `PRCP`, `SNOW`, `SNDP`.

You should read in the temperature data for the days and years that you want.

For temperature, you might take a **mean of TMAX and TMIN**.

Note that these are in Fahrenheit. You should convert them to Celcius.

Note also that there are missing data (values `9998` and `9999`). You will need to filter these and interpolate the data in some way. A median might be a good approach, but any interpolation will suffice.

With that processing then, you should have a dataset, Temperature that will look something like (in cyan, for the year 2005):

files/images/temperature.png

4.3 Coursework

You need to submit you coursework in the usual manner by the usual submission date.

You **must** work individually on this task. If you do not, it will be treated as plagiarism. By reading these instructions for this exercise, we assume that you are aware of the UCL rules on plagiarism. You can find more information on this matter in your student handbook. If in doubt about what might constitute plagiarism, ask one of the course convenors.

6.3.1 Summary of coursework requirements

- **Part 1: Data Preparation**

  The first task you must complete is to produce a dataset of the proportion of HUC catchment 13010001 (Rio Grande headwaters in Colorado, USA) that is covered by snow for **two years** (not necessarily consecutive), along with associated datasets on temperature (in C) and river discharge at the Del Norte monitoring station. You will use these data in the modelling work in Part 2 of the coursework.

  You **may not** use data from the years 2005 or 2006, as this will be given to you in illustrations of the material.

  The dataset you produce must have a value for **the mean snow cover, temperature and discharge in the catchment for every day over each year.**

  Your write up **must** include fully labelled graph(s) of snow cover, temperature and discharge for the catchment for each year (with units as appropriate), along with some summary statistics (e.g. mean or median, minimum, maximum, and the timing of these).

  You **must** provide evidence of how you got these data (i.e. the code and commands you ran to produce the data).

Checklist:

```
* provide fully commented/documented code for all operations.
* provide two years of **daily** data (not 2005 or 2006)
* Generate datasets of:
    * mean snow cover (0.0 to 1.0) for the catchment for each day of the year
    * temperature (C) at the Del Norte monitoring station for each day of the year
    * river discharge at the Del Norte monitoring station for each day of the year
* Produce a table of summary statistics for each of the 3 datasets (one for easch␣
↪year)
* produce graphs of the 3 datasets for each year (as function of day of year)
* produce an `npz` file containing the 3 datasets, one for each year.
* produce images of snow cover spatial data for the catchment for **13 samples**␣
↪spaced equally through the year, one set of images for each year. You need to do␣
↪this for the data pre-interpolation and aftyer you have done the interpolation.
```

4.3.2 Summary of Advice

The first task involves pulling datasets from different sources. No individual part of that should be too difficult, but you must put this together from the material we have done so far. It is more a question of organisation then.

Perhaps think first about where you want to end up with on this (the 'output'). This might for example be a dictionary with keys `temp`, `doy`, `snow` and `flow`, where each of these would be an array with 365 values (or 366 in a leap year).

Then consider the datasets you have: these are: (i) a stack of MODIS data with daily observations; (ii) temperature data in a file; (iii) flow data in a file.

It might be a little fiddly getting the data you want from the flow and temperature data files, but its not very complicated. You will need to consider flagging invalid observations and perhaps interpolating between these.

Processing the MODIS data might take a little more thought, but it is much the same process. Again, we read the datasets in, trying to make this efficient on data size by only using the area of the vector data mask as in a previous exercise. The data reading will be very similar to reading the MODIS LAI product, but you need to work out and implement what changes are necessary. As advised abovem you should use the `raster_mask2()` function for creating the spatial data masks. Again, you will need to interpolate or perhaps smooth between observations, and then process the snow cover proportions to get an average over the catchment.

The second task revolves around using the model that we have developed above in the function `model_accum()`. You have been through previous examples in Python where you attempt to estimate some model parameters given an initial estimate of the parameters and some cost function to be minimised. Solving the model calibration part of problem should follow those same lines then. Testing (validation) should be easy enough. Don't forget to include the estimated parameters (and other relevant information, e.g. your initial estimate, uncertainties if available) in your write up.

There is quite a lot of data presentation here, and you need to provide *evidence* that you have done the task. Make sure you use images (e.g. of snow cover varying), graphs (e.g. modelled and predicted flow, etc.), and tables (e.g. model parameter estimates) throughout, as appropriate.

If, for some reason, you are unable to complete the first part of the practical, you should submit what you can for that first part, and continue with calibrating the model using the 2005 dataset that we used above. This would be far from ideal as you would not have completed the required elements for either part in that case, but it would generall be better than not submitting anything.

4.3.3 Further advice

There is plenty of scope here for going beyond the basic requirements, if you get time and are interested (and/or want a higher mark!).

You will be given credit for all additional work included in the write up, **once you have achieved the basic requirements**. So, there is no point (i.e. you will not get credit for) going off on all sorts of interesting lines of exploration here *unless* you have first completed the core task.

4.3.4 Structure of the Report

The required elements of the report are:

```
 1. Code for temperature and river discharge data download, reading and saving, with␣
↪running of code and datasets for 2 years of daily data, and appropriate plots␣
↪showing the datasets (10%)
     * temperature (C) at the Del Norte monitoring station for each day of the year
     * river discharge at the Del Norte monitoring station for each day of the year

2. Code for downloading MODIS snow cover data, masking, cropping and extracting the␣
↪data into a 3D numpy array, storing the data, running the code for 2 years data,␣
↪and appropiate plots. (20%)
```

(continues on next page)

(continued from previous page)

```
3. Code for interpolating MODIS snow cover data and calculating mean snow cover over␣
↪the catchment, saving the data, running the code for 2 years data, and appropiate␣
↪plots. (15%)

4. Produce a table of summary statistics for each of the datasets (one for each year)␣
↪and a `npz` file containing all of the datasets (5%)

For parts 2 and 3, make sure to produce images of snow cover spatial data for the␣
↪catchment for **13 samples** spaced equally through the year, one set of images for␣
↪each year. You need to do this for the data pre-interpolation (part 2) and after␣
↪you have done the interpolation (part 3).
```

The figures in brackets indicate the percentage of marks that we will award for each section of the report.

4.3.5 Computer Code

General requirements

You will obviously need to submit computer codes as part of this assessment. Some flexibility in the style of these codes is to be expected. For example, some might write a class that encompasses the functionality for all tasks. Some poeple might have multiple versions of codes with different functionality. All of these, and other reasonable variations are allowed.

All codes needed to demonstrate that you have performed the core tasks are required to be included in the submission. You should include all codes that you make use of in the main body of the text in the main body. Any other codes that you want to refer to (e.g. something you tried out as an enhancement and didn't quite get there) you can include in appendices.

All codes should be well-commented. Part of the marks you get for code will depend on the adequacy of the commenting.

Degree of original work required and plagiarism

If you use a piece of code verbatim that you have taken from the course pages or any other source, **you must acknowledge this** in comments in your text. **Not to do so is plagiarism**. Where you have taken some part (e.g. a few lines) of someone else's code, **you should also indicate this**. If some of your code is heavily based on code from elsewhere, **you must also indicate that**.

Some examples.

The first example is guilty of strong plagiarism, it does not seek to acknowledge the source of this code, even though it is just a direct copy, pasted into a method called `model()`:

```
def model(tempThresh=9.0,K=2000.0,p=0.96):
    '''need to comment this further ...

    '''
    import numpy as np
    meltDays = np.where(temperature > tempThresh)[0]
    accum = snowProportion*0.
    for d in meltDays:
        water = K * snowProportion[d]
        n = np.arange(len(snowProportion)) - d
        m = p ** n
```

(continues on next page)

(continued from previous page)

```
        m[np.where(n<0)]=0
        accum += m * water
    return accum
```

This is **not** acceptable.

This should probably be something along the lines of:

```
def model(tempThresh=9.0,K=2000.0,p=0.96):
    '''need to comment this further ...

    This code is taken directly from
    "Modelling delay in a hydrological network"
    by P. Lewis http://www2.geog.ucl.ac.uk/~plewis/geogg122/DelNorte.html
    and wrapped into a method.
    '''
    # my code: make sure numpy is imported
    import numpy as np

    # code below verbatim from Lewis
    meltDays = np.where(temperature > tempThresh)[0]
    accum = snowProportion*0.
    for d in meltDays:
        water = K * snowProportion[d]
        n = np.arange(len(snowProportion)) - d
        m = p ** n
        m[np.where(n<0)]=0
        accum += m * water
    # my code: return accumulator
    return accum
```

Now, we acknowledge that this is in essence a direct copy of someone else's code, and clearly state this. We do also show that we have added some new lines to the code, and that we have wrapped this into a method.

In the next example, we have seen that the way m is generated is in fact rather inefficient, and have re-structured the code. It is partially developed from the original code, and acknowledges this:

```
def model(tempThresh=9.0,K=2000.0,p=0.96):
    '''need to comment this further ...

    This code after the model developed in
    "Modelling delay in a hydrological network"
    by P. Lewis
    http://www2.geog.ucl.ac.uk/~plewis/geogg122/DelNorte.html

    My modifications have been to make the filtering more efficient.
    '''
    # my code: make sure numpy is imported
    import numpy as np

    # code below verbatim from Lewis unless otherwise indicated
    meltDays = np.where(temperature > tempThresh)[0]
    accum = snowProportion*0.

    # my code: pull the filter block out of the loop
    n = np.arange(len(snowProportion))
    m = p ** n
```

(continues on next page)

(continued from previous page)

```
    for d in meltDays:
        water = K * snowProportion[d]

        # my code: shift the filter on by one day
        # ...do something clever to shift it on by one day

        accum += m * water
    # my code: return accumulator
    return accum
```

This example makes it clear that significant modifications have been made to the code structure (and probably to its efficiency) although the basic model and looping comes from an existing piece of code. It clearly highlights what the actual modifications have been. Note that this is not a working example!!

Although you are supposed to do this piece of work on your own, there might be some circumstances under which someone has significantly helped you to develop the code (e.g. written the main part of it for you & you've just copied that with some minor modifications). You **must** acknowledge in your code comments if this has happened. On the whole though, this should not occur, as you **must** complete this work on your own.

If you take a piece of code from somewhere else and all you do is change the variable names and/or other cosmetic changes, you **must** acknowledge the source of the original code (with a URL if available).

Plagiarism in coding is a tricky issue. One reason for that is that often the best way to learn something like this is to find an example that someone else has written and adapt that to your purposes. Equally, if someone has written some tool/library to do what you want to do, it would generally not be worthwhile for you to write your own but to concentrate on using that to achieve something new. Even in general code writing (i.e. when not submitting it as part of your assessment) you and anyone else who ever has to read your code would find it of value to make reference to where you found the material to base what you did on. The key issue to bear in mind in this work, as it is submitted 'as your own work' is that, to avoid being accused of plagiarism and to allow a fair assessment of what you have done, you must clearly acknowledge which parts of it are your own, and the degree to which you could claim them to be your own.

For example, based on ... is absolutely fine, and you would certainly be given credit for what you have done. In many circumstances 'taken verbatim from ...' would also be fine (provided it is acknowledged) but then you would be given credit for what you had done with the code that you had taken from elsewhere (e.g. you find some elegant way of doing the graphs that someone has written and you make use of it for presenting your results).

The difference between what you submit here and the code you might write if this were not a piece submitted for assessment is that you the vast majority of the credit you will gain for the code will be based on the degree to which you demonstrate that you can write code to achieve the required tasks. There would obviously be some credit for taking codes from the coursenotes and bolting them together into something that achieves the overall aim: provided that worked, and you had commented it adequately and acknowledge what the extent of your efforts had been, you should be able to achieve a pass in that component of the work. If there was no original input other than vbolting pieces of existing code together though, you be unlikely to achieve more than a pass. If you get less than a pass in another component of the coursework, that then puts you in danger of an overall fail.

Provided you achieve the core tasks, the more original work that you do/show (that is of good quality), the higher the mark you will get. Once you have achieved the core tasks, even if you try something and don't quite achieve it, is is probably worth including, as you may get marks for what you have done (or that fact that it was a good or interesting thing to try to do).

Documentation

Note: All methods/functions and classes must be documented for the code to be adequate. Generally, this will contain:

- some text on the purpose of the method (/function/class)
- some text describing the inputs and outputs, including reference to any relevant details such as datatype, shape etc where such things are of relevance to understanding the code.
- some text on keywords, e.g.:

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)

    Example taken verbatim from:
    http://www.python.org/dev/peps/pep-0257/
    """
    if imag == 0.0 and real == 0.0: return complex_zero
```

You should look at the document on good docstring conventions when considering how to document methods, classes etc.

To demonstrate your documentation, you **must** include the help text generated by your code after you include the code. e.g.:

```
def print_something(this,stderr=False):
    '''This does something.

    Keyword arguments:
    stderr -- set to True to print to stderr (default False)
    '''

    if stderr:
        # import sys.stderr
        from sys import stderr

        # print to stderr channel, converting this to str
        print >> stderr,str(this)

        # job done, return
        return

    # print to stdout, converting this to str
    print (str(this))

    return
```

Then the help text would be:

```
help(print_something)
```

```
Help on function print_something in module __main__:

print_something(this, stderr=False)
    This does something.

    Keyword arguments:
    stderr -- set to True to print to stderr (default False)
```

The above example represents a 'good' level of commenting as the code broadly adheres to the style suggestions and most of the major features are covered. It is not quite 'very good/excellent' as the description of the purpose of the method (rather important) is trivial and it fails to describe the input this in any way. An excellent piece would do all of these things, and might well tell us about any dependencies (e.g. requires sys if stderr set to True).

An inadequate example would be:

```python
def print_something(this,stderr=False):
    '''This prints something'''
    if stderr:
        from sys import stderr
        print >> stderr,str(this)
        return
    print (str(this))
```

It is inadequate because it still only has a trivial description of the purpose of the method, it tells us nothing about inputs/outputs and there is no commenting inside the method.

Word limit

There is no word limit per se on the computer codes, though as with all writing, you should try to be succint rather than overly verbose.

Code style

A good to excellent piece of code would take into account issues raised in the style guide. The 'degree of excellence' would depend on how well you take those points on board.

2.8.20 Table of Contents

2.8.21 Fitting to the Mauna Loa CO_2 record

Introduction

CO_2 concentration in the atmosphere has been steadily increasing. The flask measurements collected in Mauna Loa provide a fairly long time series that allows us to see the temporal evolution of this trace gas.

Why is CO2 important?

The time series looks like this

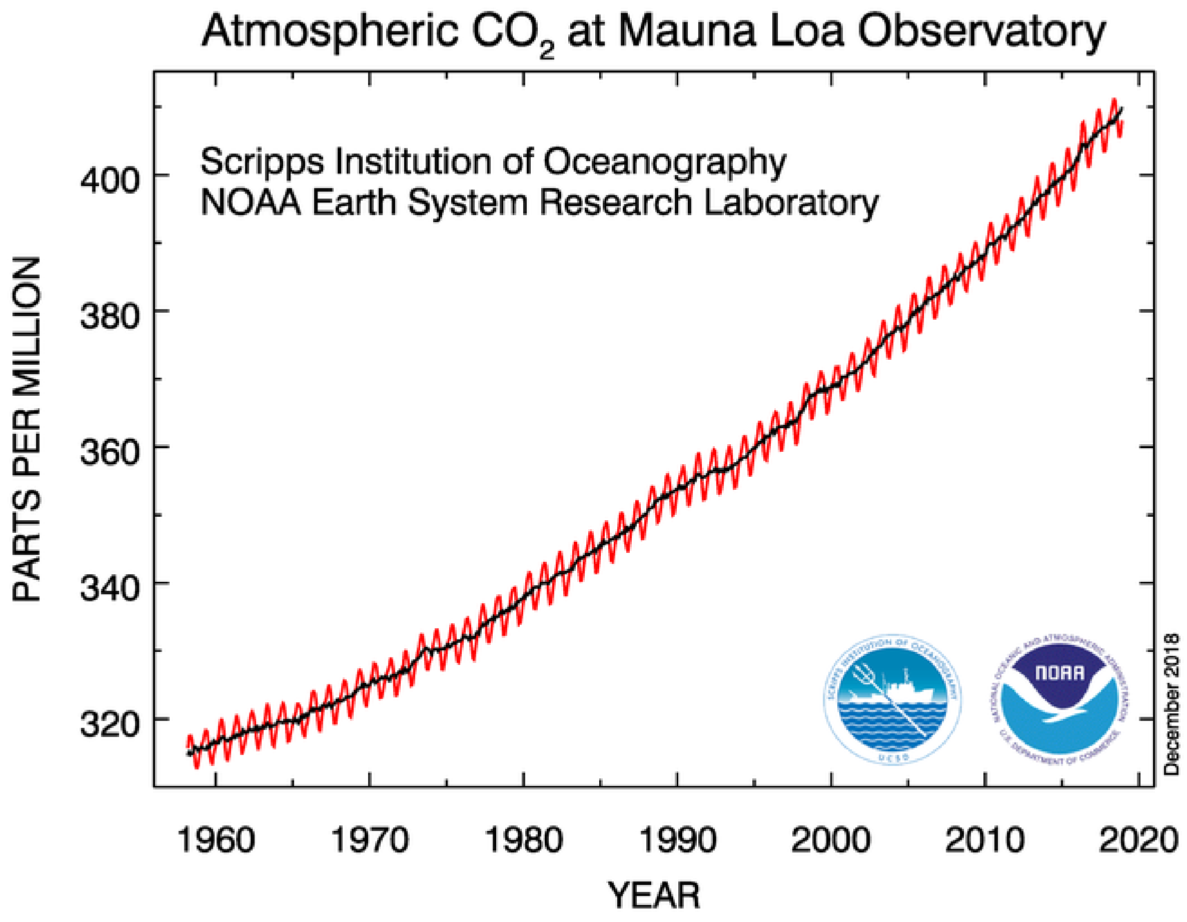


Fig. 19: Mauna Loa CO2 record

Knowing that Mauna Loa is in Hawaii (Northern Hemisphere), can you broadly explain what's going on?

In today's session, we shall consider how to "model" the CO_2 record. We'll develop some simple models of CO_2 as a function of time, and will try to "fit" them to the data.

Can you think why fitting some models based on time to the Mauna Loa record might not be particularly insightful?

We first have an import cell that deals with importing all the usual modules that we'll require: numpy, matplotlib as well as pandas to read the data file. If you put all your imports at the top and run them first, they should be available for all other cells. . .

```
from pathlib import Path  # Checks for files and so on
import numpy as np  # Numpy for arrays and so on
import pandas as pd
import sys
import matplotlib.pyplot as plt  # Matplotlib for plotting
# Ensure the plots are shown in the notebook
%matplotlib inline

# Instead of using requests, we might as well use Python's buil-in
# HTTP downloader
from urllib.request import urlretrieve
```

Obtaining the data

Downloading the data

The data are available on line from NOAA. We want the monthly average dataset, which can be found there. If the data is not yet available in your system, the next Python cell will download it from `` `ftp://aftp.cmdl.noaa.gov/products/trends/co2/co2_mm_mlo.txt `` <ftp://aftp.cmdl.noaa.gov/products/trends/co2/co2_mm_mlo.txt>`__. In this case, because the url is an FTP one we will use the `urllib2` package rather than requests, that doesn't deal with FTP. We will save it to a file with the same name locally:

```
# The remote URL for the data file is address:
address = 'ftp://aftp.cmdl.noaa.gov/products/trends/co2/'
# We'll create a folder if it doesn't exist in the data folder for
# the data
dest_path = Path("data/Mauna_Loa/").mkdir(parents=True, exist_ok=True)
fname = Path("data/Mauna_Loa/co2_mm_mlo.txt")
if not fname.exists():
    # Data file not present, let's download it
    print("Downloading remote file")
    urlretrieve(f"{address:s}/{fname.name:s}", fname.as_posix())
    print(f"Remote file downloaded to {fname.name:s}")
else:
    print(f"{fname.name:s} already present, no need to download again")
```

```
co2_mm_mlo.txt already present, no need to download again
```

Exploring the data

We can have a peek at the text file. We note that most of the first few lines are "comments" (lines start by `#`), which describe useful *metadata*. We note that we have several columns of data:

1. The year
2. The month
3. The decimal date
4. The monthly mean CO2 mole fraction determined from daily averages

We will mostly be bothered about columns three and four.

We can peek at the data (first 73 lines) using the UNIX shell `` `head `` <http://www.linfo.org/head.html>`__ command (this will not work on Windows, but will probably work on OSX):

```
!head -n 73 co2_mm_mlo.txt
```

```
head: cannot open ‘co2_mm_mlo.txt’ for reading: No such file or directory
```

Loading the data into Python

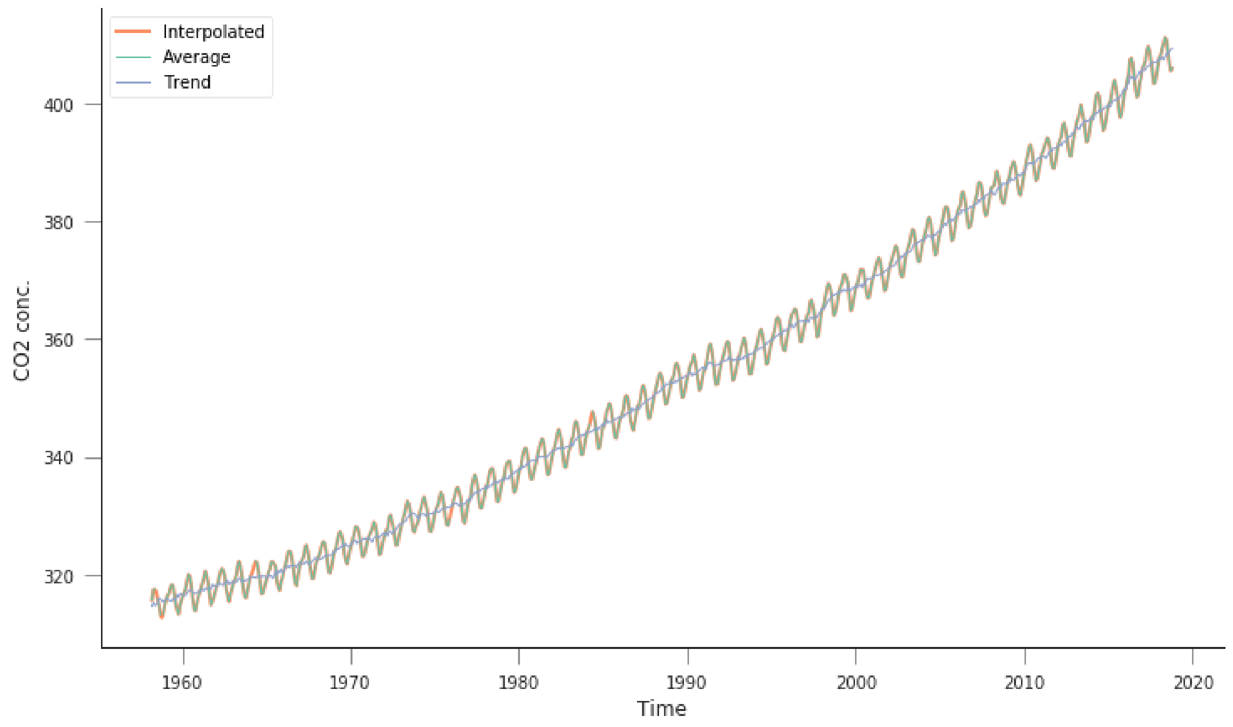
This is quite straightforward using `np.loadtxt <https://scipython.com/book/chapter-6-numpy/examples/using-numpys-loadtxt-method/>`__…

We will also "mask" if the data is missing checking for the value -99.99…

```
hdr = [
    "year", "month", "decimal_date", "average", "interpolated", "trend", "days"
]
co2 = pd.read_csv(
    fname,
    comment='#',
    delim_whitespace=True,
    names=hdr,
    na_values=[-99.99, -1])

plt.figure(figsize=(12, 7))
plt.plot(co2.decimal_date, co2.interpolated, '-', lw=2, label="Interpolated")
plt.plot(co2.decimal_date, co2.average, '-', lw=1, label="Average")
plt.plot(co2.decimal_date, co2.trend, '-', lw=1, label="Trend")
plt.xlabel("Time")
plt.ylabel("CO2 conc.")
plt.legend(loc="best")
```

```
<matplotlib.legend.Legend at 0x7f00481c2160>
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



So this is quite similar to what we had above. There's an average line, an interpolated line, as well as some smoothed trend line. We're interested in the interpolated line.

A model for CO_2 concentration

A linear trend model

We might be curious about a simple model for CO_2 concentration. Perhaps the simplest model is a linear trend, which we can write as the concentration at some time step i, W_i being just a linear scaling of the time t_i:

$$W_i = m \cdot t_i + c.$$

We can define a Python function for this very easily:

```
def linear_model(p, t):
    m, c = p
    return m * t + c
```

We can now try to plot some model trajectories and the data by supplying parameters for the slope (m) and intercept (c). Let's start by assuming that the slope can be approximated by the difference between minimum and maximum concentrations divided by the number of timesteps:

$$m \approx \frac{411 - 310}{728}$$

c is the minimum value, so $c \approx 310$.

```
n_times = co2.interpolated.shape[0]
max_co2 = co2.interpolated.max()
min_co2 = co2.interpolated.min()
```

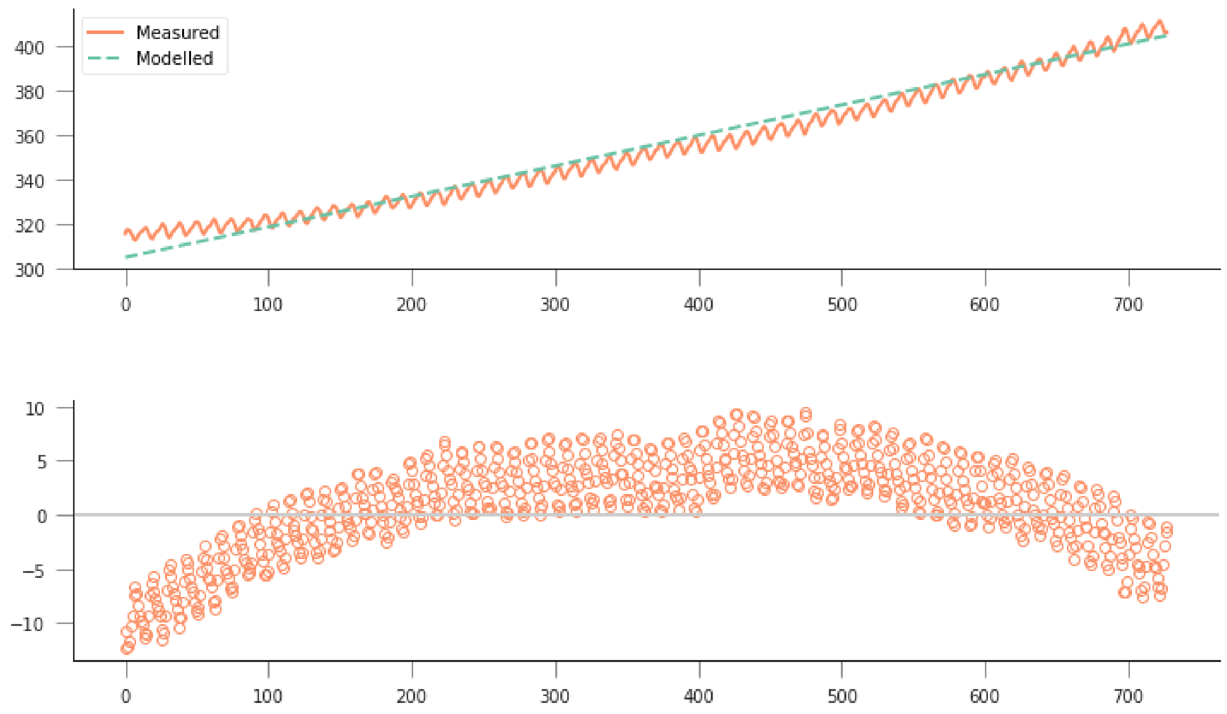
(continues on next page)

(continued from previous page)

```
print(f"There are {n_times:d} steps in the data")
print(f"Maximum CO2 concentration {max_co2:f}")
print(f"Minimum CO2 concentration {min_co2:f}")
x = np.arange(n_times)
fig, axs = plt.subplots(nrows=2, ncols=1, figsize=(12, 7))
axs[0].plot(x, co2.interpolated, '-', label="Measured")
m = (403. - 305.) / 716
c = 305.
axs[0].plot(x, linear_model([m, c], x), '--', label="Modelled")
axs[0].legend(loc="best")
axs[1].plot(x, linear_model([m, c], x) - co2.interpolated, 'o', mfc="none")
axs[1].axhline(y=0, lw=2, c="0.8")
S = np.sum((linear_model([m, c], x) - co2.interpolated)**2)
print("Sum of squared residuals: {}".format(S))
```

```
There are 728 steps in the data
Maximum CO2 concentration 411.240000
Minimum CO2 concentration 312.660000
Sum of squared residuals: 14997.256482063598
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



So, not really a great fit… The overall shape is a bit off, and the model isn't really fitting the annual seasonality in the curve. The residuals plot tells us that the residuals aren't really noise around zero: they show a very clear trend, suggesting that **the model is too simple to fit the data**.

A quadratic model

Maybe we need a higher order model, like a quadratic model:

$$W_i = a_0 \cdot t_i^2 + a_1 \cdot t_i + a_2.$$

In this case, it is a bit harder to eyeball what good starting parameters for $[a_0, a_1, a_2]$ would be. A strategy for this would be to consider what a good fit would look like, and then use this to define a metric of good fit. A good fit would basically overlap the measurements, being indistinguishable from them. The *residual* is the difference between the measurement and the model. In this case, it can be positive or negative (whether the model over- or undershoots the observations), but by squaring the residual we get rid of the sign. Then we can add up all the squared residuals, and the best fit will be the one that has the lowest sum of squares. This is in essence the method of least squares. Let's see how this works *intuitevely*: we'll loop over the parameters and plot the different predicted concentrations… First we need our model function…

```
def quadratic_model(p, t):
    a0, a1, a2 = p
    return a0 * t**2 + a1 * t + a2
```

We can get a feeling of what the parameters might be just by eyeballing reading up some points from the graph, and then solving the system manually:

$$\begin{aligned} 403 &= a_0 \cdot (728)^2 + a_1 \cdot (728) + a_2 \\ 340 &= a_0 \cdot (300)^2 + a_1 \cdot (300) + a_2 \\ 315 &= a_0 \cdot (0)^2 + a_1 \cdot (0) + a_2 \end{aligned}$$

From this, we can get some rough estimates, which in this case are

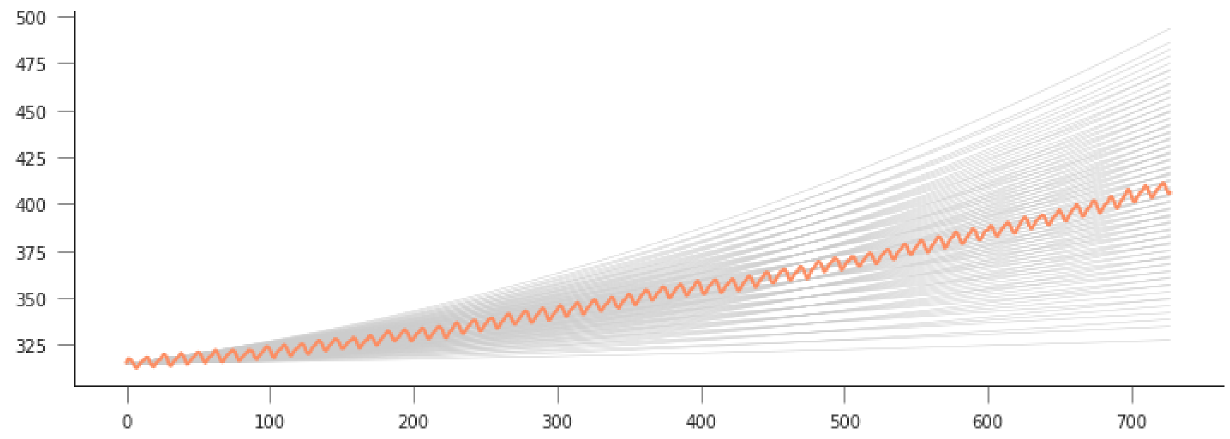
$$\begin{aligned} a_0 &= 9.5 \cdot 10^{-5} \\ a_1 &= 5.48 \cdot 10^{-2} \\ a_2 &= 315. \end{aligned}$$

We can just basically run the model around these numbers and plot the different model predictions with a loop over a_0 and another one over a_1 (assuming a_2 is well defined)

```
plt.figure(figsize=(12, 4))

a2 = 315.
for a0 in np.linspace(1e-5, 20e-5, 10):
    for a1 in np.linspace(1e-2, 10e-2, 10):
        plt.plot(x, quadratic_model([a0, a1, a2], x), '-', lw=0.5, c="0.8")

plt.plot(x, co2.interpolated, '-', label="Measured")
```

```
[<matplotlib.lines.Line2D at 0x7f00480047b8>]
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```

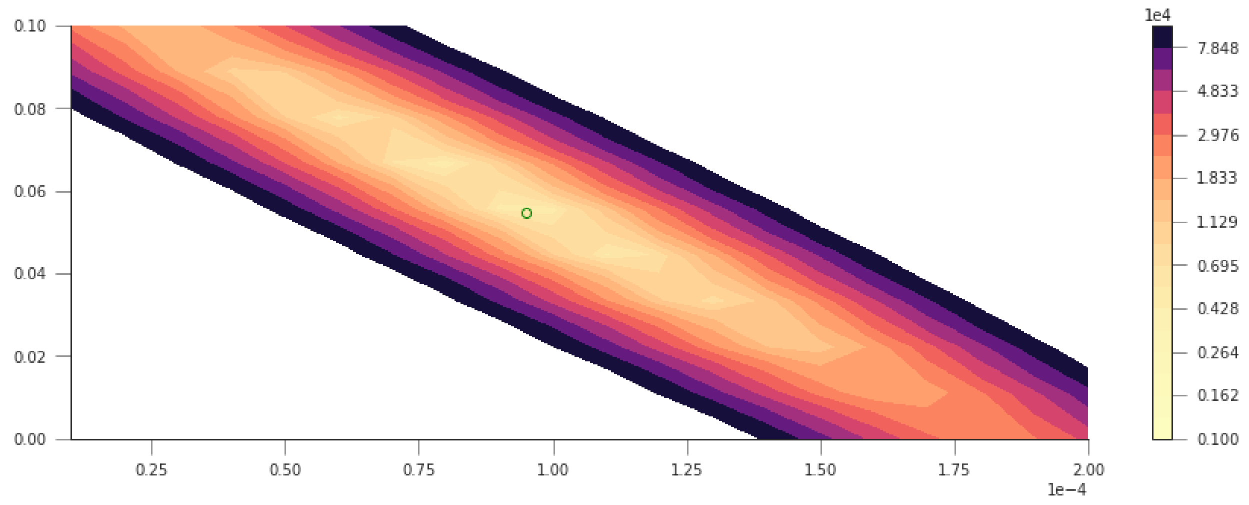


This is quite complicated, we can see that there might be a good line of fit, but we don't see clearly what parameters provide it! We can store the goodness of fit metric (sum of squared residuals) in a 2D array and then plot it as an image. It should be more obvious where the minimum lies…

```
# Define a 2D array for the sum of squares (sos)
sos = np.zeros((10, 20))
# the time axis redefined again, in case it got confused with something else
x = np.arange(n_times)

# first loop is over a0, 20 steps between 1e-5 and 20e-5
for ii, a0 in enumerate(np.linspace(1e-5, 20e-5, 20)):
    # 2nd loop is over a1, 10 steps between 1e-2 and 10e-2
    for jj, a1 in enumerate(np.linspace(1e-2, 10e-2, 10)):
        # for the current values of a0 and a1, calculate the residual
        residual = quadratic_model([a0, a1, a2], x) - co2.interpolated
        sq_residual = residual * residual
        sum_of_residuals = sq_residual.sum()
        # Store the sum_of_residuals into our array
        sos[jj, ii] = np.sum(
            (quadratic_model([a0, a1, a2], x) - co2.interpolated)**2)

# Plotting!
plt.figure(figsize=(15, 5))
# Set up the x and y axis for the plot
yy = np.linspace(1e-5, 20e-5, 20)
xx = np.linspace(1e-5, 10e-2, 10)
# Do a contour plot. The logspace bit basically defines the location
# of 20 contour lines
c = plt.contourf(yy, xx, sos, np.logspace(3, 5, 20), cmap=plt.cm.magma_r)
# Colorbar
plt.colorbar()
# Now, just plot the rough guess of a0 and a1 into this plot
# We want to plot an empty circle with a green edge
plt.plot(9.51242659e-05, 5.47960536e-02, 'o', mfc="None", mec="g")
```

```
[<matplotlib.lines.Line2D at 0x7f00422d7cc0>]
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



So that's pretty interesting, we get a very clear "valley", with a minimum pretty close to where our first rough guess is... The shape is quite interesting: if we start at the first guess point, and move along the $x-$ or $y-$ axes, we quickly go into areas of large error. However, if we move along the diagonal line, we will be in the "trough" of the cost function, provided that when you move "up" (positive a_0), you also move "left" (negative a_1), or if you move "down" (negative a_0), you also move "right" (positive a_1). Basically, the cost function does not change if you can get the two parameters to co-operate and compensate the effect of each other.

Let's find out where the actual minimum from our brute-force approach is. We can do this quickly by creating a mask where all the elements are `False` except where the minimum value of `sos` is located. We can then use this mask to multiply our `x` and `y` axes and just select the unique values that are larger than 0.

```
print(f"Best SoS: {sos.min():g}")
sos_mask = sos == sos.min()
u1 = np.unique(yy[None, :] * sos_mask)
yy_opt = u1[u1 > 0]
u2 = np.unique(xx[:, None] * sos_mask)
xx_opt = u2[u2 > 0]
```

```
Best SoS: 4182.24
```

The Sum of Squares of the first example was around 15000, so we've improved our modelling by adding an extra (quadratic term). This is usually the case: you can improve your goodness of fit by adding extra terms, but usually at the cost of *specialising* your model too much to the training data. This will usually result in poor predictive abilities for the model outside the training region. Which isn't cool.

We can plot now the cost function, as well as our first rough guess and the final guess:

```
# Plotting!
plt.figure(figsize=(15, 5))
# Set up the x and y axis for the plot
yy = np.linspace(1e-5, 20e-5, 20)
xx = np.linspace(1e-5, 10e-2, 10)
# Do a contour plot. The logspace bit basically defines the location
# of 20 contour lines
c = plt.contourf(yy, xx, sos, np.logspace(3, 5, 20), cmap=plt.cm.magma_r)
# Colorbar
plt.colorbar()
# Now, just plot the rough guess of a0 and a1 into this plot
# We want to plot an empty circle with a green edge
```

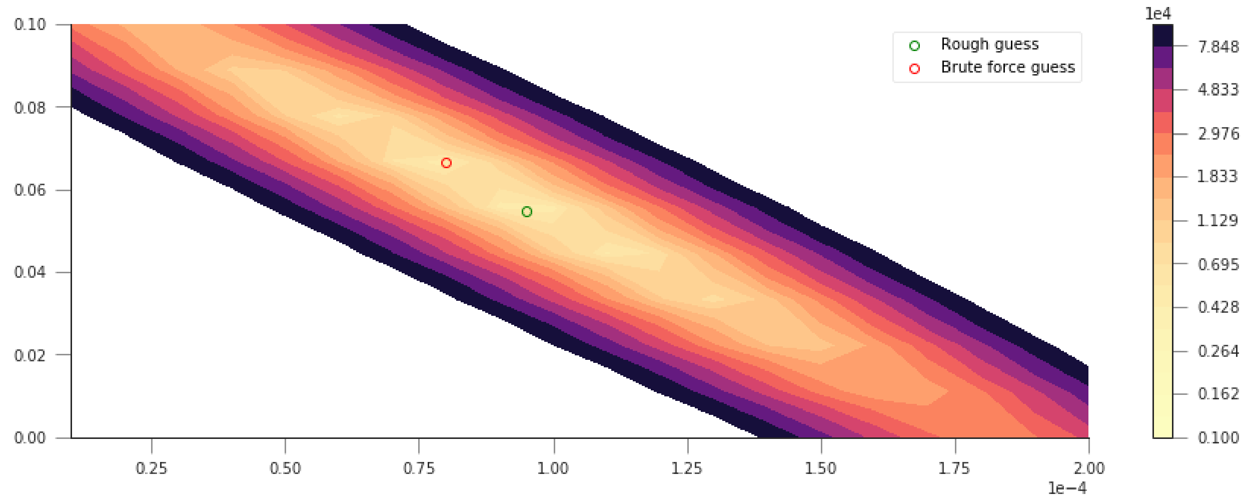
(continues on next page)

(continued from previous page)

```
plt.plot(
    9.51242659e-05,
    5.47960536e-02,
    'o',
    mfc="None",
    mec="g",
    label="Rough guess")
plt.plot(yy_opt, xx_opt, 'o', mfc="None", mec="r", label="Brute force guess")
plt.legend(loc="best")
```

```
<matplotlib.legend.Legend at 0x7f00421ad470>
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



That's not *too bad*! But althogh we found a minimum, we haven't shown how well our model really fits the observations! Let's plot the prediction (with the "optimised parameters" as well as the roughly guessed ones):

```
fig, axs = plt.subplots(nrows=2, ncols=1, figsize=(15, 4))

a2 = 315.
axs[0].plot(
    x,
    quadratic_model([9.51242659e-05, 5.47960536e-02, a2], x),
    '-',
    label="Rough guess")
axs[0].plot(
    x, quadratic_model([yy_opt, xx_opt, a2], x), '-', label="Brute force")

axs[0].plot(x, co2.interpolated, '-', label="Measured")
axs[0].legend(loc="best")

axs[1].plot(
    x,
    co2.interpolated - quadratic_model([yy_opt, xx_opt, a2], x),
    's-',
```

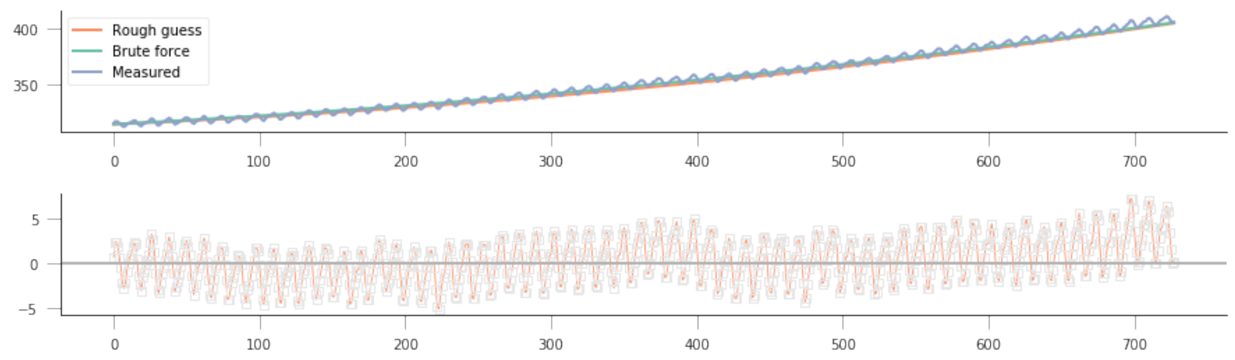
(continues on next page)

(continued from previous page)

```
    lw=0.8,
    mfc="none",
    mec="0.9")
axs[1].axhline(0, color="0.7")
```

```
<matplotlib.lines.Line2D at 0x7f00420cd940>
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



So solvng by brute force with a quadratic appears to have worked better than fitting with our linear model. The residuals now mostly lie in the -5 to 5 units range, whereas the linear model had residuals floating around -12 and 12 or thereabouts. It is also clear that we're missing out on the seasonality, and some rates of growth (particularly at the end) seem to be underemphasised.

Solving the problem using linear algebra

So we can see that our brute force search has given us a better fit than eyeballing it, which is what one might expect. It should be possible to solve this analytically. Let's write this as a matrix problem:

$$\mathbf{A}\cdot\vec{x}=\vec{y}$$

$$\mathbf{A}=\begin{bmatrix} t_1^2 & t_1 & 1\\ t_2^2 & t_2 & 1\\ t_3^2 & t_3 & 1\\ \vdots & \vdots\vdots & \\ t_N^2 & t_N & 1\end{bmatrix}$$

$$\vec{x}=\begin{bmatrix} a_0\\ a_1\\ a_2\end{bmatrix}$$

$$\vec{y}=\begin{bmatrix} W_1\\ W_2\\ W_3\\ \vdots\\ W_N\end{bmatrix}$$

Spend some time satisfying yourself that you understand how the previous matrices and vectors work together.

So, we see that this is really an overdetermined linear problem, where we've got more observations (N) than parameters (3). We can solve this by calculating the pseudo inverse:

$$\vec{x} = \left[\mathbf{A}^{\top}\mathbf{A}\right]^{-1}\mathbf{A}^{\top}\vec{y},$$

where $^{\top}$ is the **transpose**, and $^{-1}$ is the inverse matrix. We can solve this problem easily in Python, which can deal with linear algebra nicely. The `` `np.linalg.lstsq `` <https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.linalg.lstsq.html>`__ method has a direct solver, or you can also work it out by calculating the inverse matrix yourself. The latter approach is usually numerically more unstable, so we won't be looking into it.

Solution using `np.linalg.lstsq`

In this case, we need to define the matrix $\mathbf{A}$. The observations vector $\vec{y}$ is already defined. What is needed is to weed out the invalid measurements in both $\mathbf{A}$ and $\vec{y}$. We then use `` `np.linalg.lstsq `` <https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.linalg.lstsq.html>`__ to solve the linear overdetermined system. This returns a number of things:

1. The solution vector.
2. The sum of squared residuals.
3. The rank of the matrix $\mathbf{A}$.
4. The eigenvalues.

We're really only interested in the first two (the other two outputs are important, but this is not your methods course!).

```
# We create the A matrix
x = np.arange(n_times)
A = np.array([x**2, x, np.ones_like(x)])
# Now put the observations into y
y = co2.interpolated

# Call lstsq
xopt, sum_of_residuals, r, evals = np.linalg.lstsq(A.T, y)
rough_guess = [9.51242659e-05, 5.47960536e-02, 315]
brute_force = [yy_opt, xx_opt, 315]
print("Parameter   Matrix       Brute force     Rough guess")
for par in range(3):
    print("a{}:         {:08.5e}\t {:08.5e}\t {:08.5e}".format(
        par, xopt[par], float(brute_force[par]), rough_guess[par]))
print("Sum of residuals: {:g}".format(float(sum_of_residuals)))
```

```
Parameter   Matrix       Brute force     Rough guess
a0:         8.79184e-05      8.00000e-05     9.51243e-05
a1:         6.53383e-02      6.66700e-02     5.47961e-02
a2:         3.14462e+02      3.15000e+02     3.15000e+02
Sum of residuals: 3563.41
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/ipykernel_
↪launcher.py:8: FutureWarning: rcond parameter will change to the default␣
↪of machine precision times max(M, N) where M and N are the input matrix␣
↪dimensions.
To use the future default and silence this warning we advise to pass␣
↪rcond=None, to keep using the old, explicitly pass rcond=-1.
```

The parameters we got from the linear solver are very similar to the brute force method. If we had used a finer grid in the brute force model, we'd get even closer, but at the price of incresing the number of model evaluations. We can also see that using the analytic least squares solution results in the actual minimum of the cost function, not a value close to it.

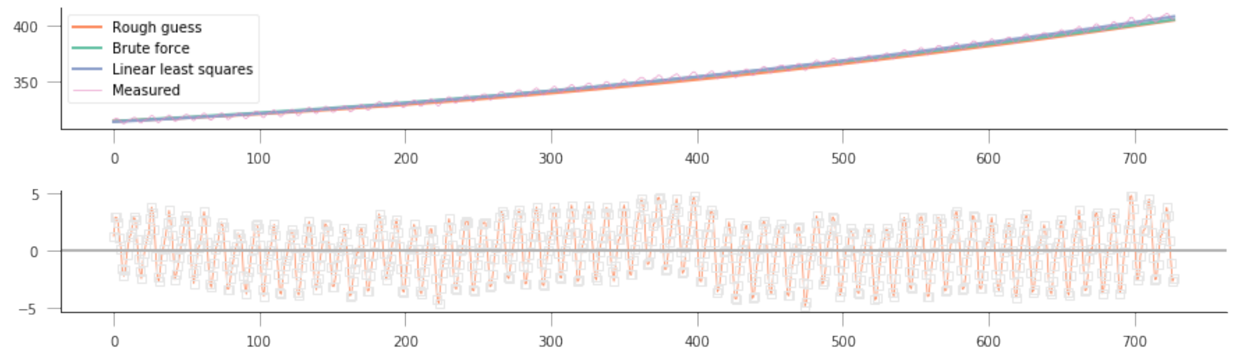
In the linear algebra case, the procedure is very simple, and provided the matrix $\mathbf{A}$ is invertible, one is mostly guaranteed a good solution.

As usual, let's us plot model, data and residuals and see what we can spot…

```
fig, axs = plt.subplots(nrows=2, ncols=1, figsize=(15, 4))
x = np.arange(n_times)
a2 = 315.

axs[0].plot(x, quadratic_model(rough_guess, x), '-', label="Rough guess")
axs[0].plot(
    x, quadratic_model([yy_opt, xx_opt, a2], x), '-', label="Brute force")
axs[0].plot(x, quadratic_model(xopt, x), '-', label="Linear least squares")
axs[0].plot(x, co2.interpolated, '-', lw=0.6, label="Measured")
axs[0].legend(loc="best")

axs[1].plot(
    x,
    co2.interpolated - quadratic_model(xopt, x),
    's-',
    lw=0.8,
    mfc="none",
    mec="0.9")
axs[1].axhline(0, color="0.7")
```

```
<matplotlib.lines.Line2D at 0x7f00401834a8>
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



So we can see that the optimal value is quite similar to the other two solutions, but results in a better fit (3470 versus 4090). We can see that with this method we can solve for all three parameters, even though our first guess of 315 for a_2 was pretty close to the true solution.

Try to use the linear least squares method to fit the first order linear model that we fitted "by eye" at the start of the notebook.

A model with seasonality

While the quadratic model appears to go through the centre of the Mauna Loa curve, it clearly misses an important feature: the seasonality of the CO_2 concentration. We can't really emulate that behaviour with a simple quadratic function, but need a different model, one that deals with the seasonality. We can think that the seasonality is an additive cosine term, so that our model for CO_2 concentration is now

$$W_i = a_0 \cdot t_i^2 + a_1 \cdot t_i + a_2 + a_3 \cdot \cos\left(2\pi\frac{t_i}{T}\right),$$

where T is the period of the seasonality, in this case, annual so $T = 12$.

Although the model looks quite ugly, we see that we can write it like a sum (or a *linear combination*) of some functions (t^2, t,, the cosine term) weighted by the model parameters $a_0, \cdots, a_3$. So this is a linear model like the ones we've seen before and with which you should be familiar.

In this case, the $\mathbf{A}$ matrix is now given by

$$\mathbf{A} = \begin{bmatrix} t_1^2 & t_1 & 1 & \cos\left(2\pi\frac{t_1}{T}\right) \\ t_2^2 & t_2 & 1 & \cos\left(2\pi\frac{t_2}{T}\right) \\ t_3^2 & t_3 & 1 & \cos\left(2\pi\frac{t_3}{T}\right) \\ \vdots & \vdots & \vdots & \vdots \\ t_N^2 & t_N & 1 & \cos\left(2\pi\frac{t_N}{T}\right) \end{bmatrix}.$$

We can still solve the problem by making use of `lstsq`. Let's see how that works!

```python
def quadratic_with_season(p, t, period=12.):
    a0, a1, a2, a3 = p
    return a0 * t * t + a1 * t + a2 + a3 * np.cos(2 * np.pi * (t / period))


period = 12.
# We create the A matrix
x = np.arange(n_times)
A = np.array([x * x, x, np.ones_like(x),
              np.cos(2 * np.pi * (x / period))])
# Now put the observations into y
y = co2.interpolated

# Call lstsq
xopt, sum_of_residuals, r, evals = np.linalg.lstsq(A.T, y)
for par in range(4):
    print("a{}:         {:08.5e}".format(par, xopt[par]))
print(f"Sum of squares: {float(sum_of_residuals):g}")
```

```
a0:         8.80710e-05
a1:         6.52761e-02
a2:         3.14461e+02
a3:         2.28764e+00
Sum of squares: 1654.67
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/ipykernel_
↪launcher.py:15: FutureWarning: rcond parameter will change to the default␣
↪of machine precision times max(M, N) where M and N are the input matrix␣
↪dimensions.
To use the future default and silence this warning we advise to pass␣
↪rcond=None, to keep using the old, explicitly pass rcond=-1.
  from ipykernel import kernelapp as app
```

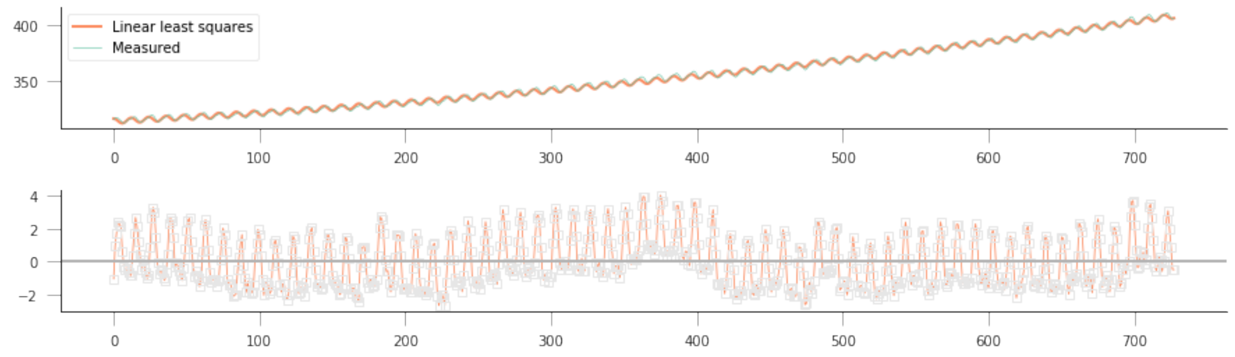
Let's do some plots of the function fitting and residuals, and compare to previous results…

Doing these sort of plots should be second nature to you by now. So do them!

```
fig, axs = plt.subplots(nrows=2, ncols=1, figsize=(15, 4))
x = np.arange(n_times)
a2 = 315.

axs[0].plot(x, quadratic_with_season(xopt, x), '-', label="Linear least squares")
axs[0].plot(x, co2.interpolated, '-', lw=0.6, label="Measured")
axs[0].legend(loc="best")

axs[1].plot(
    x,
    co2.interpolated - quadratic_with_season(xopt, x),
    's-',
    lw=0.8,
    mfc="none",
    mec="0.9")
axs[1].axhline(0, color="0.7")
```

```
<matplotlib.lines.Line2D at 0x7f00400fb4a8>
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



So that's pretty good: by adding a simple cosine term, we can now start to model the annual seasonality in the measurements, and the sum of squared residuals is now further shrunk to around 1500. This is good, but in some ways unsurprising: you're now solving for 4 parameters, rather than 3 or 2 (for the simple linear case), so you have more degrees of freedom, and you expect to be able to fit your data better.

A phase shift

Looking at the residuals, we might decide that there's some mileage in shifting the cosine term a bit to get a better fit. We could do this by adding a phase shift so that the cosine terms would look like

$$\cos\left[\frac{2\pi}{T}(t+\phi)\right]$$

However, it'd be hard to guess ϕ (we've effectively assumed it was 0 radians above!). So we'd need to use some non-linear solving approach. However, we might exploit the following trigonometrical identity:

$$A\cos(\theta) + B\sin(\theta) = C\sin(\theta+\phi),$$

Can you prove the above identity?

This means that we can just add (drumroll. . .) yet another term to our model (a sine term), and the ratio of the cosine and sine terms will result in a phase shift. As we're adding another term, we expect a better result, but in this case, we hope that the aim of adding this extra term is to have **uncorrelated residuals around 0**.

You should be able to do this yourself, including model fitting and plotting.

```
def quadratic_with_season_shift(p, t, period=12.):
    a0, a1, a2, a3, a4 = p
    return a0 * t * t + a1 * t + a2 + \
                a3 * np.cos(2 * np.pi * (t / period)) + \
                a4 * np.sin(2 * np.pi * (t / period))



period = 12.
# We create the A matrix
x = np.arange(n_times)
A = np.array([x * x, x, np.ones_like(x),
              np.cos(2 * np.pi * (x / period)),
              np.sin(2 * np.pi * (x / period))])
# Now put the observations into y
y = co2.interpolated

# Call lstsq
xopt, sum_of_residuals, r, evals = np.linalg.lstsq(A.T, y)
for par in range(5):
    print("a{}:          {:08.5e}".format(par, xopt[par]))
print(f"Sum of squares: {float(sum_of_residuals):g}")
```

```
a0:         8.76557e-05
a1:         6.55875e-02
a2:         3.14413e+02
a3:         2.28558e+00
a4:         1.66696e+00
Sum of squares: 645.521
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/ipykernel_
↪launcher.py:17: FutureWarning: rcond parameter will change to the default␣
↪of machine precision times max(M, N) where M and N are the input matrix␣
↪dimensions.
To use the future default and silence this warning we advise to pass␣
↪rcond=None, to keep using the old, explicitly pass rcond=-1.
```

The squared sum of residuals is now around 645, again an improvement on the solution. We can see the fit and residuals I got (yours should be similar) here

We're now within the +/- 2 units band, which is a reasonable estimate.

Prediction

A model isn't very good if you don't challenge it to predict phenomean outside the training range. We could just extend the $x-$ axis further left or right and see what the model predicts, but we'd be **extrapolating**.

Given the simplicity of the model, and what you know about CO_2 dynamics over the past ~100 years, would you trust these extrapolations?

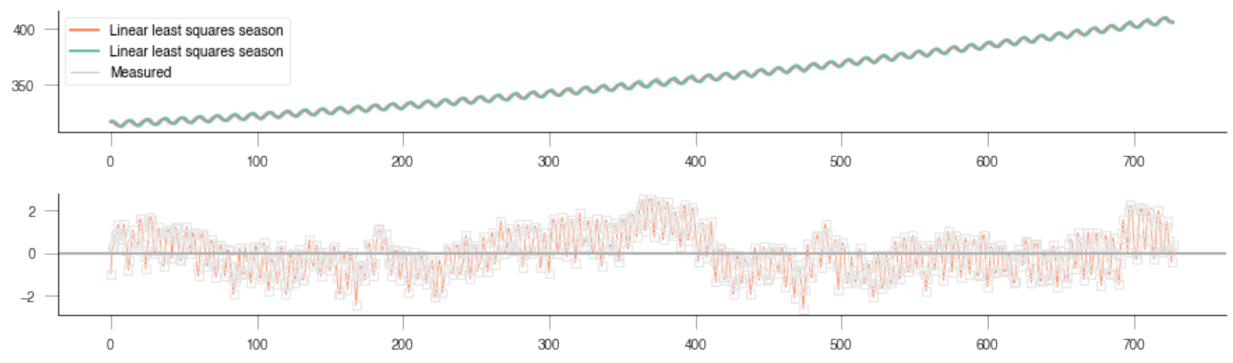


Fig. 20: quadratic with seasonal shift

We can sort of mimic this behaviour by fitting the model only to a subset of years, and then test it for the rest of the available time series. For example, fit for the first 20 years, and then forecasat the remaining years. Or fit the last 20 years and forecast the previous years until the 1950s… If the model is successful in its predictions, then we can say that the model is probably OK, but if the quality of the predictions is poor, then we need to start thinking about **discarding** the model, and looking for alternatives!

Fit the model to the first 30 years of data, and then use it to predict the complete time series. Your result should look something like below. Can you explain what's going on there?

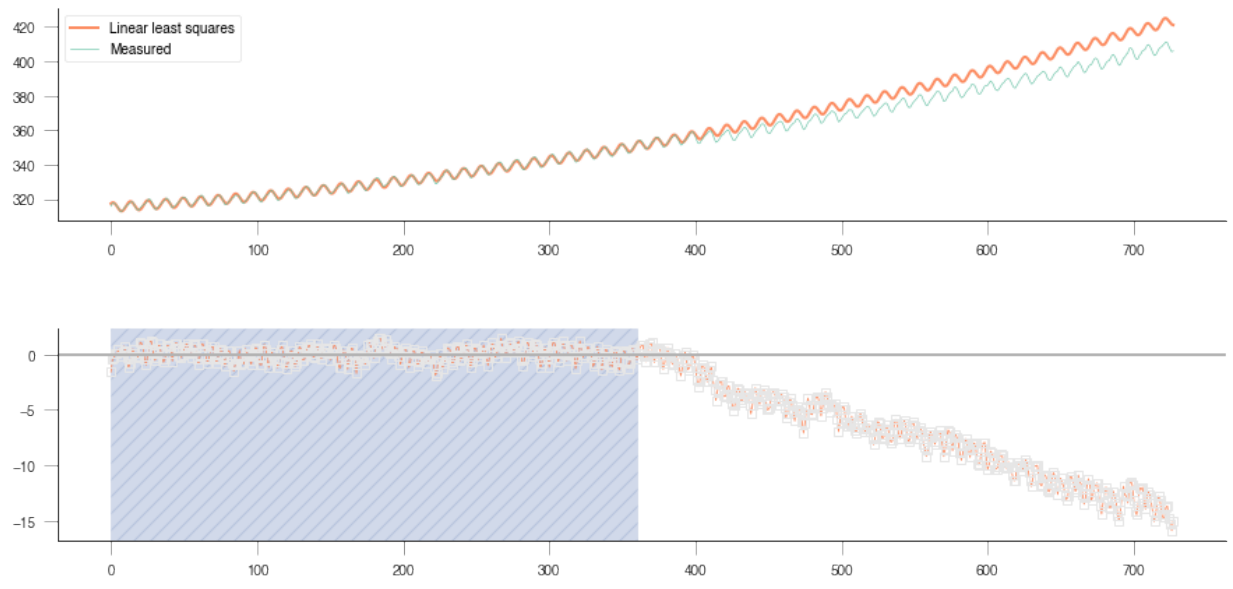


Fig. 21: Extrapolation plot

Uncertainty

We have not said anything about how the model predictions are *uncertain*: we only used a limited dataset, with measurements errors associated to it. Even within the training period, the residuals are not 0, so we can expect that the model has some bits of reality missing from it (it *is* a model, after all!). Uncertainty would allow us to quantify how good or bad the predictions from the model are, but so far, we have ignored it…

In the 30 year training experiment, can you sketch how you think uncertainty should look like?

```
def quadratic_with_season_shift(p, t, period=12.):
    a0, a1, a2, a3, a4 = p
    return a0 * t * t + a1 * t + a2 + \
                a3 * np.cos(2 * np.pi * (t / period)) + \
                a4 * np.sin(2 * np.pi * (t / period))



period = 12.
# We create the A matrix
x = np.arange(n_times)[:12*30]
A = np.array([x * x, x, np.ones_like(x),
              np.cos(2 * np.pi * (x / period)),
              np.sin(2 * np.pi * (x / period))])
# Now put the observations into y
y = co2.interpolated[:12*30]

# Call lstsq
xopt, sum_of_residuals, r, evals = np.linalg.lstsq(A.T, y)
for par in range(5):
    print("a{}:         {:08.5e}".format(par, xopt[par]))
print(f"Sum of squares: {float(sum_of_residuals):g}")
```

```
a0:         1.37489e-04
a1:         4.93777e-02
a2:         3.15199e+02
a3:         2.10176e+00
a4:         1.70324e+00
Sum of squares: 166.009
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/ipykernel_
↪launcher.py:17: FutureWarning: rcond parameter will change to the default␣
↪of machine precision times max(M, N) where M and N are the input matrix␣
↪dimensions.
To use the future default and silence this warning we advise to pass␣
↪rcond=None, to keep using the old, explicitly pass rcond=-1.
```

2.8.22 5 Modelling and optimisation

Table of Contents

5.1 Introduction

In this sections, we will build some models to describe environmental processes. We wil then use observational data to calibrate and test these models.

5.2 Get datasets

We first get the datasets we will need.

These are the MODIS LAI and land cover data and associated ECMWF temperature data. Datasets are available in `npz` files that we have previously generated.

```
# required general imports
import matplotlib.pyplot as plt
import cartopy.crs as ccrs
%matplotlib inline
import numpy as np
import sys
import os
from pathlib import Path
import gdal
from datetime import datetime, timedelta
from geog0111.geog_data import procure_dataset
```

```
# conditions
year = 2016
country_code = 'UK'
```

```
'''
Load the prepared LAI data
'''
# read in the LAI data for given country code
lai_filename = f'data/lai_data_{year}_{country_code}.npz'
# get the dataset in case its not here
procure_dataset(Path(lai_filename).name,verbose=False)

lai_data = np.load(lai_filename)
print(lai_filename,list(lai_data.keys()))

# unload for use
dates, lai, weights, interpolated_lai = lai_data['dates'],lai_data['lai'],\
                            lai_data['weights'],lai_data['interpolated_lai']
lai[weights==0.] = np.nan

print(lai.shape)
```

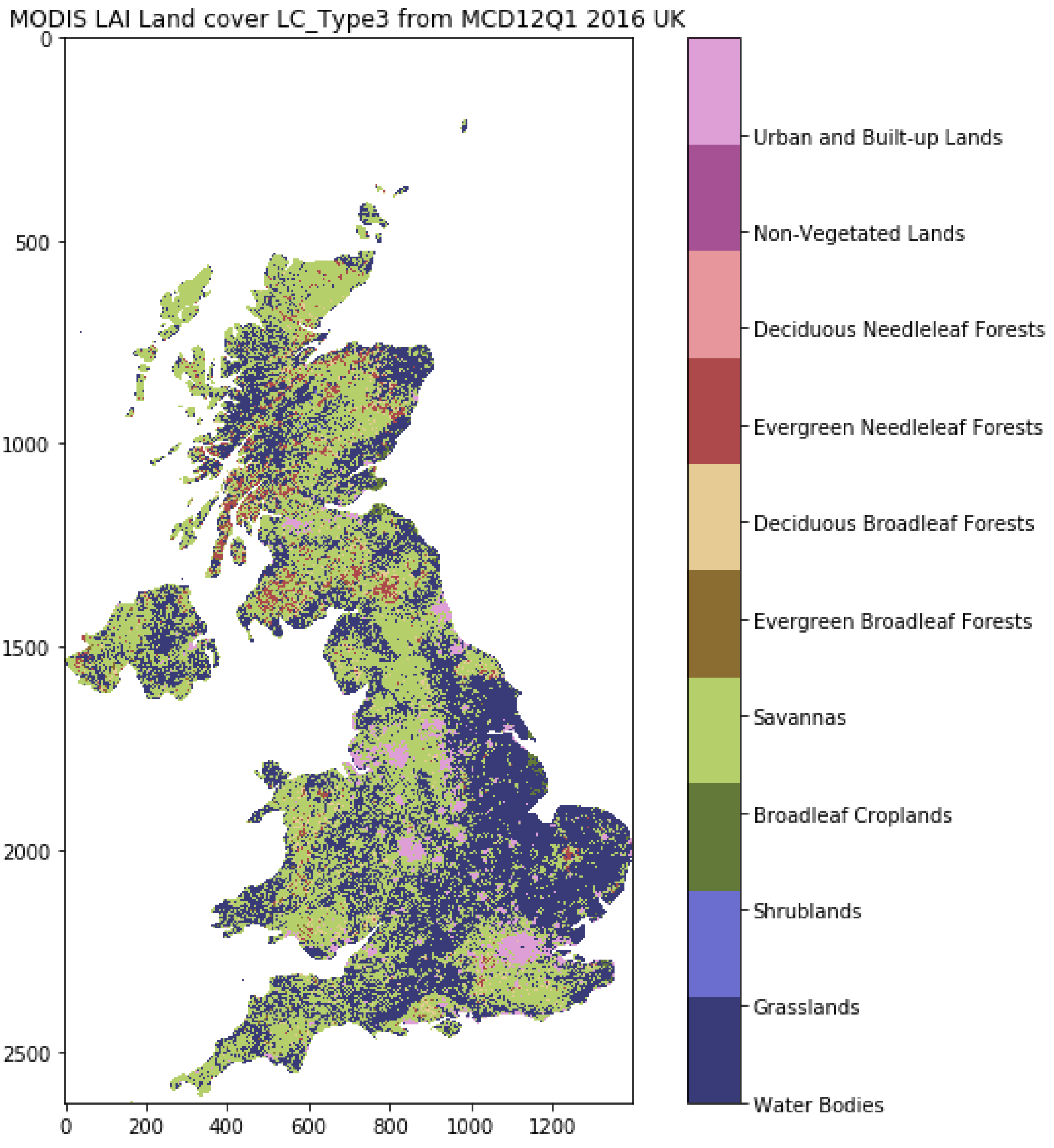
```
data/lai_data_2016_UK.npz ['dates', 'lai', 'weights', 'interpolated_lai']
(2624, 1396, 92)
```

Recall that land cover is interpreted as:

Name	Value	Description
Water Bodies	0	At least 60% of area is covered by permanent water bodies.
Grasslands	1	Dominated by herbaceous annuals (<2m) includ- ing cereal croplands.
Shrublands	2	Shrub (1-2m) cover >10%.
Broadleaf Croplands	3	Dominated by herbaceous annuals (<2m) that are cultivated with broadleaf crops.
Savannas	4	Between 10-60% tree cover (>2m).
Evergreen Broadleaf Forests	5	Dominated by evergreen broadleaf and palmate trees (>2m). Tree cover >60%.
Deciduous Broadleaf Forests	6	Dominated by deciduous broadleaf trees (>2m). Tree cover >60%.
Evergreen Needle-leaf Forests	7	Dominated by evergreen conifer trees (>2m). Tree cover >60%.
Deciduous Needle-leaf Forests	8	Dominated by deciduous needleleaf (larch) trees (>2m). Tree cover >60%.
Non-Vegetated Lands	9	At least 60% of area is non-vegetated barren (sand, rock, soil) or permanent snow and ice with less than 10% vegetation.
Urban and Built-up Lands	10	At least 30% impervious surface area including building materials, asphalt, and vehicles.
Unclassified	255	Has not received a map label because of missing inputs.

```
'''
Load the prepared landcover data
'''
# read in the LAI data for given country code
lc_filename = f'data/landcover_{year}_{country_code}.npz'
# get the dataset in case its not here
procure_dataset(Path(lc_filename).name,verbose=False)

lc_data = np.load(lc_filename)
print(lc_filename,list(lc_data.keys()))

# unload for use
LC_Type3, lc_data = lc_data['LC_Type3'],lc_data['lc_data']

from geog0111.plot_landcover import plot_land_cover
print(plot_land_cover(lc_data,year,country_code))
print(lc_data.shape)
```

```
data/landcover_2016_UK.npz ['LC_Type3', 'lc_data']
['Water Bodies' 'Grasslands' 'Shrublands' 'Broadleaf Croplands' 'Savannas'
 'Evergreen Broadleaf Forests' 'Deciduous Broadleaf Forests'
 'Evergreen Needleleaf Forests' 'Deciduous Needleleaf Forests'
 'Non-Vegetated Lands' 'Urban and Built-up Lands']
(2624, 1396)
```


MODIS LAI Land cover LC_Type3 from MCD12Q1 2016 UK
0
500
1000
1500
2000
2500
0
200
400
600
800
1000
1200
Urban and Built-up Lands
Non-Vegetated Lands
Deciduous Needleleaf Forests
Evergreen Needleleaf Forests
Deciduous Broadleaf Forests
Evergreen Broadleaf Forests
Savannas
Broadleaf Croplands
Shrublands
Grasslands
Water Bodies

```
'''
Load the prepared T 2m data
'''
t2_filename = f'data/europe_data_{year}_{country_code}.npz'
# get the dataset in case its not here
procure_dataset(Path(t2_filename).name,verbose=False)
t2data = np.load(t2_filename)
print(t2_filename,list(t2data.keys()))

timer, temp2, extent = t2data['timer'], t2data['temp2'], t2data['extent']
print(temp2.shape)
```

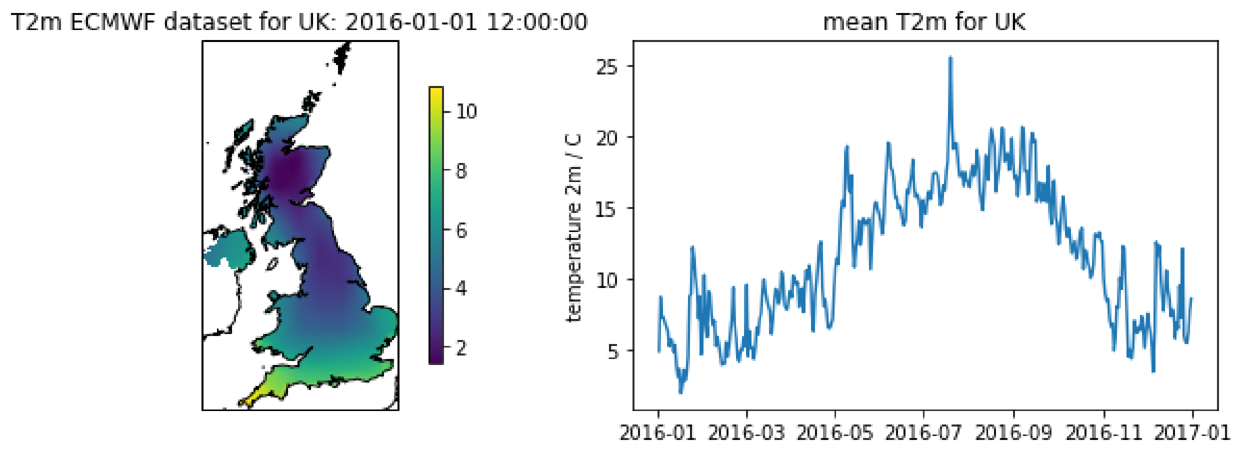
```
data/europe_data_2016_UK.npz ['timer', 'temp2', 'extent']
(366, 2624, 1396)
```

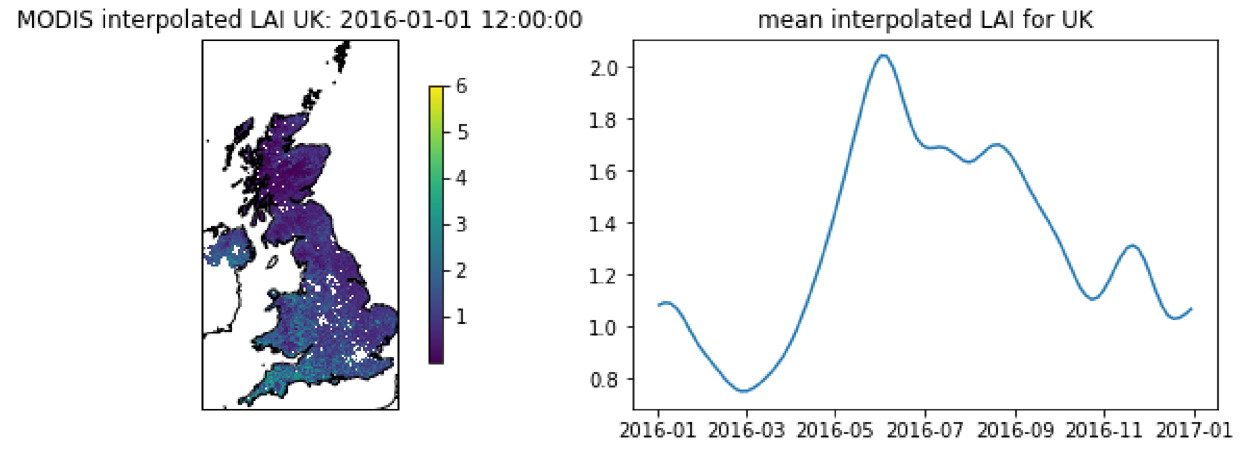
Now let's plot the datasets:

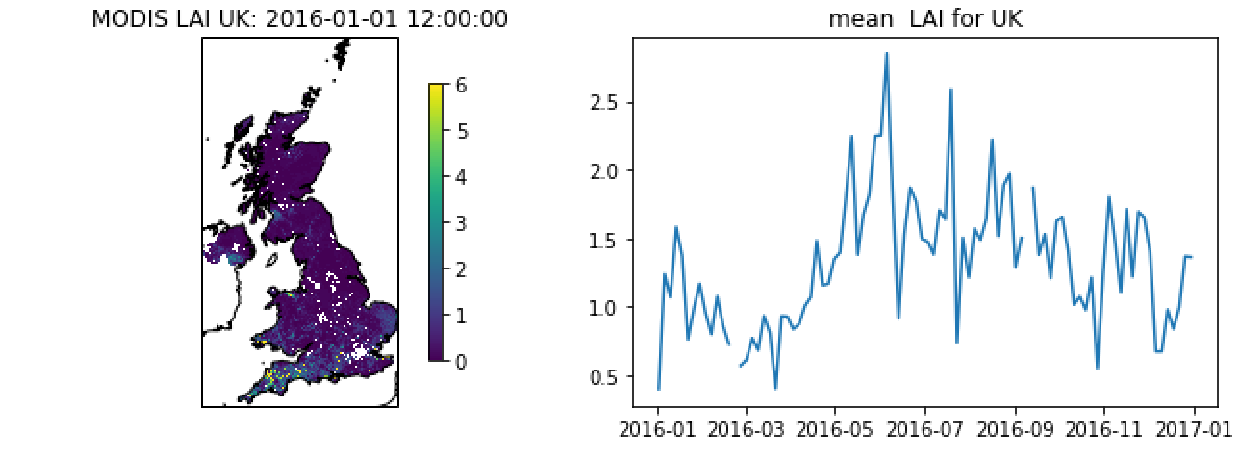
```
# visualise the interpolated dataset
import matplotlib.pylab as plt
import cartopy.crs as ccrs
%matplotlib inline

plt.figure(figsize=(12,12))
ax = plt.subplot ( 3, 2, 1 ,projection=ccrs.Sinusoidal.MODIS)
ax.coastlines('10m')
ax.set_title(f'T2m ECMWF dataset for {country_code}: {str(t2data["timer"][0])}')
im = ax.imshow(temp2[0],extent=extent)
plt.colorbar(im,shrink=0.75)

ax = plt.subplot ( 3, 2, 3 ,projection=ccrs.Sinusoidal.MODIS)
ax.coastlines('10m')
ax.set_title(f'MODIS interpolated LAI {country_code}: {str(t2data["timer"][0])}')
im = plt.imshow(interpolated_lai[:,:,0],vmax=6,extent=extent)
plt.colorbar(im,shrink=0.75)

ax = plt.subplot ( 3, 2, 5 ,projection=ccrs.Sinusoidal.MODIS)
ax.coastlines('10m')
ax.set_title(f'MODIS LAI {country_code}: {str(t2data["timer"][0])}')
im = plt.imshow(lai[:,:,0],vmax=6,extent=extent)
plt.colorbar(im,shrink=0.75)



plt.subplot ( 3, 2, 2 )
plt.title(f'mean T2m for {country_code}')
plt.plot(timer,np.nanmean(temp2,axis=(1,2)))
plt.ylabel('temperature 2m / C')
plt.subplot ( 3,2, 4 )
plt.title(f'mean interpolated LAI for {country_code}')
mean = np.nanmean(interpolated_lai,axis=(0,1))
plt.plot(timer[::4],mean)
plt.subplot ( 3,2, 6 )
plt.title(f'mean  LAI for {country_code}')
mean = np.nanmean(lai,axis=(0,1))
plt.plot(timer[::4],mean)
```

```
/Users/plewis/anaconda/envs/geog0111/lib/python3.6/site-packages/ipykernel_launcher.
↪py:37: RuntimeWarning: Mean of empty slice
```

```
[<matplotlib.lines.Line2D at 0x129b90e10>]
```







5.3 Interpretation of the data

We can see that the raw LAI temporal profile (bottom right plot) can be very noisy, even when averaged spatially.

The 'true' temporal profile is probably much better represented in the 'interpolated LAI' dataset, although this may be ober-smoothed.

From the interpolated dataset, we see that the LAI trajectory 'takes off' in the Spring (March/April), and 'falls' in the Autumn (October/November), which is the pattern we would expect of Western European vegetation. There is some evidence of multiple 'peaks' in the higher LAI values, which is suggestive of the signal being a compound of thebehaviour of multiple vegetation types.

The periods of rapid change in LAI correspond to when the mean (2m) temperature is around 10 C.

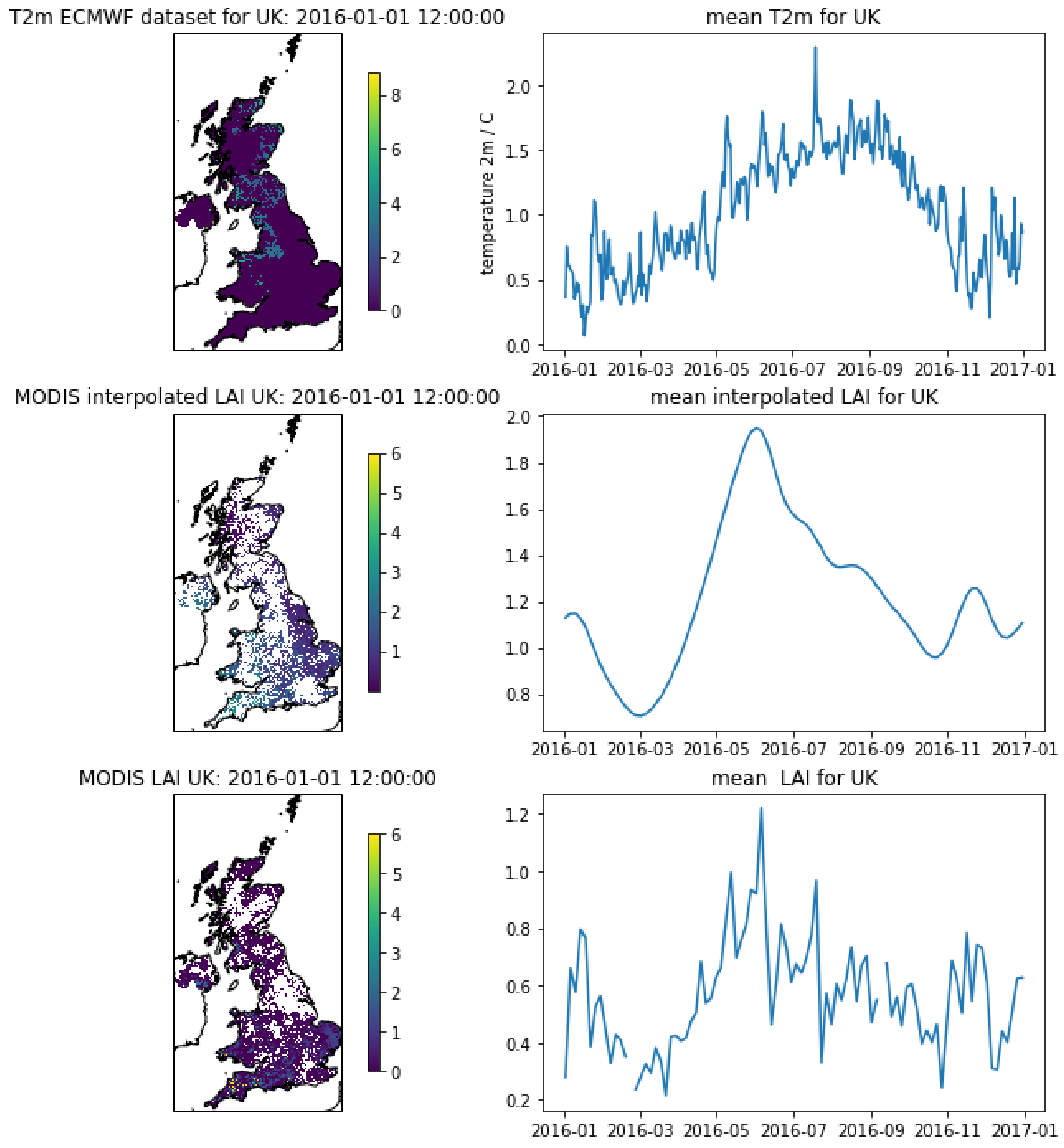
Now let's look at a particular land cover type: grasslands.

```
lc = 1

# need 2 versions of this as datasets
# have time stacked differently
flc_data1 = lc_data[...,np.newaxis]
flc_data2 = lc_data[np.newaxis,...]

for d in [flc_data1,flc_data2]:
    mask = d==lc
    d[mask]  = 1
    d[~mask] = 0
```

```
'''
filter datasets by land cover
'''
interpolated_lai_ = interpolated_lai*flc_data1
interpolated_lai_[interpolated_lai_==0] = np.nan
lai_ = lai*flc_data1
lai_[lai==0] = np.nan
temp2_ = temp2*flc_data2
temp2_[temp2==0] = np.nan

plt.figure(figsize=(12,12))
ax = plt.subplot ( 3, 2, 1 ,projection=ccrs.Sinusoidal.MODIS)
ax.coastlines('10m')
ax.set_title(f'T2m ECMWF dataset for {country_code}: {str(t2data["timer"][0])}')
im = ax.imshow((temp2_)[0],extent=extent)
plt.colorbar(im,shrink=0.75)

ax = plt.subplot ( 3, 2, 3 ,projection=ccrs.Sinusoidal.MODIS)
ax.coastlines('10m')
ax.set_title(f'MODIS interpolated LAI {country_code}: {str(t2data["timer"][0])}')
im = plt.imshow(interpolated_lai_[:,:,0],vmax=6,extent=extent)
plt.colorbar(im,shrink=0.75)

ax = plt.subplot ( 3, 2, 5 ,projection=ccrs.Sinusoidal.MODIS)
ax.coastlines('10m')
ax.set_title(f'MODIS LAI {country_code}: {str(t2data["timer"][0])}')
im = plt.imshow((lai_)[:,:,0],vmax=6,extent=extent)
plt.colorbar(im,shrink=0.75)

plt.subplot ( 3, 2, 2 )
plt.title(f'mean T2m for {country_code}')
plt.plot(timer,np.nanmean(temp2_,axis=(1,2)))
plt.ylabel('temperature 2m / C')
plt.subplot ( 3,2, 4 )
plt.title(f'mean interpolated LAI for {country_code}')
mean = np.nanmean(interpolated_lai_,axis=(0,1))
plt.plot(timer[::4],mean)
plt.subplot ( 3,2, 6 )
plt.title(f'mean  LAI for {country_code}')
mean = np.nanmean(lai_,axis=(0,1))
plt.plot(timer[::4],mean)
```

```
/Users/plewis/anaconda/envs/geog0111/lib/python3.6/site-packages/ipykernel_launcher.
↪py:39: RuntimeWarning: Mean of empty slice
```

```
[<matplotlib.lines.Line2D at 0x130ef5438>]
```



```python
# All imports go here. Run me first!
import datetime
from pathlib import Path  # Checks for files and so on
import numpy as np  # Numpy for arrays and so on
import pandas as pd
```

(continues on next page)

(continued from previous page)

```
import sys
import matplotlib.pyplot as plt  # Matplotlib for plotting
# Ensure the plots are shown in the notebook
%matplotlib inline

import gdal
import osr
import numpy as np
%matplotlib inline
```

2.8.23 Fitting non-linear models

In the previous session, we looked at fitting linear models to observations. While this is a very common task, complex processes might require models which are non-linear.

Can you think of any non-linear models that you have come across?

One non-linear model is modelling LAI as a function of time (or temperature). In the Northern Hemisphere, and for temperate latitudes, there is a clear seasonal cycle in vegetation, particularly visible in leaf area index (LAI). LAI dynamics can possibly be depicted by a "double logistic" curve. Mathematically, the double logistic looks like this

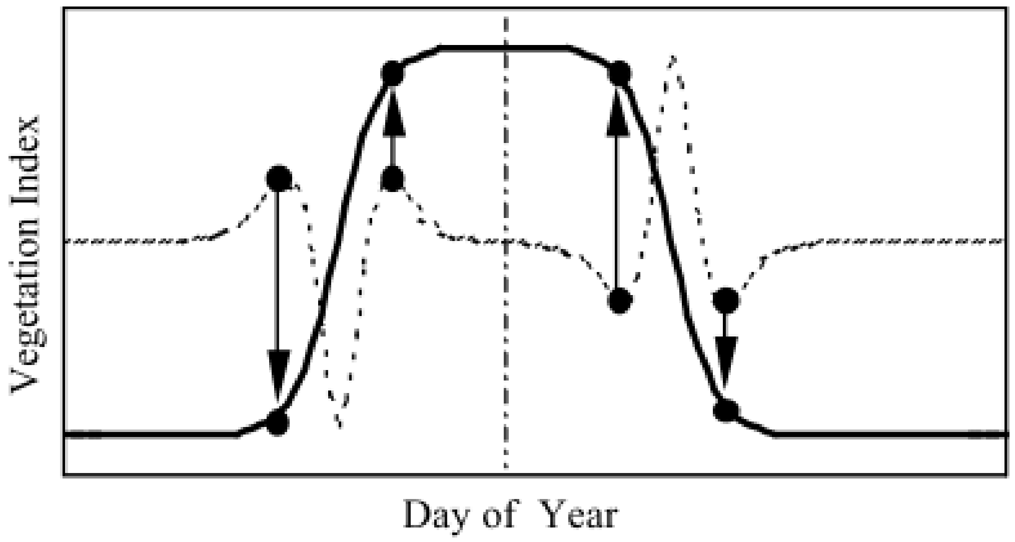


Fig. 2. A schematic showing how transition dates are calculated using minimum and maximum values in the rate of change in curvature. The solid line is an idealized time series of vegetation index data, and the dashed line is the rate of change in curvature from the VI data. The circles indicate transition dates. The extreme values located between each circle indicate the point at which the rate of change in curvature changes sign.

Fig. 22: A double logistic

Mathematically, the function predicts the e.g. LAI (or some vegetation index) as

$$y = p_0 - p_1 \cdot \left[\frac{1}{1+\exp\left(p_2\cdot(t-p_3)\right)} + \frac{1}{1+\exp\left(-p_4\cdot(t-p_5)\right)} - 1\right].$$

If we inspect this form, we can probably guess that p_0 and p_1 scale the vertical span of the function, whereas p_3 and p_5 are some sort of temporal shift, and the remaining parameters p_2 and p_4 control the slope of the two flanks. Something that will give rise to a self-respecting LAI curve might be

- $p_0 = 0.1$
- $p_1 = 2.5$
- $p_2 = 0.19$
- $p_3 = 120$
- $p_4 = 0.13$
- $p_5 = 220$

Write a function that produces the double logistic when passed an array of time steps (e.g. 1 to 365), and an array with six parameters.

Do some plots and try to get some intuition on the model parameters!

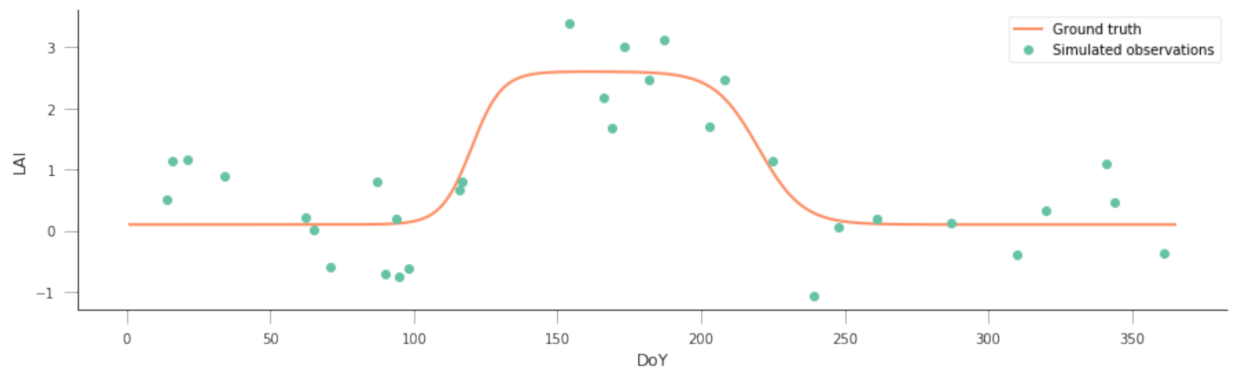
A synthetic experiment

A first step is to do a synthetic experiment. This has the marked advantage of being a situation where we're in control of everything.

```
t = np.arange(1, 366)
p = np.array([0.1, 2.5, 0.19, 120, 0.13, 220])
y = dbl_sigmoid_function(p, t)
yn = y + np.random.randn(len(t))*0.6

selector = np.random.rand(365)

passer = np.where(selector > 0.9, True, False)

tn = t[passer]
yn = yn[passer]

fig = plt.figure(figsize=(15, 4))
_ = plt.plot(t, y, '-', label="Ground truth")
_ = plt.plot(tn, yn, 'o', label="Simulated observations")
plt.legend(loc="best")
plt.xlabel("DoY")
plt.ylabel("LAI")
```

```
Text(0,0.5,'LAI')
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



We know that the "true parameters" are given by `p = np.array([0.1, 2.5, 0.19, 120, 0.13, 220])`, but we see that the data is quite noisy and has significant gaps. As per last session, we could try to modify the parameters "by hand", and see how far we get, but given that it's 6, with different ranges, it looks a bit daunting. Also, we'd need to assess how good the solution is for a particular set of parameters, in other words, select a metric to quantify the goodness of fit.

It is useful to consider a model of the incomplete, noisy observations of LAI (y_n) and the true value of LAI, y. For overlapping time steps, the noisy data are just the "true" data plus some random Gaussian value with zero mean and a given variance σ^2_{obs} (in the experiment above, $\sigma_{obs} = 0.6$):

$$y^i_n = y^i + \mathcal{N}(0, \sigma^2_{obs}).$$

Rearranging things, we have that $y_n - y$ should be a zero mean Gaussian distribution with known variance. We have assumed that our model is $f(\vec{p}) = y$, so we can write the *likelihood function*, $l(\vec{p})$

$$l(\vec{p}) = \left[\frac{1}{\sqrt{2\pi\sigma^2_{obs}}}\right]^N \prod_{i=1}^{N} \exp\left[-\frac{(y^i_n - f(\vec{p})^i)^2}{2\sigma^2_{obs}}\right].$$

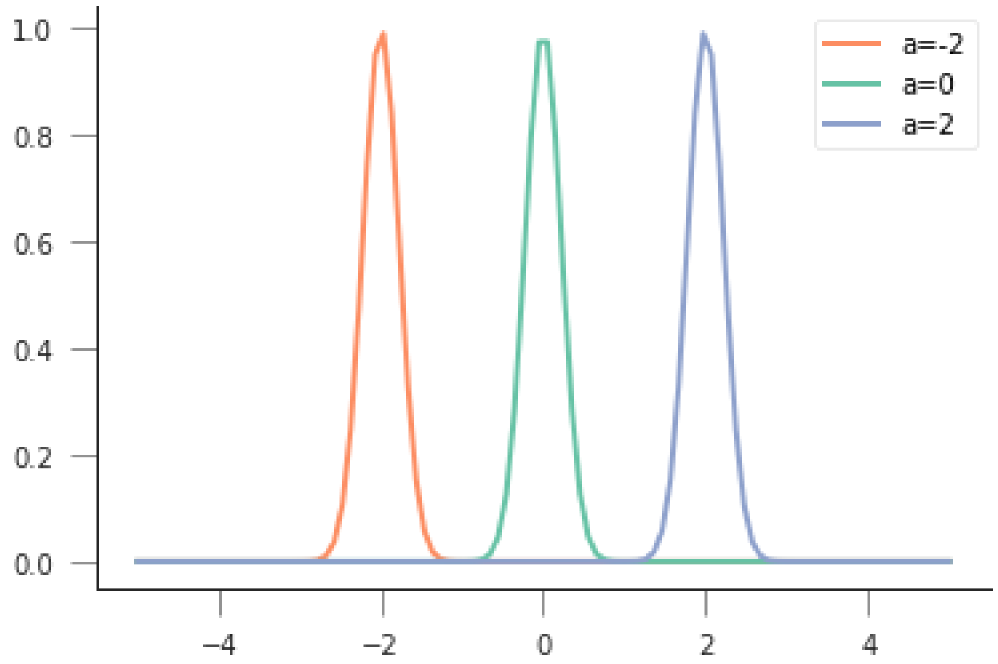
It is convenient to take a logarithm of $l(\vec{p})$, so that we have the **log-likelihood**:

$$L(\vec{p}) = -\sum_{i=1}^{N}\left[\frac{(y^i_n - f(\vec{p})^i)^2}{2\sigma^2_{obs}}\right] + \text{Const.}$$

Think about the likelihood and log-likelihood... Think (and possibly plot) how a negative exponential curve looks like, and what conditions are for some interesting points.

```
<matplotlib.legend.Legend at 0x7fdd3698d358>
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



So for a sum of squares, the most likely result would be if all the mismatches were zero, which means that the log-likelihood is 0, and the likelihood, $exp(0) = 1$!

However, the mismatch might not be 0, due to the added noise. So what we're effectively looking for is a **maximum** in the log-likelihood, or a **minimum** of its negative as a function of $\vec{p}$:

$$\frac{\partial(-L(\vec{p}))}{\partial\vec{p}} \triangleq \min$$

So, we can try our brute force guessing approach by **minimising the cost function given by :math:'L(vec{p})'**

Write the cost function! Test it possibly shifting one parameter over some range of values

The easiest way to obtain the solution is to use numerical optimisation techniques to minimise the cost function. In scipy, there's a good selection of function optimisers. We'll be looking at **local** optimisers: these will look for a minimum in the vicinity of a user-given starting point, usually by looking at the gradient of the cost function. The main function to consider here is `` `minimise `` <https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html#scipy.optimize.minimize>'__. Basically, `minimize` takes a cost function, a starting point, and maybe extra arguments that are passed to the cost function, and uses one of several algorithms to minimise the cost function. We import it with

```
from scipy.optimize import minimize
```

From the documentation,

```
minimize(fun, x0, args=(), method=None,
         jac=None, hess=None, hessp=None,
         bounds=None, constraints=(), tol=None,
         callback=None, options=None)
```

Basically, `fun` is the name of the cost function. The first parameter you pass to the cost function has to be an array with the parameters that will be used to calculate the cost. `x0` is the starting point. `args` allows you to add extra parameters that are required for the cost function (in our example, these would be `t, y_obs, passer, sigma_obs`).

The `minimize` function returns an object with the

1. Value of the function at the minimum,
2. The value of the input parameters that attain the minimum,
3. A message telling you whether the optimisation succeeded
4. The number of iterations (`nit`) and total function evaluations (`nfev`)
5. Some diagnostics

```
from scipy.optimize import minimize
```

```
from scipy.optimize import minimize

p0 = np.array([0, 5, 0.01, 90, 0.01, 200])

retval = minimize(cost_function, p0, args=(t, yn, passer, 0.6))

print(retval)

print ("********************************************")
if retval.success:
    print("Optimisation successful!")
    print(f"Value of the function at the minimum: {retval.fun:g}")
    print(f"Value of the solution: {str(retval.x):s}")
```

```
      fun: 21.342085853280015
 hess_inv: array([[ 1.72923106e-02, -2.15349312e-02,  2.11636346e-03,
         2.98572159e-02,  4.47526187e-03, -5.30376564e-02],
       [-2.15349312e-02,  1.13986002e-01, -4.73509772e-02,
         5.02075575e-01, -5.17466259e-02, -4.55944527e-01],
       [ 2.11636346e-03, -4.73509772e-02,  7.51102753e-02,
        -5.51690150e-01,  2.61989471e-02,  2.73170740e-01],
       [ 2.98572159e-02,  5.02075575e-01, -5.51690150e-01,
         9.54855368e+00, -3.19209970e-01, -3.30673084e+00],
       [ 4.47526187e-03, -5.17466259e-02,  2.61989471e-02,
        -3.19209970e-01,  5.74465509e-02,  3.44439220e-01],
       [-5.30376564e-02, -4.55944527e-01,  2.73170740e-01,
        -3.30673084e+00,  3.44439220e-01,  8.29858275e+00]])
      jac: array([ 4.76837158e-07,  4.52995300e-06, -4.76837158e-07, -4.
↪76837158e-07,
        9.53674316e-07,  4.76837158e-07])
  message: 'Optimization terminated successfully.'
     nfev: 616
      nit: 63
     njev: 77
   status: 0
  success: True
        x: array([2.15760787e-01, 2.22197385e+00, 2.73744085e-01, 1.
↪23505589e+02,
       2.63305511e-01, 2.22200329e+02])
********************************************
Optimisation successful
Value of the function at the minimum: 21.3421
Value of the solution: [2.15760787e-01 2.22197385e+00 2.73744085e-01 1.
↪23505589e+02
 2.63305511e-01 2.22200329e+02]
```

Do some synthetic experiments. For example:

Change the true parameters and see how the solution tracks the change.

Increase the added variance

Reduce or increase the number of observations

Use these experiments to challenge your understanding of the problem: Try to think what the expected result of these changes is, and write a set of functions that simplify the exploration.

Next: Real data

In the next session, you'll be applying these techniques to MODIS LAI data

```
# All imports go here. Run me first!
import datetime
from pathlib import Path  # Checks for files and so on
import numpy as np  # Numpy for arrays and so on
import pandas as pd
import sys
import matplotlib.pyplot as plt  # Matplotlib for plotting
# Ensure the plots are shown in the notebook
%matplotlib inline

import gdal
import osr
import numpy as np
%matplotlib inline
```

2.8.24 Fitting non-linear models

In the previous session, we looked at fitting linear models to observations. While this is a very common task, complex processes might require models which are non-linear.

Can you think of any non-linear models that you have come across?

One non-linear model is modelling LAI as a function of time (or temperature). In the Nothern Hemisphere, and for temperate latitudes, there is a clear seasonal cycle in vegetation, particularly visible in leaf area index (LAI). LAI dynamics can possibly be depicted by a "double logistic" curve. Mathematically, the double logistic looks like this

Mathematically, the function predicts the e.g. LAI (or some vegetation index) as

$$y = p_0 - p_1 \cdot \left[\frac{1}{1+\exp\left(p_2\cdot(t-p_3)\right)} + \frac{1}{1+\exp\left(-p_4\cdot(t-p_5)\right)} - 1\right].$$

If we inspect this form, we can probably guess that p_0 and p_1 scale the vertical span of the function, whereas p_3 and p_5 are some sort of temporal shift, and the remaining parameters p_2 and p_4 control the slope of the two flanks. Something that will give rise to a self-respecting LAI curve might be

- $p_0 = 0.1$
- $p_1 = 2.5$
- $p_2 = 0.19$
- $p_3 = 120$
- $p_4 = 0.13$

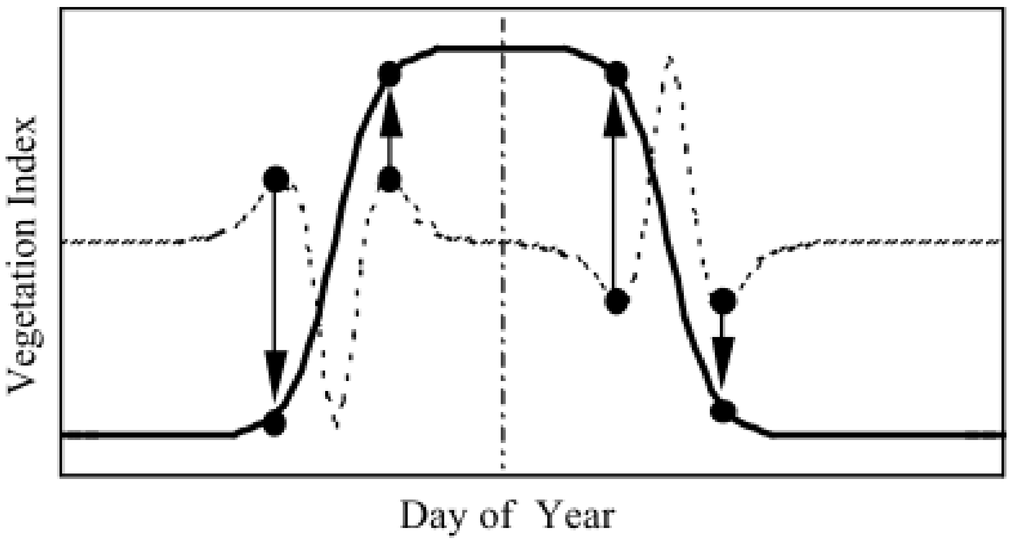


Fig. 2. A schematic showing how transition dates are calculated using minimum and maximum values in the rate of change in curvature. The solid line is an idealized time series of vegetation index data, and the dashed line is the rate of change in curvature from the VI data. The circles indicate transition dates. The extreme values located between each circle indicate the point at which the rate of change in curvature changes sign.

Fig. 23: A double logistic

- $p_5 = 220$

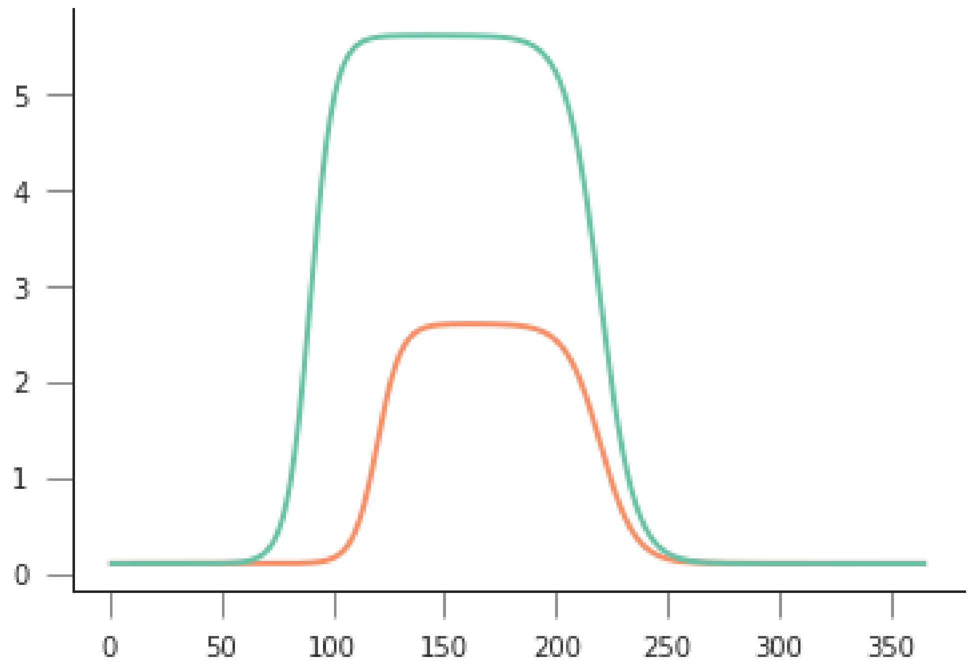
Write a function that produces the double logistic when passed an array of time steps (e.g. 1 to 365), and an array with six parameters.

Do some plots and try to get some intuition on the model parameters!

```
def dbl_sigmoid_function(p, t):

    sigma1 = 1./(1+np.exp(p[2]*(t-p[3])))
    sigma2 = 1./(1+np.exp(-p[4]*(t-p[5])))
    y = p[0] - p[1]*(sigma1 + sigma2 - 1)
    return y

p = np.array([0.1, 2.5, 0.19, 120, 0.13, 220])
t = np.arange(1, 366)
y = dbl_sigmoid_function(p, t)
plt.plot(t, y)


p = np.array([0.1, 5.5, 0.19, 90, 0.13, 220])
t = np.arange(1, 366)
y = dbl_sigmoid_function(p, t)
plt.plot(t, y)
```

```
[<matplotlib.lines.Line2D at 0x7f49fc222e80>]
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



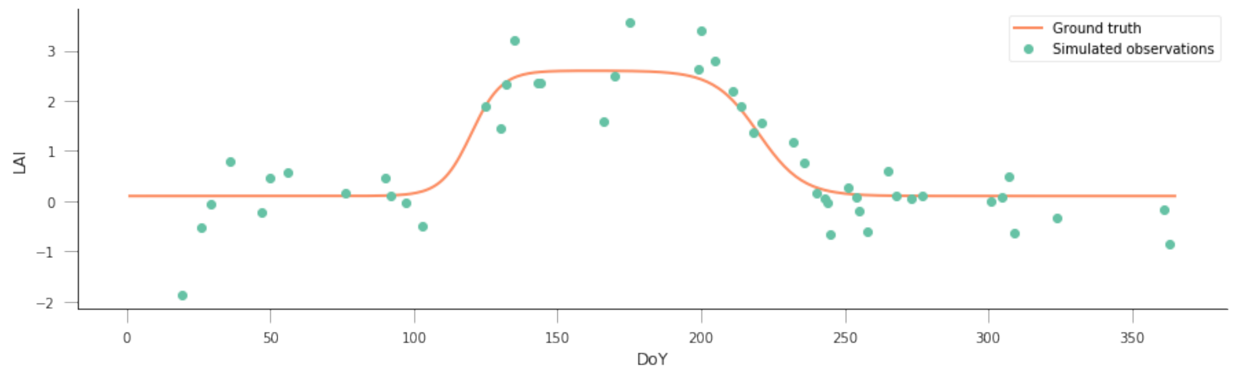
A synthetic experiment

A first step is to do a synthetic experiment. This has the marked advantage of being a situation where we're in control of everything.

```
def dbl_sigmoid_function(p, t):

    sigma1 = 1./(1+np.exp(p[2]*(t-p[3])))
    sigma2 = 1./(1+np.exp(-p[4]*(t-p[5])))
    y = p[0] - p[1]*(sigma1 + sigma2 - 1)
    return y

t = np.arange(1, 366)
p = np.array([0.1, 2.5, 0.19, 120, 0.13, 220])
y = dbl_sigmoid_function(p, t)
yn = y + np.random.randn(len(t))*0.6

selector = np.random.rand(365)

passer = np.where(selector > 0.9, True, False)

tn = t[passer]
yn = yn[passer]

fig = plt.figure(figsize=(15, 4))
_ = plt.plot(t, y, '-', label="Ground truth")
_ = plt.plot(tn, yn, 'o', label="Simulated observations")
plt.legend(loc="best")
plt.xlabel("DoY")
plt.ylabel("LAI")
```

```
Text(0,0.5,'LAI')
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



We know that the "true parameters" are given by `p = np.array([0.1, 2.5, 0.19, 120, 0.13, 220])`, but we see that the data is quite noisy and has significant gaps. As per last session, we could try to modify the parameters "by hand", and see how far we get, but given that it's 6, with different ranges, it looks a bit daunting. Also, we'd need to assess how good the solution is for a particular set of parameters, in other words, select a metric to quantify the goodness of fit.

It is useful to consider a model of the incomplete, noisy observations of LAI (y_n) and the true value of LAI, y. For overlapping time steps, the noisy data are just the "true" data plus some random Gaussian value with zero mean and a given variance σ_{obs}^2 (in the experiment above, $\sigma_{obs} = 0.6$):

$$y_n^i = y^i + \mathcal{N}(0, \sigma_{obs}^2).$$

Rearranging things, we have that $y_n - y$ should be a zero mean Gaussian distribution with known variance. We have assumed that our model is $f(\vec{p}) = y$, so we can write the *likelihood function*, $l(\vec{p})$

$$l(\vec{p}) = \left[\frac{1}{\sqrt{2\pi\sigma_{obs}^2}}\right]^N \prod_{i=1}^{N} \exp\left[-\frac{(y_n^i - f(\vec{p})^i)^2}{2\sigma_{obs}^2}\right].$$

It is convenient to take a logarithm of $l(\vec{p})$, so that we have the **log-likelihood**:

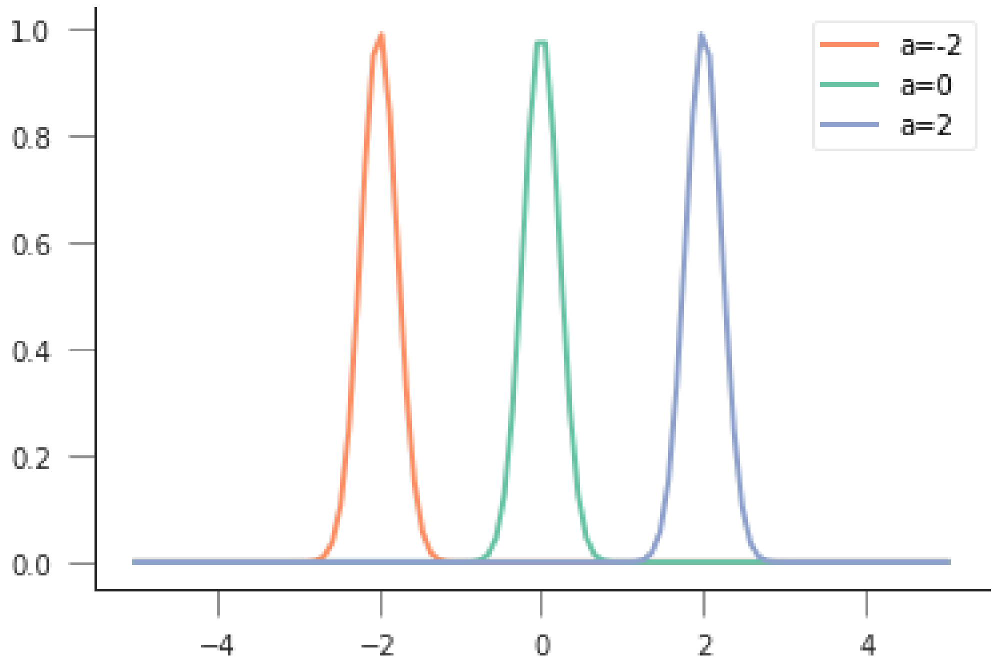
$$L(\vec{p}) = -\sum_{i=1}^{N}\left[\frac{(y_n^i - f(\vec{p})^i)^2}{2\sigma_{obs}^2}\right] + \text{Const.}$$

Think about the likelihood and log-likelihood… Think (and possibly plot) how a negative exponential curve looks like, and what conditions are for some interesting points.

```
x = np.linspace(-5, 5, 100)
for a in [-2, 0, 2]:
    plt.plot(x, np.exp(-(x-a)**2/0.1), '-', label=f"a={a:g}")
plt.legend(loc="best")
```

```
<matplotlib.legend.Legend at 0x7f49fbee64e0>
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



So for a sum of squares, the most likely result would be if all the mismatches were zero, which means that the log-likelihood is 0, and the likelihood, $exp(0) = 1$!

However, the mismatch might not be 0, due to the added noise. So what we're effectively looking for is a **maximum** in the log-likelihood, or a **minimum** of its negative as a function of $\vec{p}$:

$$\frac{\partial(-L(\vec{p}))}{\partial\vec{p}} \triangleq \min$$

So, we can try our brute force guessing approach by **minimising the cost function given by :math:'L(vec{p})'**

Write the cost function! Test it possibly shifting one parameter over some range of values

```
def cost_function(p, t, y_obs, passer, sigma_obs, func=dbl_sigmoid_function):
    y_pred = func(p, t)
    cost = -0.5* (y_pred[passer]-y_obs)**2/sigma_obs**2
    return -cost.sum()

# Let's switch p3
p = np.array([0.1, 2.5, 0.19, 120, 0.13, 220])
for ps in [60, 90, 120, 150, 180]:
    pa = p*1
    pa[3] = ps
    print(f"p_3: {ps:g}=> " +
          f"Cost: {cost_function(pa, t, yn, passer, 0.6):g}")
```

```
p_3: 60=> Cost: 66.0294
p_3: 90=> Cost: 42.9216
p_3: 120=> Cost: 20.4129
p_3: 150=> Cost: 51.5062
p_3: 180=> Cost: 76.2889
```

The easiest way to obtain the solution is to use numerical optimisation techniques to minimise the cost function. In scipy, there's a good selection of function optimisers. We'll be looking at **local** optimisers: these will look for a minimum in the vicinity of a user-given starting point, usually by looking at the gradient of the cost function. The main function to consider here is `minimise <https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html#scipy.optimize.minimize>'__. Basically, `minimize` takes a cost function, a starting point, and maybe extra arguments that are passed to the cost function, and uses one of several algorithms to minimise the cost function. We import it with

```
from scipy.optimize import minimize
```

From the documentation,

```
minimize(fun, x0, args=(), method=None,
         jac=None, hess=None, hessp=None,
         bounds=None, constraints=(), tol=None,
         callback=None, options=None)
```

Basically, `fun` is the name of the cost function. The first parameter you pass to the cost function has to be an array with the parameters that will be used to calculate the cost. `x0` is the starting point. `args` allows you to add extra parameters that are required for the cost function (in our example, these would be `t, y_obs, passer, sigma_obs`).

The `minimize` function returns an object with the

1. Value of the function at the minimum,
2. The value of the input parameters that attain the minimum,
3. A message telling you whether the optimisation succeeded

4. The number of iterations (`nit`) and total function evaluations (`nfev`)
5. Some diagnostics

```
from scipy.optimize import minimize
```

```
from scipy.optimize import minimize

p0 = np.array([0, 5, 0.01, 90, 0.01, 200])

retval = minimize(cost_function, p0, args=(t, yn, passer, 0.6))

print(retval)

print ("********************************************")
if retval.success:
    print("Optimisation successful!")
    print(f"Value of the function at the minimum: {retval.fun:g}")
    print(f"Value of the solution: {str(retval.x):s}")
```

```
      fun: 18.294311148740324
 hess_inv: array([[ 1.36696628e-02, -1.45579536e-02,  2.18040346e-03,
         1.04214972e-01,  1.29518890e-03, -1.61553074e-01],
       [-1.45579536e-02,  8.11850410e-02, -1.67483584e-02,
         1.49813443e-01, -4.37870879e-03, -4.38517048e-01],
       [ 2.18040346e-03, -1.67483584e-02,  2.16380108e-02,
         3.71000276e-01,  6.94701180e-04,  1.26794966e-01],
       [ 1.04214972e-01,  1.49813443e-01,  3.71000276e-01,
         2.26129353e+01, -6.50788440e-03, -2.82996618e+00],
       [ 1.29518890e-03, -4.37870879e-03,  6.94701180e-04,
        -6.50788440e-03,  1.46725192e-03, -5.11647642e-03],
       [-1.61553074e-01, -4.38517048e-01,  1.26794966e-01,
        -2.82996618e+00, -5.11647642e-03,  1.85405981e+01]])
      jac: array([-4.05311584e-06, -1.66893005e-06,  9.53674316e-07,  2.
↪38418579e-07,
       -8.82148743e-06,  4.76837158e-07])
  message: 'Optimization terminated successfully.'
     nfev: 448
      nit: 50
     njev: 56
   status: 0
  success: True
        x: array([-9.01539704e-02,  2.77958809e+00,  1.95754564e-01,  1.
↪22633761e+02,
        1.28439911e-01,  2.24233838e+02])
****************************************
Optimisation successful!
Value of the function at the minimum: 18.2943
Value of the solution: [-9.01539704e-02  2.77958809e+00  1.95754564e-01  1.
↪22633761e+02
  1.28439911e-01  2.24233838e+02]
```

Do some synthetic experiments. For example:

Change the true parameters and see how the solution tracks the change.

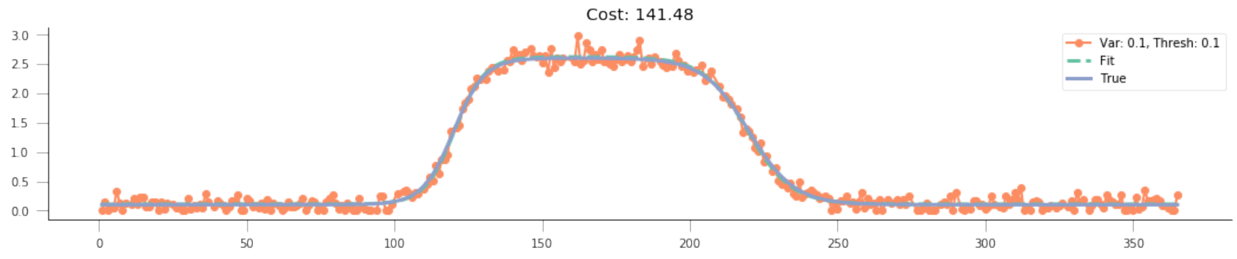
Increase the added variance

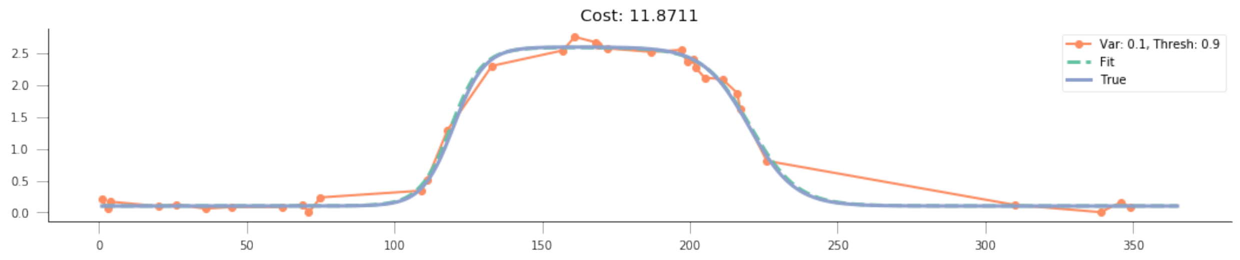
Reduce or increase the number of observations

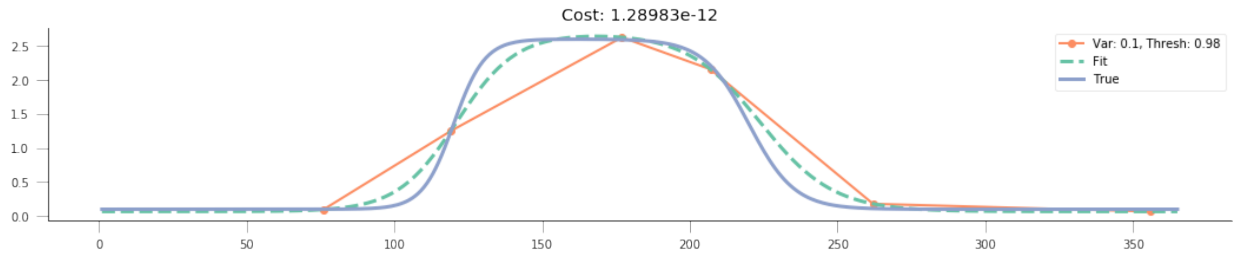
Use these experiments to challenge your understanding of the problem: Try to think what the expected result of these changes is, and write a set of functions that simplify the exploration.

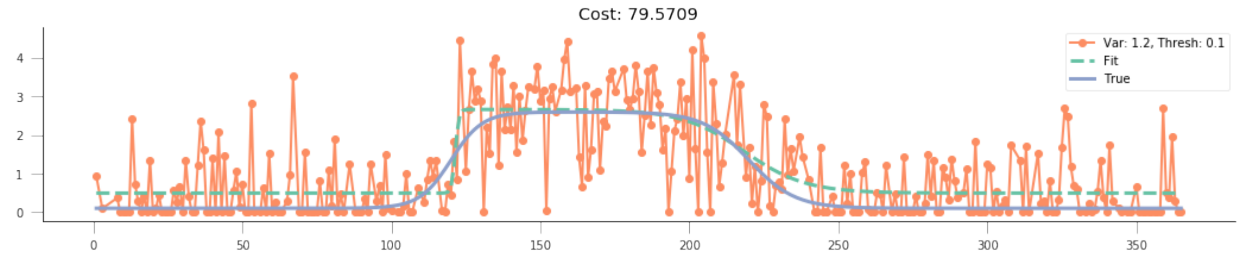
```
p = np.array([0.1, 2.5, 0.19, 120, 0.13, 220])

def create_observations(p, variance=0.3,
                        threshold=0.9):
    t = np.arange(1, 366)
    y = dbl_sigmoid_function(p, t)
    yn = y + np.random.randn(len(t))*variance

    yn = np.clip(yn, 0, None)

    selector = np.random.rand(365)

    passer = np.where(selector > threshold, True, False)

    tn = t[passer]
    yn = yn[passer]
    return tn, yn, passer


for variance in [0.1, 1.2]:

    for thresh in [0.1, 0.9, 0.98]:
        plt.figure(figsize=(18,3))

        tn, yn, passer = create_observations(p, variance=variance,
                                             threshold=thresh)
        label = f"Var: {variance:g}, Thresh: {thresh:g}"
        plt.plot(tn, yn, "-o", label=label)
        p0 = np.array([0, 5, 0.01, 90, 0.01, 200])
        retval = minimize(cost_function, p0, args=(t, yn, passer, variance))
        plt.plot(t, dbl_sigmoid_function(retval.x, t), '--', lw=3, label="Fit")
        plt.plot(t, dbl_sigmoid_function(p, t), '-', lw=3, label="True")

        plt.title(f"Cost: {retval.fun:g}")
        plt.legend(loc="best")
```

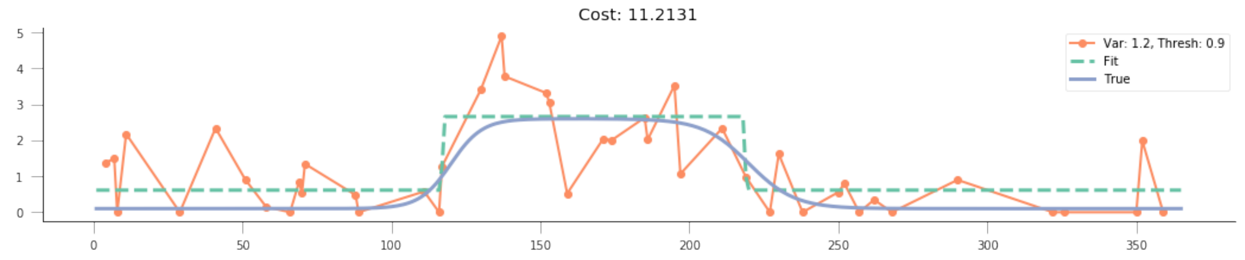
```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```

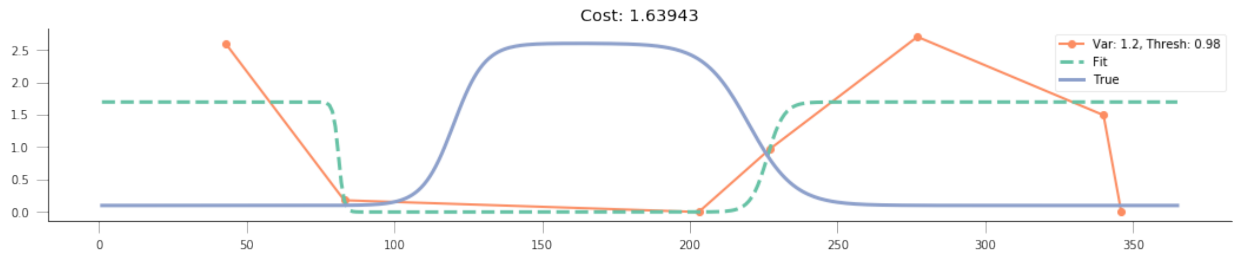








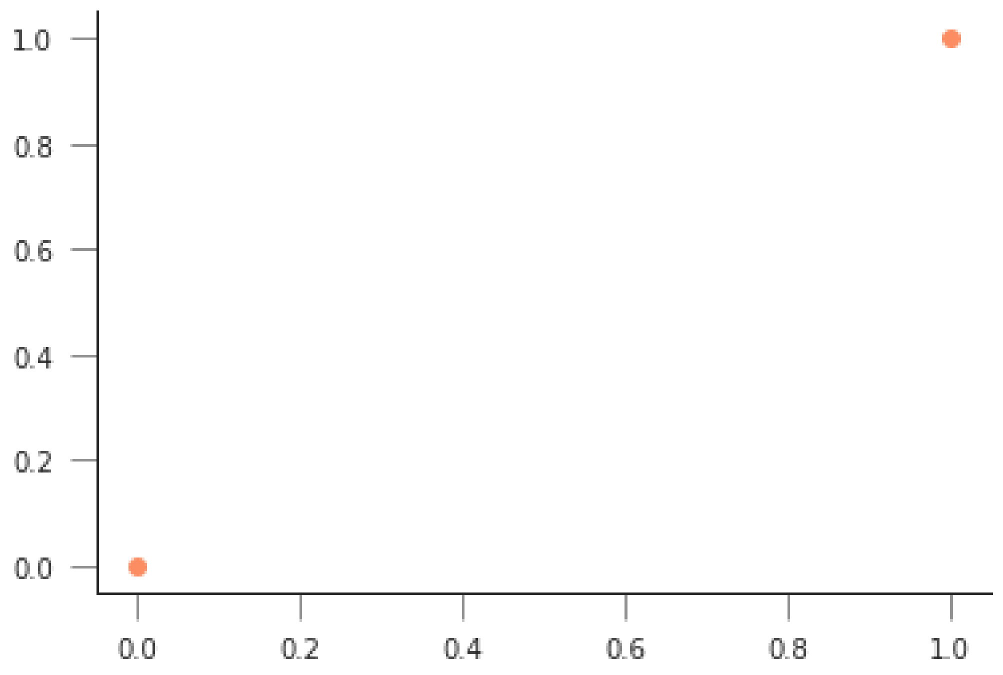




```
plt.plot([0,1], [0,1], 'o')
```

```
[<matplotlib.lines.Line2D at 0x7f49f626cb00>]
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



Fitting MODIS LAI data

We're now ready for trying out MODIS LAI data. We will use a lot of the machinery we encountered earlier. A mosaic of MODIS LAI and QA over Europe has been prepared for you (you should be able to do this yourselves by now though!). We also rescuing the functions

```
def get_sfc_qc(qa_data, mask57 = 0b11100000):
    sfc_qa = np.right_shift(np.bitwise_and(qa_data, mask57), 5)
    return sfc_qa

def get_scaling(sfc_qa, golden_ratio=0.61803398875):
    weight = np.zeros_like(sfc_qa, dtype=np.float)
    for qa_val in [0, 1, 2, 3]:
        weight[sfc_qa == qa_val] = np.power(golden_ratio, float(qa_val))
    return weight

def get_point_from_LL(raster, lat, long):
    g = gdal.Open(raster)
    geot = g.GetGeoTransform()
    wgs84 = osr.SpatialReference()
    wgs84.ImportFromEPSG(4326)
    modis_sinu = osr.SpatialReference()
    modis_sinu.ImportFromProj4("+proj=sinu +lon_0=0 +x_0=0 +y_0=0 " +
                               "+a=6371007.181 +b=6371007.181 +units=m +no_defs")
    tx = osr.CoordinateTransformation(wgs84, modis_sinu)
    point_x, point_y, point_z = tx.TransformPoint(long, lat)
    inv_geoT = gdal.InvGeoTransform(geot)
    r, c = (gdal.ApplyGeoTransform(inv_geoT, point_x, point_y))
    return int(r + 0.5), int(c + 0.5)
```

(continues on next page)

(continued from previous page)

```
def read_tseries(raster, lat, long):
    g = gdal.Open(raster)
    px, py = get_point_from_LL(raster, lat, long)
    xbuf = 1
    ybuf = 1
    n_doys = g.RasterCount
    buf = g.ReadRaster (px, py,
                xbuf, ybuf, buf_xsize=xbuf, buf_ysize=ybuf, \
                band_list=np.arange (1, n_doys+1))
    lai = np.frombuffer ( buf, dtype=np.uint8)
    return lai
```

```
# All imports go here. Run me first!
import datetime
from pathlib import Path  # Checks for files and so on
import numpy as np  # Numpy for arrays and so on
import pandas as pd
import sys
import matplotlib.pyplot as plt  # Matplotlib for plotting
# Ensure the plots are shown in the notebook
%matplotlib inline

import gdal
import osr
import numpy as np

from geog0111.geog_data import procure_dataset
%matplotlib inline
```

2.8.25 Fitting models of phenology to MODIS LAI data

In the previous Section, we have looked at a very simple model of phenology, probably applicable to a lot of deciduous vegetation. We have created some synthetic data, and we have tried to fit the model to these "pseudo observations". It is now the time to extract time series of MODIS data, and try to fit the model to them.

In this Section, we'll use some of the work we used before. In particular, we'll use a mosaic of LAI over Western Europe derived from the MODIS MCD15 product. The product has already been packed in an easy to use GeoTIFF file, together with the corresponding QA dataset. We'll define some functions here again (they are from other sections, but added here for simplicity):

The phenology model

The phenology model is the double logistic/sigmoid curve, given by

$$y = p_0 - p_1 \cdot \left[\frac{1}{1+\exp\left(p_2\cdot(t-p_3)\right)} + \frac{1}{1+\exp\left(-p_4\cdot(t-p_5)\right)} - 1\right].$$

We also define a cost function for it based on observations of LAI, uncertainty and a mask to indicate missing observations.

```
def dbl_sigmoid_function(p, t):
    """The double sigmoid function defined over t (where t is an array).
    Takes a vector of 6 parameters"""
```

(continues on next page)

(continued from previous page)

```
    sigma1 = 1./(1+np.exp(p[2]*(t-p[3])))
    sigma2 = 1./(1+np.exp(-p[4]*(t-p[5])))
    y = p[0] - p[1]*(sigma1 + sigma2 - 1)
    return y

def cost_function(p, t, y_obs, passer, sigma_obs, func=dbl_sigmoid_function):
    y_pred = func(p, t)
    cost = -0.5* (y_pred[passer]-y_obs)**2/sigma_obs**2
    return -cost.sum()
```

Interpreting the QA LAI data

From before, we interpret the QA layer in the MODIS product by looking at bits 5 to 7, and turning them into a weight given by ϕ^{QA}, where ϕ is the golden ratio $(= 0.618\ldots)$, and QA can take values between 0 and 3, indicating a decreasing quality in the retrieval, and hence, a lower weight.

```
def get_sfc_qc(qa_data, mask57 = 0b11100000):
    sfc_qa = np.right_shift(np.bitwise_and(qa_data, mask57), 5)
    return sfc_qa

def get_scaling(sfc_qa, golden_ratio=0.61803398875):
    weight = np.zeros_like(sfc_qa, dtype=np.float)
    for qa_val in [0, 1, 2, 3]:
        weight[sfc_qa == qa_val] = np.power(golden_ratio, float(qa_val))
    return weight
```

Selecting data from a raster file

In this activity, we want to select a pixel in the map and read in the data for all time steps. This can be achieved by plotting the map with e.g. `imshow`, and judiciously selecting a row and a column. However, in many cases, we have locations of interest as a list of latitude and longitude pairs. These do not automatically map to rows and columns, because the the MODIS data are **projected** in a projection that is not latitude/longitude. So the process of going from latitude-longitude pair to row and column needs two steps: converting latitude-longitude coordinates into the coordinates of the raster of interest (in the case of MODIS, MODIS sinusoidal projection), and then converting the raster coordinates into pixel numbers.

Converting geographic projections

Converting from latitude-longitude pairs (or any other representation) into a different projection can be accomplished by GDAL (surprise!). We first need to find a way to defining the projection. There are several ways to do this:

- **EPSG codes** These are numerical codes that have been internationally agreed and fully define a projection
- **Proj4 strings** Proj4 is the library the manages coordinate conversions under the hood in GDAL. It has a method to define a projection as a text string.
- **WKT (Well-known text) format** This is a standard that defines the projection as a text block

Generally speaking, their simplicity of use recommends EPSG, a single number. In some cases, proj4 strings are best (e.g. for some product-specific projections), and WKT is generally used by other GIS software.

In any case, the spatialreference website provides a convenient "Rosetta stone" of projections in these different conventions.

Use spatialreference.org to find out what projection the EPSG code 4326 corresponds to

In Python, using the OSR part of the GDAL library, we define the source and destinations projections using `SpatialReference` objects, which are then populated with e.g. EPSG codes or proj4 strings:

```
import osr
# Define the Lat/Long object
wgs84 = osr.SpatialReference()
# In this case, we use EPSG code
wgs84.ImportFromEPSG(4326)
# Define the MODIS projection object
modis_sinu = osr.SpatialReference()
# In this case, we use the proj4 string
modis_sinu.ImportFromProj4("+proj=sinu +lon_0=0 +x_0=0 +y_0=0 " +
                           "+a=6371007.181 +b=6371007.181 +units=m +no_defs")
```

The previous code snippet defines two `SpatialReference` objects. These can be used to map from MODIS to/and from Latitude Longitude (or "WGS84") coordinates by using the `osr.CoordinateTransformation` object:

```
transformation = osr.CoordinateTransformation(wgs84, modis_sinu)
modis_x, modis_y, modis_z = transformation.TransformPoint(longitude,
                                                          latitude)
```

Clearly, changing the order of the parameters in `osr.CoordinateTransformation` would reverse the transformation.

Write some python code to convert the location of the Pearson Building (latitude: 51.524750 decimal degrees, longitude=-0.134560 decimal degrees) between WGS84 and OSGB 1936/British National Grid and UTM zone 30N/WGS84. Use mygeodata.cloud to test that your results are sensible

Finding a pixel based on its coordinates

Geospatial data usually contain a definition of how to go from a coordinate to a pixel location. In GDAL, the generic way this is encoded is through the `GeoTransform` element, a six element vector that details the location of the **U**pper **L**eft corner of the raster file (pixel position (0, 0)), the pixel spacing, as well as a possible angular shift. Here are the elements of the geotransform array:

1. The Upper Left easting coordinate (i.e., horizontal)
2. The E-W pixel spacing
3. The rotation (0 degrees if image is "North Up")
4. The Upper left northing coordinate (i.e., vertical)
5. The rotation (0 degrees)
6. The N-S pixel spacing, negative as we will be counting from the UL corner

With this in mind, and remembering that in Python arrays start at 0, and ignoring the rotation contributions, the pixel numbers can be calculated as follows

```
 pixel_x = (x_location - geo_transform[0])/geo_transform[1] \
     # The difference in distance between the UL corner (geot[0] \
     #and point of interest. Scaled by geot[1] to get pixel number

pixel_y = (y_location - geo_transform[3])/(geo_transform[5]) # Like for pixel_x, \
     #but in vertical direction. Note the different elements of geot \
     #being used
```

Since it's easy to get this wrong, GDAL provides a couple of methods to do this conversion directly:

```
inv_geoT = gdal.InvGeoTransform(geotransform)
r, c = (gdal.ApplyGeoTransform(inv_geoT, x_location, y_location))
```

Let's see a whole example of this zooming in the fAPAR map from the MODIS MCD15 product near the fine city of A Coruña in Galicia, NW Spain (latitude: 43.3623, longitude: -8.4115):

```
%%html
<iframe src="https://www.google.com/maps/embed?pb=!1m14!1m12!1m3!1d238659.69294928786!
↪2d-8.664931741126212!3d43.39317238062582!2m3!1f0!2f0!3f0!3m2!1i1024!2i768!4f13.1!
↪5e1!3m2!1sen!2suk!4v1542817273641" width="600" height="450" frameborder="0" style=
↪"border:0" allowfullscreen></iframe>
```

```
###################################################################
# Define transformations and variables. This is like above!
###################################################################

y_location, x_location = 43.3623, -8.4115 # In degs
# Define the Lat/Long object
wgs84 = osr.SpatialReference()
# In this case, we use EPSG code
wgs84.ImportFromEPSG(4326)
# Define the MODIS projection object
modis_sinu = osr.SpatialReference()
# In this case, we use the proj4 string
modis_sinu.ImportFromProj4("+proj=sinu +lon_0=0 +x_0=0 +y_0=0 " +
                           "+a=6371007.181 +b=6371007.181 +units=m +no_defs")

transformation = osr.CoordinateTransformation(wgs84, modis_sinu)
modis_x, modis_y, modis_z = transformation.TransformPoint(x_location,
                                                          y_location)
print("MODIS coordinates: ", modis_x, modis_y)

###################################################################
# We use a random file in the UCL filesystem
###################################################################

fname = "/home/plewis/public_html/geog0111_data/lai_files/" + \
        "MCD15A3H.A2016273.h17v04.006.2016278070708.hdf"
g = gdal.Open('HDF4_EOS:EOS_GRID:"%s":MOD_Grid_MCD15A3H:Fpar_500m' % fname)


###################################################################
# This is where new stuff begins
# Find out the pixel location from the MODIS Easting & Northing
###################################################################

geoT = g.GetGeoTransform()
inv_geoT = gdal.InvGeoTransform(geoT)
r, c = (gdal.ApplyGeoTransform(inv_geoT, modis_x, modis_y))
r = int(r+0.5)
c = int(c+0.5)
print("Pixel location: ", r,c)

###################################################################
# Now, read in the data, and plot it
###################################################################
```

(continues on next page)

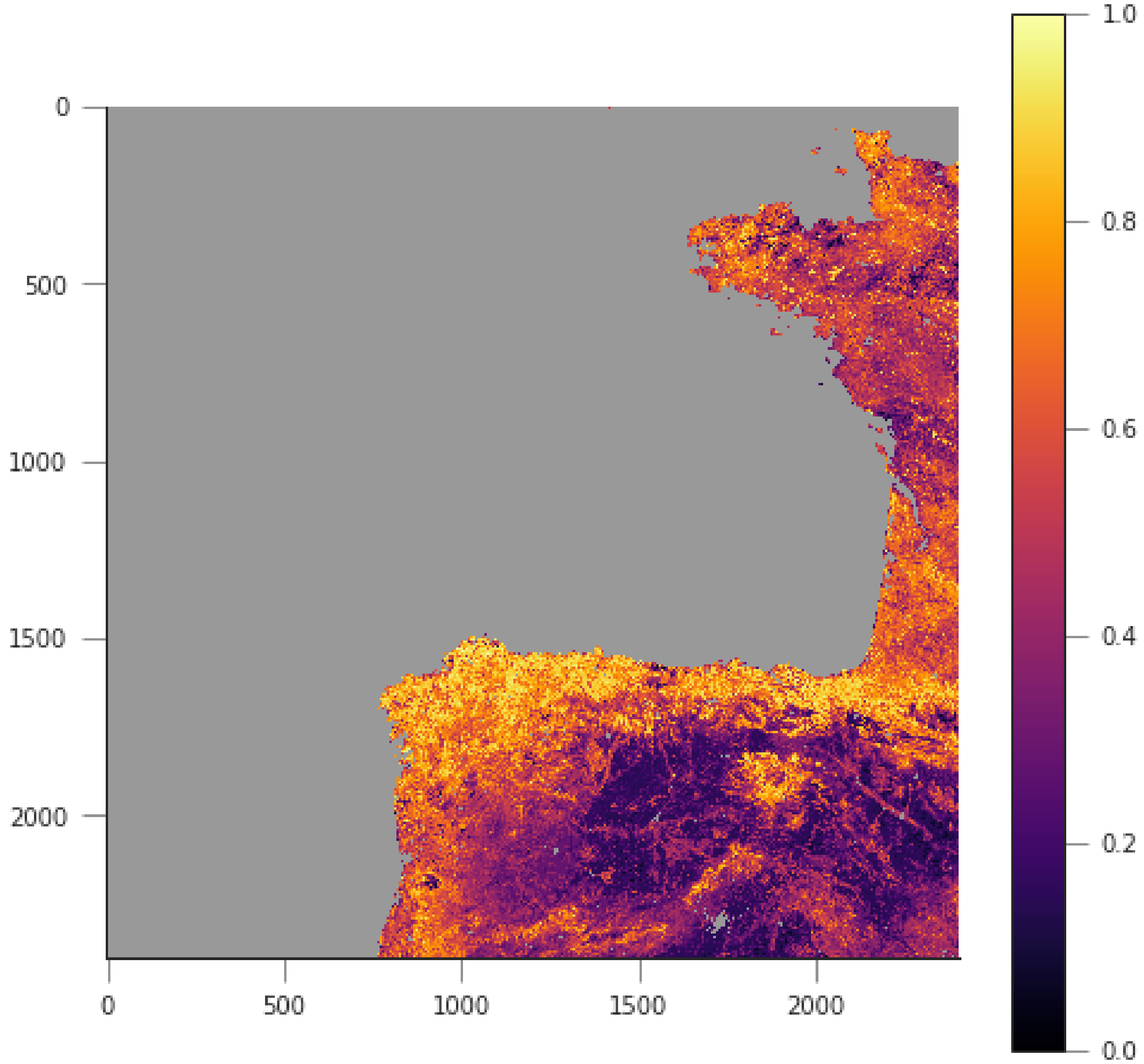
(continued from previous page)

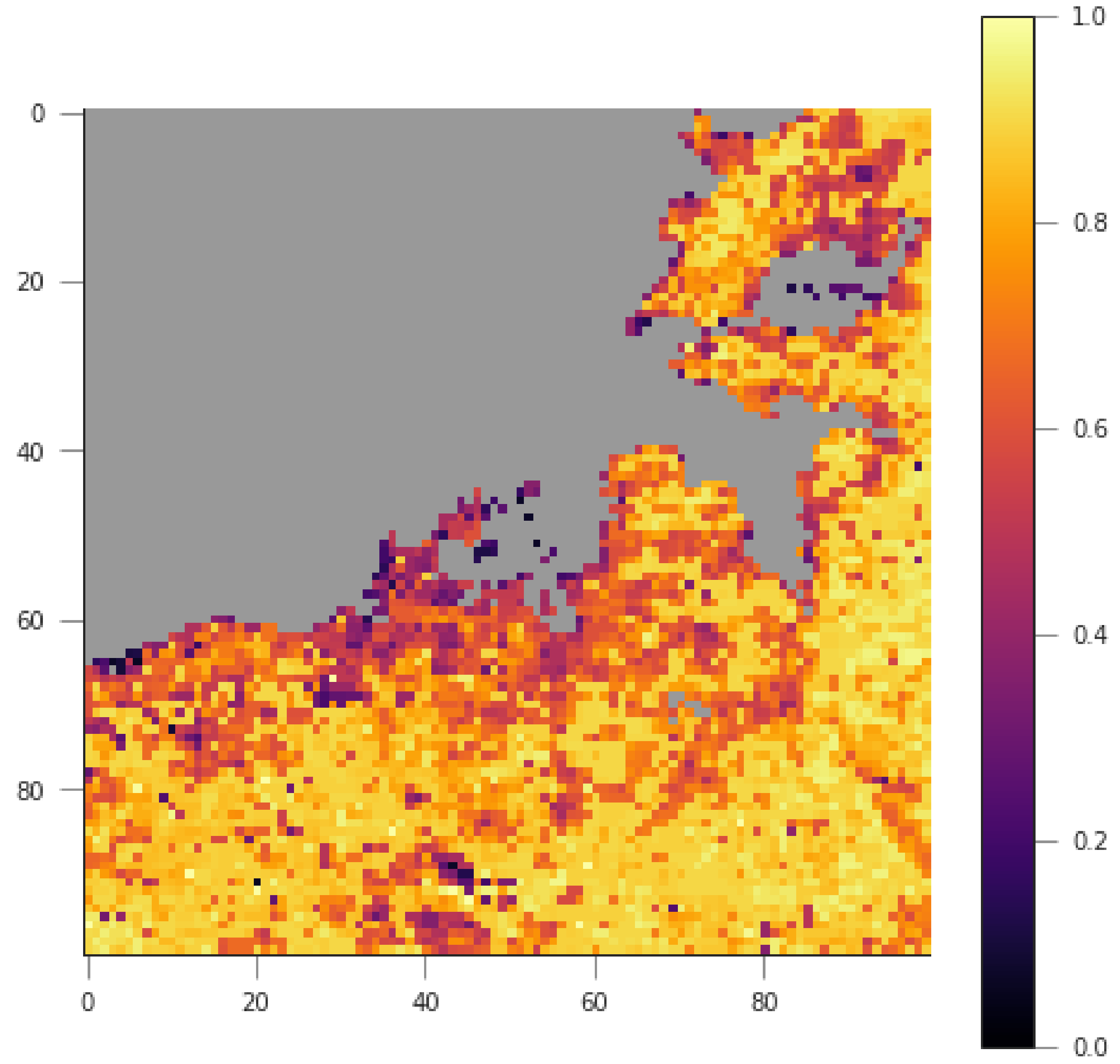
```
fapar = g.ReadAsArray()/100
fapar[fapar>1] = np.nan
cmap = plt.cm.inferno
cmap.set_bad("0.6")
plt.figure(figsize=(8, 8))
plt.imshow(fapar, interpolation="nearest", vmin=0, vmax=1, cmap=cmap)
plt.colorbar()

#################################################################
# Plot a zoomed-in version
#################################################################


plt.figure(figsize=(8, 8))
plt.imshow(fapar[(c-50):(c+50), (r-50):(r+50)], interpolation="nearest",
           vmin=0, vmax=1, cmap=cmap)
plt.colorbar()
```

```
MODIS coordinates:  -680000.4782137175 4821673.202327191
Pixel location:  932 1593
```

```
<matplotlib.colorbar.Colorbar at 0x7f0645b9fa20>
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```


0
500
1000
1500
2000
0
500
1000
1500
2000
1.0
0.8
0.6
0.4
0.2
0.0



We can see that in the example above, we're getting the right pixel number. Clearly, the code above is a bit of a mess, and needs to be cleaned up, split into functions and tested. This is an example, and you can take this as a reference of how to document functions etc.

```
def convert_coordinates(x_location, y_location,
                        src_transform={'EPSG':4326},
                        dst_transform={'Proj4':
                                           "+proj=sinu +lon_0=0 +x_0=0 " +
                                           "+y_0=0 +a=6371007.181 " +
                                           "+b=6371007.181 +units=m +no_defs"
                                          }):
    """A function to convert coordinates from one target coordinate
    representation to another. The input an output transformation can be given
    in either EPSG codes or Proj4 strings, by providing the function with a
    dictionary with the desired convention as a key, and with the relevant
```

(continues on next page)

(continued from previous page)

```
    codes as its only element.

    Parameters
    ----------
    x_location: float
        The x location
    y_location: float
        The y location
    src_transform: dict
        A dictionary with keys either "EPSG" or "Proj4" (anything else throws
        an exception) with the description of the **input** projection
    dst_transform: dict
        A dictionary with keys either "EPSG" or "Proj4" (anything else throws
        an exception) with the description of the **output** projection
    Returns
    --------

    The transformed x and y coordinates"""
    input_coords = osr.SpatialReference()
    # In this case, we use EPSG code
    try:
        input_coords.ImportFromEPSG(src_transform["EPSG"])
    except KeyError:
        input_coords.ImportFromProj4(src_transform["Proj4"])
    except KeyError:
        raise ValueError("src_transform not dictionary with EPSG/Proj4 keys!")


    output_coords = osr.SpatialReference()
    try:
        output_coords.ImportFromEPSG(dst_transform["EPSG"])
    except KeyError:
        output_coords.ImportFromProj4(dst_transform["Proj4"])
    except KeyError:
        raise ValueError("src_transform not dictionary with EPSG/Proj4 keys!")


    transformation = osr.CoordinateTransformation(input_coords,
                                                  output_coords)
    output_x, output_y, output_z = transformation.TransformPoint(x_location,
                                                                  y_location)
    return output_x, output_y


##################################################################
# Test function
##################################################################

y_location, x_location = 43.3623, -8.4115 # In WGS84
print (convert_coordinates(x_location, y_location))
```

```
(-680000.4782137175, 4821673.202327191)
```

```
def get_pixel(raster, point_x, point_y):
    """Get the pixel for given coordinates (in the raster's convention, not
    checked!) for a raster file.
```

(continues on next page)

(continued from previous page)

```
    Parameters
    ----------
    raster: string
        A GDAL-friendly raster filename
    point_x: float
        The Easting in the same coordinates as the raster (not checked!)
    point_y: float
        The Northing in the same coordinates as the raster (not checked!)

    Returns
    -------
    The row/column (or column/row, depending on how you define it)
    """
    g = gdal.Open(raster)
    if g is None:
        raise ValueError(f"{raster:s} cannot be opened!")
    geoT = g.GetGeoTransform()
    inv_geoT = gdal.InvGeoTransform(geoT)
    r, c = (gdal.ApplyGeoTransform(inv_geoT, point_x, point_y))
    return int(r + 0.5), int(c + 0.5)

###################################################################
# Test function
###################################################################


fname = "/home/plewis/public_html/geog0111_data/lai_files/" + \
            "MCD15A3H.A2016273.h17v04.006.2016278070708.hdf"
gdal_fname = 'HDF4_EOS:EOS_GRID:"%s":MOD_Grid_MCD15A3H:Fpar_500m' % fname
print (get_pixel(gdal_fname, -680000.4782137175, 4821673.202327191))
```

```
(932, 1593)
```

Retrieving a time series from a multi-band raster

We have produced 4 rasters, with the LAI value for 2016 and 2017, as well as the correspodingn `FparLai_QC` layer. They're avaialable in `data/euro_lai`. Let's quickly have a look at the data:

```
success = procure_dataset("euro_lai", destination_folder="data/euro_lai/",
                         verbose=True)
if not success:
    print("Something happened copying files across to data/euro_lai")

print(gdal.Info("data/euro_lai/Europe_mosaic_Lai_500m_2017.tif").split("\n")[:10])
```

```
Running on UCL's Geography computers
trying /archive/rsu_raid_0/plewis/public_html/geog0111_data
trying /data/selene/ucfajlg/geog0111_data/lai_data/
trying /data/selene/ucfajlg/geog0111_data/
Linking /data/selene/ucfajlg/geog0111_data/euro_lai/Europe_mosaic_FparLai_QC_2016.tif␣
↪to data/euro_lai/Europe_mosaic_FparLai_QC_2016.tif
Linking /data/selene/ucfajlg/geog0111_data/euro_lai/Europe_mosaic_FparLai_QC_2017.tif␣
↪to data/euro_lai/Europe_mosaic_FparLai_QC_2017.tif
```

(continues on next page)

(continued from previous page)

```
Linking /data/selene/ucfajlg/geog0111_data/euro_lai/Europe_mosaic_Lai_500m_2016.tif␣
↪to data/euro_lai/Europe_mosaic_Lai_500m_2016.tif
Linking /data/selene/ucfajlg/geog0111_data/euro_lai/Europe_mosaic_Lai_500m_2017.tif␣
↪to data/euro_lai/Europe_mosaic_Lai_500m_2017.tif
['Driver: GTiff/GeoTIFF', 'Files: data/euro_lai/Europe_mosaic_Lai_500m_2017.tif',
↪'Size is 4800, 4800', 'Coordinate System is:', 'PROJCS["unnamed",', '    GEOGCS[
↪"Unknown datum based upon the custom spheroid",', '        DATUM["Not_specified_
↪based_on_custom_spheroid",', '            SPHEROID["Custom spheroid",6371007.181,
↪0]],', '        PRIMEM["Greenwich",0],', '        UNIT["degree",0.
↪0174532925199433]],']
```

We have 90 (or 91) layers, from day 1 to day 360/364 in the year. While we could read all the data in memory, it's wasteful of resources, and we might as well try to read in all the bands for a given pixel.

We can do this with the `read_tseries` function below. Basically, we this function calls the previous pixel-location functions, and then reads the entire time series for a pixel in one go. The function is defined below:

```
def read_tseries(raster, lat, long):
    """Read a time series (or all bands) for a raster file given latitude and
    longitude coordinates.
    **NOTE** Only works with Byte/UInt8 data types!
    """
    g = gdal.Open(raster)
    px, py = get_pixel(raster, *convert_coordinates(long, lat))
    if 0 <= px >= g.RasterXSize:
        raise ValueError(f"Point outside of raster ({px:d}/{g.RasterXSize:d})")
    if 0 <= py >= g.RasterYSize:
        raise ValueError(f"Point outside of raster ({py:d}/{g.RasterYSize:d})")

    xbuf = 1
    ybuf = 1
    n_doys = g.RasterCount
    buf = g.ReadRaster (px, py,
                xbuf, ybuf, buf_xsize=xbuf, buf_ysize=ybuf,
                band_list=np.arange (1, n_doys+1))
    data = np.frombuffer ( buf, dtype=np.uint8)
    return data
```

The Nature reserve of Muniellos (43.0156, -6.7038) is mostly populated by *Quercus Robur*, which shows a strong phenology. It should be a good test point to see whether the data we have is sensible or not.

Using the provided function, plot the time series of LAI over the Muniellos Reserve for 2016 and 2017. Extra points for using QA flags to filter the data

```
%%html

<div>
     <iframe width="500" height="400" frameborder="0" src="https://www.bing.com/maps/
↪embed?h=400&w=500&cp=43.029897999999996~-6.734863999999996&lvl=11&typ=d&sty=h&
↪src=SHELL&FORM=MBEDV8" scrolling="no">
     </iframe>
     <div style="white-space: nowrap; text-align: center; width: 500px; padding: 6px␣
↪0;">
        <a id="largeMapLink" target="_blank" href="https://www.bing.com/maps?cp=43.
↪029897999999996~-6.734863999999996&amp;sty=h&amp;lvl=11&amp;FORM=MBEDLD">View␣
↪Larger Map</a> &nbsp; | &nbsp;
        <a id="dirMapLink" target="_blank" href="https://www.bing.com/maps/directions?
↪cp=43.029897999999996~-6.734863999999996&amp;sty=h&amp;lvl=11&amp;rtp=~pos.43.
↪029897999999996_-6.734863999999996____&amp;FORM=MBEDLD">Get Directions</a>
```

(continues on next page)

(continued from previous page)

```
    </div>
</div>
```

We can now try to fit our double logistic model to the observations, weighted by their uncertainty. We make use the previously defined functions for the model and the cost function that we defined above. We will start by fitting the data to 2016, but will also try to "eyeball" a good starting point for the optimisation. And obviously, we'll want some plots…

A nice way to plot points with errorbars is (surprisingly enought) the `` `plt.errorbar `` method <https://matplotlib.org/api/_as_gen/matplotlib.pyplot.errorbar.html>`__. It's like the `plt.plot` method, but it also takes a `yerr` (or `xerr`) keyword with the extent of the error in the y direction.

```
from scipy.optimize import minimize

plt.figure(figsize=(15, 6))
y_location, x_location = 43.0156, -6.7038
###################################################################
# Start by reading in the data
###################################################################
# Filenames
year = 2016
lai_raster = f"data/euro_lai/Europe_mosaic_Lai_500m_{year:d}.tif"
qa_raster = f"data/euro_lai/Europe_mosaic_FparLai_QC_{year:d}.tif"
# Actually read the data
data = read_tseries(lai_raster, y_location, x_location)/10. # Read LAI
qa = read_tseries(qa_raster, y_location, x_location) # Read QA/QC
# We only want to use QA flags 0 or 1
passer = get_sfc_qc(qa) <= 1
# This is the uncertainty
sigma = get_scaling(get_sfc_qc(qa))[passer]
# This is the time axis: every 4 days
t = np.arange(len(passer))*4 + 1

###################################################################
# Plot the observations of LAI with uncertainty bands
###################################################################

plt.errorbar(t[passer], data[passer], yerr=sigma, fmt="o",
             mfc="none", label=f"Obs {year:d}")

###################################################################
# Plot a first prediction with some random model parameters
###################################################################
# First eyeballing test:
p0 = np.array([0.5, 6, 0.2, 150, 0.23, 240])
plt.plot(t, dbl_sigmoid_function(p0, t), '--', label="1st test")
print("Cost: ",
      cost_function(p0, t, data[passer], passer, sigma))

###################################################################
# Plot a second, more refined prediction
###################################################################

# Second eyeballing test:
p0 = np.array([0.1, 5, 0.2, 140, 0.23, 260])
plt.plot(t, dbl_sigmoid_function(p0, t), '--', label="2nd test")
```

(continues on next page)

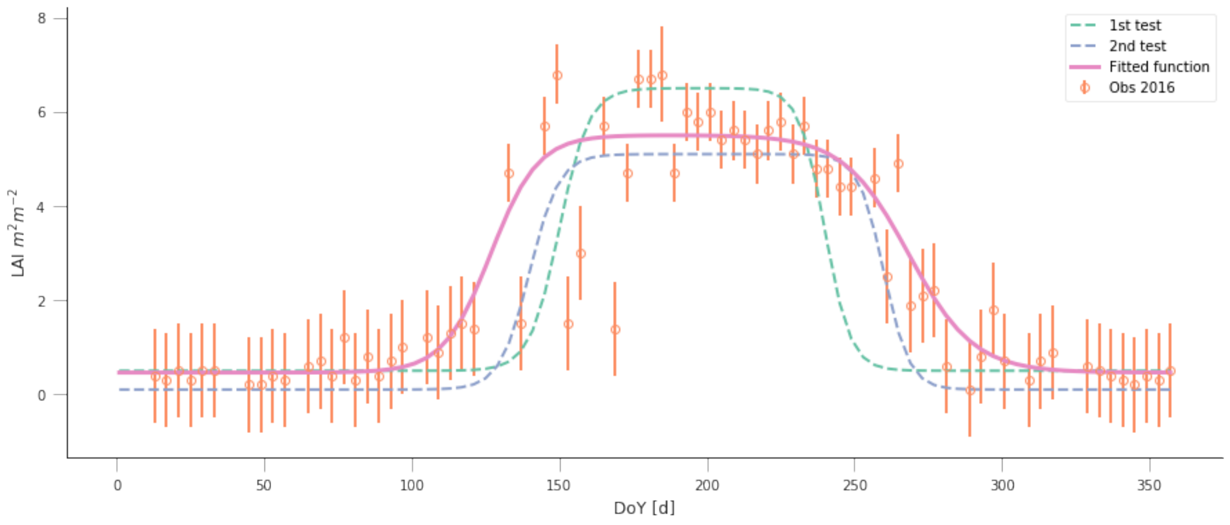
(continued from previous page)

```
print("Cost: ",
      cost_function(p0, t, data[passer], passer, sigma))

#################################################################
# Do the minimisation starting from the second prediction
#################################################################

# Now, minimise based on the second test, which appears better

retval2016 = minimize(cost_function, p0, args=(t, data[passer],
                                                passer, sigma))

print(f"Value of the function at the minimum: {retval2016.fun:g}")
print(f"Value of the solution: {str(retval2016.x):s}")

#################################################################
# Plot the fitted model
#################################################################

plt.plot(t, dbl_sigmoid_function(retval2016.x, t), '-', lw=3,
         label="Fitted function")
plt.legend(loc="best")
plt.ylabel("LAI $m^{2}m^{-2}$")
plt.xlabel("DoY [d]")
```

```
Cost:  173.30984000080434
Cost:  89.70477658818041
Value of the function at the minimum: 43.7381
Value of the solution: [4.61179571e-01 5.04359055e+00 1.24979141e-01 1.26813992e+02
 9.04932760e-02 2.68361852e+02]
```

```
Text(0.5,0,'DoY [d]')
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```

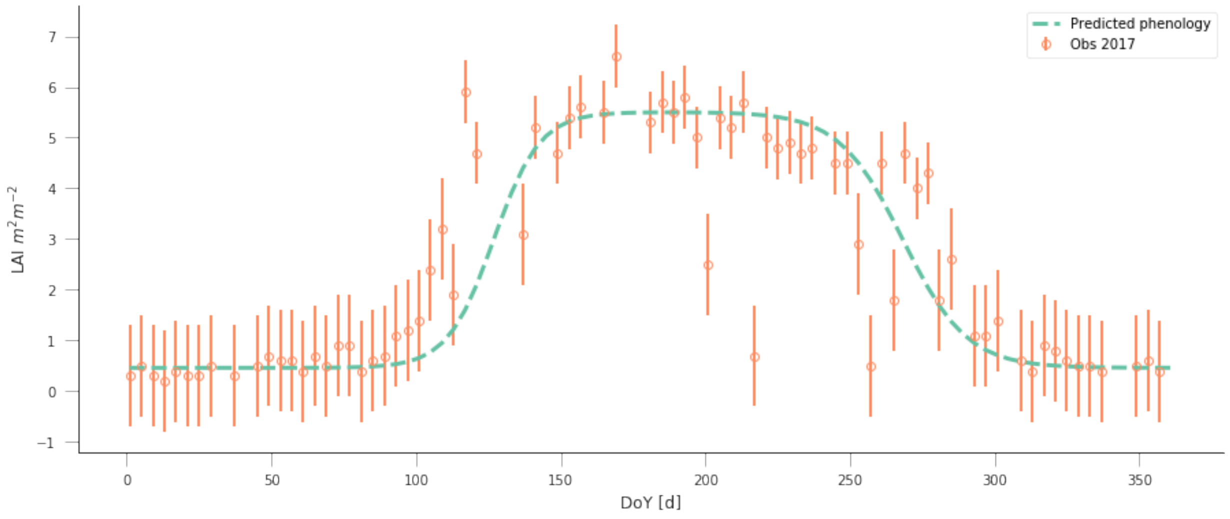


A model isn't very useful if you can't use it to make predictions. So let's just use the optimal solution to predict the LAI for 2017:

```
plt.figure(figsize=(15, 6))

year = 2017
lai_raster = f"data/euro_lai/Europe_mosaic_Lai_500m_{year:d}.tif"
qa_raster = f"data/euro_lai/Europe_mosaic_FparLai_QC_{year:d}.tif"

data = read_tseries(lai_raster, y_location, x_location)/10.
qa = read_tseries(qa_raster, y_location, x_location)
passer = get_sfc_qc(qa) <= 1
sigma = get_scaling(get_sfc_qc(qa))[passer]

t = np.arange(len(passer))*4 + 1

# Plot the data with uncertainty bars
plt.errorbar(t[passer], data[passer], yerr=sigma, fmt="o",
             mfc="none", label=f"Obs {year:d}")

# Print the fitted model
plt.plot(t, dbl_sigmoid_function(retval2016.x, t), '--', lw=3,
         label="Predicted phenology")
plt.legend(loc="best")
plt.ylabel("LAI $m^{2}m^{-2}$")
plt.xlabel("DoY [d]")
```

```
Text(0.5,0,'DoY [d]')
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



The results are very encouraging, but given the large error bars, and the paucity of data in spring and autumn, can we be sure? We could definitely compare the fit from last year to the fit from this year and see whether they're different. But it'll be hard to decide whether they are different or not if we don't have error bars in the parameters!

Fit the phenology model to the 2017 data, and comment on the optimal parameters

Do a similar experiment for other sites. In the benefit of efficiency, you could write a set of functions that would allow you to quickly do the entire process and relevant plots for different latitude/longitude points

Some quick notions on how to develop the functions:

You probably want a read data function, that returns LAI, `sigma` and `passer`.

A second function could take the data and a starting guess, and minimise it, returning the optimal parameters.

A third function would be in charge of plotting the data, and the predictions, so it could take the observations and uncertainties, etc., as well as a vector of parameters.

A final function could wrap the previous three and allow the user to select a year to fit to and a location

You can probably come up with interesting sites, but here are some that you **may** want to try

```
sites = [[43.015364, -6.703704],
         [51.775511, -1.336993],
         [43.487178, 1.283292]
        ]
```

Uncertainty

In the previous examples, we have seen that the model can be made to fit observations and used in prediction pretty well, but there's the obvious question of how exactly can we define the different parameters. Ideally, we'd like to have some error bars on e.g. the date and slope of the spring and autumn flanks, so that we can decide whether a shift has ocurred or not. Similarly for the maximum/minimum LAI. Intuitively, we can see that if we change the optimal parameters a bit, we'll get a solution that might still have an acceptable performance (meaning that it will probably go through all the error bars).

One way to test this is to do some Monte Carlo sampling around the optimal solution. We can evaluate the shape of the cost function around the optimum and that will give us an idea of the uncertainty: a cost function that changes very rapidly around the optimal point in one direction suggests that if you change the parameters by a small amount, the goodness of fit changes drastically, so that the parameter is very well and accurately defined. In contrast, if changing the parameter doesn't change the cost function value by much, we have an uncertain parameter.

The Metropolis-Hastings algorithm

A way to do this Monte Carlo sampling is to use the Metropolis-Hastings algorithm. This is a sequential method that proposes and accepts samples based on the likelihood value. Basically, if the cost function improves, the sample gets accepted, if it doesn't improve, then a uniform random number between 0 and 1 is drawn. If the ratio of proposed to previous likelihoods is greater than the random number, the samples gets accepted. This means that for solutions that don't improve the cost function, there's a chance that the algorithm will improve on them, meaning that it doesn't get trapped on local minima, and provides an exploration of the entire problem space.

In a nutshell, here's some **pseudocode** of the MH algorithm

```
1. Initialise $\vec{x}^0$.
2. For $i = 1$ to $i = N_{iterations}$:
    a. Sample a proposed new $\vec{x}^*$ as $\vec{x}^* = \vec{x}^{i-1} + \mathcal{N}(0, \Sigma)$.
    b. Calculate the **likelihood** associated with $\vec{x}^*$, $L(\vec{x}^*, i)$.
    c. Calculate the **likelihood ratio** $\alpha = \dfrac{L(\vec{x}^*, i)}{L(\vec{x}^*, i-1)}$.
    d. Draw a random uniform number between 0 and 1 $u = \mathcal{U}(0, 1)$
```

e. if $u \leq \min\{1, \alpha\}$:

- $\vec{x}^{i+1} = \vec{x}^{*}$: we accept the new proposal

f. else:

- $\vec{x}^{i+1} = \vec{x}^{i}$: we reject the new proposal

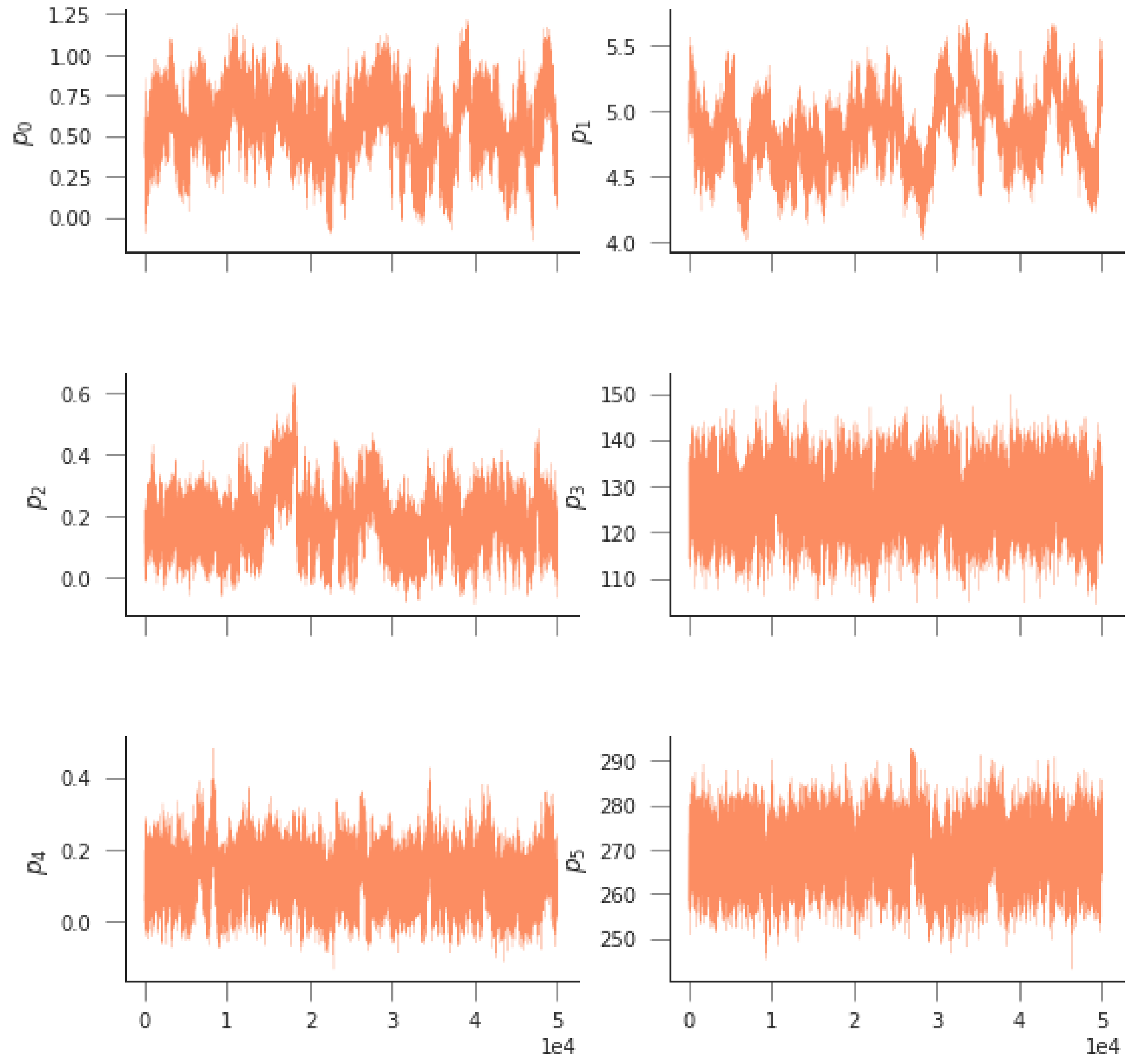
Examine the MH algorithm, and try to see whether you can see how you could go and implement it in Python. The pseudocode above presents you with the barebones recipe, and you will need to add a couple of extra ingredients. You can assume that you have available the **log-likelihood** function that we have described above, and you may need to use the `np.random.normal` and `np.random.rand` functions which respectively provide Gaussian random numbers and uniform random numbers between 0 and 1

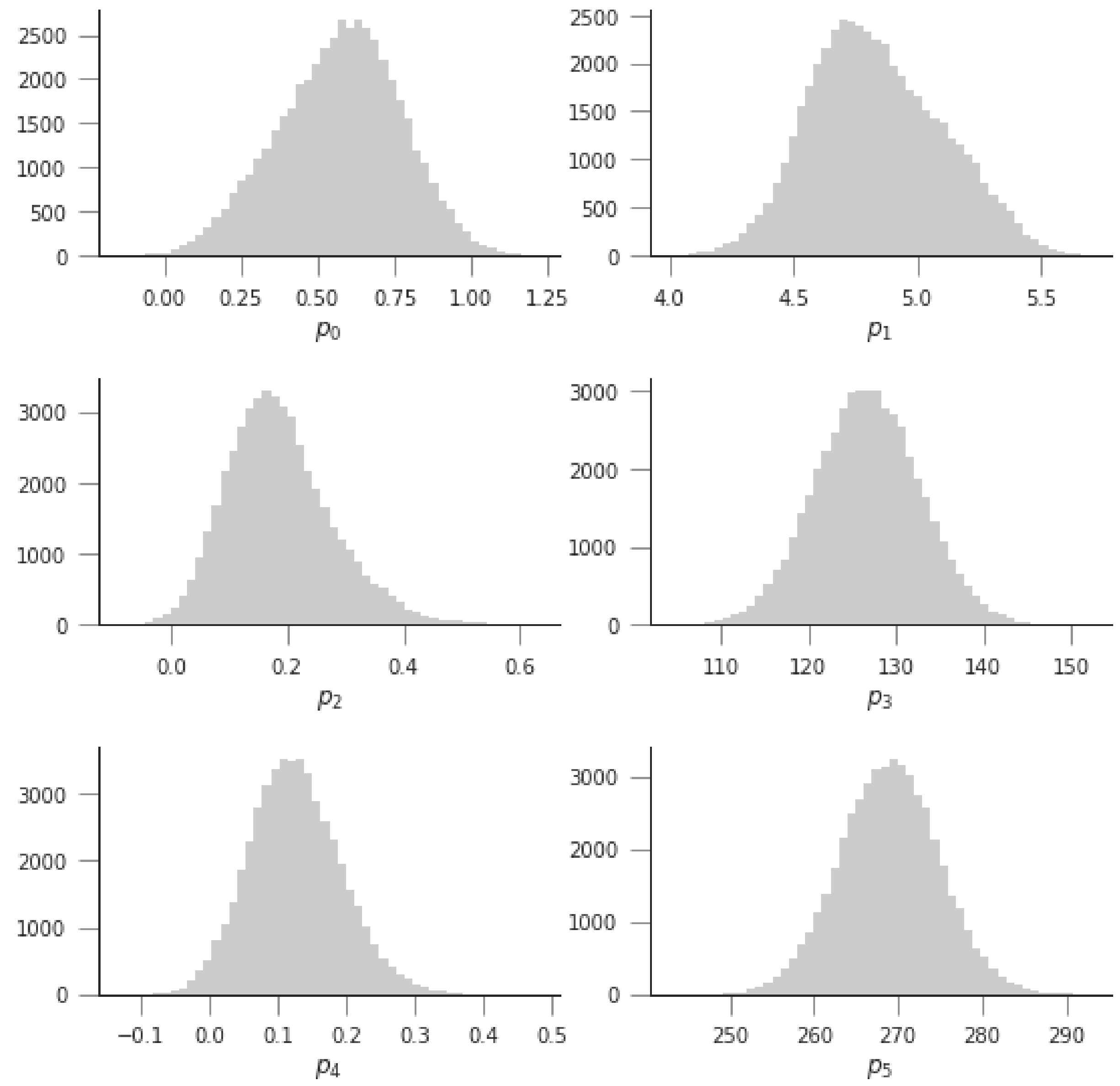
```python
y_location, x_location = 43.0156, -6.7038

def read_data(year, x_location, y_location):
    lai_raster = f"data/euro_lai/Europe_mosaic_Lai_500m_{year:d}.tif"
    qa_raster = f"data/euro_lai/Europe_mosaic_FparLai_QC_{year:d}.tif"
    print(lai_raster)
    data = read_tseries(lai_raster, y_location, x_location)/10.
    qa = read_tseries(qa_raster, y_location, x_location)
    passer = get_sfc_qc(qa) <= 1
    sigma = get_scaling(get_sfc_qc(qa))[passer]

    t = np.arange(len(passer))*4 + 1
    return t, data, sigma, passer

def lklhood(p, t, y_obs, passer, sigma_obs, func=dbl_sigmoid_function):
    y_pred = func(p, t)
    n = passer.sum()
    cost = -0.5* (y_pred[passer]-y_obs)**2/sigma_obs**2
    return cost.sum()


samples = metropolis_hastings(retval2016.x, 2016, x_location, y_location)
```

The previous code has produced samples of parameters that we can now visualise as “traces” as well as histograms. The shape of the histograms gives us some idea of the uncertainty of the parameters in their units. We can also use these samples and propagate them through the phenology model to produce an *ensemble* of model trajectories that define a region of uncertainty.

```python
fig, axs = plt.subplots(nrows=3, ncols=2, figsize=(9, 9), sharex=True)
axs = axs.flatten()
for i in range(len(retval2016.x)):
    axs[i].plot(samples[:, i], '-', lw=0.2)
    axs[i].set_ylabel(f"$p_{i}$")

fig, axs = plt.subplots(nrows=3, ncols=2, figsize=(9, 9))
axs = axs.flatten()
for i in range(len(retval2016.x)):
    axs[i].hist(samples[2000:, i], bins=50, color="0.8")
    axs[i].set_xlabel(f"$p_{i}$")
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```


p_0
p_1
p_2
p_3
p_4
p_5
1.25
1.00
0.75
0.50
0.25
0.00
5.5
5.0
4.5
4.0
0.6
0.4
0.2
0.0
150
140
130
120
110
0.4
0.2
0.0
290
280
270
260
250
0
1
2
3
4
5
1e4



```
n_samples = samples.shape[0]
y_location, x_location = 43.0156, -6.7038

t, data, sigma, passer = read_data(2016, x_location, y_location)

pred_lai = np.zeros((n_samples, len(t)))
for i in range(n_samples):
    pred_lai[i, :] = dbl_sigmoid_function(samples[i], t)


plt.figure(figsize=(15, 7))
pcntiles = np.percentile( pred_lai, [5, 25, 50, 75, 95], axis=0)
plt.fill_between(t, pcntiles[0], pcntiles[-1], color="0.9")
plt.fill_between(t, pcntiles[1], pcntiles[-2], color="0.7")
plt.plot(t, pcntiles[2], '--', lw=3, label="Median")
```

(continues on next page)

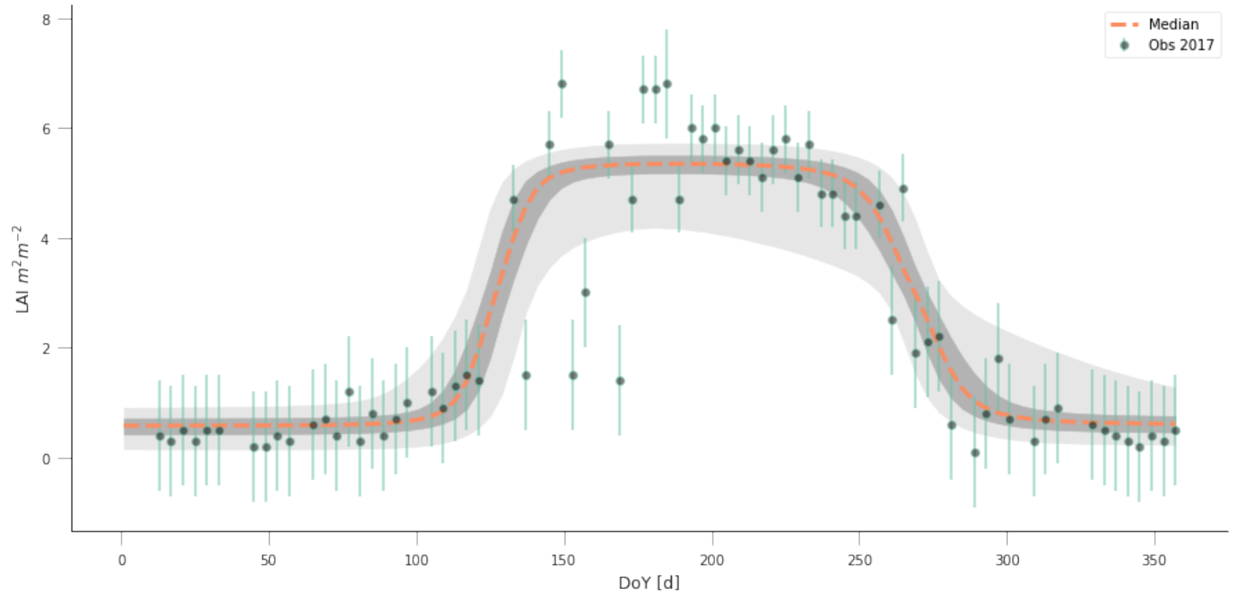
(continued from previous page)

```
plt.errorbar(t[passer], data[passer], yerr=sigma, fmt="o",
             mfc="none", label=f"Obs {year:d}", alpha=0.5)

plt.legend(loc="best")
plt.ylabel("LAI $m^{2}m^{-2}$")
plt.xlabel("DoY [d]")
```

```
data/euro_lai/Europe_mosaic_Lai_500m_2016.tif
```

```
Text(0.5,0,'DoY [d]')
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



```
# All imports go here. Run me first!
import datetime
from pathlib import Path  # Checks for files and so on
import numpy as np  # Numpy for arrays and so on
import pandas as pd
import sys
import matplotlib.pyplot as plt  # Matplotlib for plotting
# Ensure the plots are shown in the notebook
%matplotlib inline

import gdal
import osr
import numpy as np

from geog0111.geog_data import procure_dataset
%matplotlib inline
```

2.8.26 Fitting models of phenology to MODIS LAI data

In the previous Section, we have looked at a very simple model of phenology, probably applicable to a lot of deciduous vegetation. We have created some synthetic data, and we have tried to fit the model to these "pseudo observations". It is now the time to extract time series of MODIS data, and try to fit the model to them.

In this Section, we'll use some of the work we used before. In particular, we'll use a mosaic of LAI over Western Europe derived from the MODIS MCD15 product. The product has already been packed in an easy to use GeoTIFF file, together with the corresponding QA dataset. We'll define some functions here again (they are from other sections, but added here for simplicity):

The phenology model

The phenology model is the double logistic/sigmoid curve, given by

$$y = p_0 - p_1 \cdot \left[\frac{1}{1+\exp\left(p_2 \cdot (t-p_3)\right)} + \frac{1}{1+\exp\left(-p_4 \cdot (t-p_5)\right)} - 1 \right].$$

We also define a cost function for it based on observations of LAI, uncertainty and a mask to indicate missing observations.

```python
def dbl_sigmoid_function(p, t):
    """The double sigmoid function defined over t (where t is an array).
    Takes a vector of 6 parameters"""

    sigma1 = 1./(1+np.exp(p[2]*(t-p[3])))
    sigma2 = 1./(1+np.exp(-p[4]*(t-p[5])))
    y = p[0] - p[1]*(sigma1 + sigma2 - 1)
    return y

def cost_function(p, t, y_obs, passer, sigma_obs, func=dbl_sigmoid_function):
    y_pred = func(p, t)
    cost = -0.5* (y_pred[passer]-y_obs)**2/sigma_obs**2
    return -cost.sum()
```

Interpreting the QA LAI data

From before, we interpret the QA layer in the MODIS product by looking at bits 5 to 7, and turning them into a weight given by ϕ^{QA}, where ϕ is the golden ratio ($= 0.618\ldots$), and QA can take values between 0 and 3, indicating a decreasing quality in the retrieval, and hence, a lower weight.

```python
def get_sfc_qc(qa_data, mask57 = 0b11100000):
    sfc_qa = np.right_shift(np.bitwise_and(qa_data, mask57), 5)
    return sfc_qa

def get_scaling(sfc_qa, golden_ratio=0.61803398875):
    weight = np.zeros_like(sfc_qa, dtype=np.float)
    for qa_val in [0, 1, 2, 3]:
        weight[sfc_qa == qa_val] = np.power(golden_ratio, float(qa_val))
    return weight
```

Selecting data from a raster file

In this activity, we want to select a pixel in the map and read in the data for all time steps. This can be achieved by plotting the map with e.g. `imshow`, and judiciously selecting a row and a column. However, in many cases, we have locations of interest as a list of latitude and longitude pairs. These do not automatically map to rows and columns, because the the MODIS data are **projected** in a projection that is not latitude/longitude. So the process of going from latitude-longitude pair to row and column needs two steps: converting latitude-longitude coordinates into the coordinates of the raster of interest (in the case of MODIS, MODIS sinusoidal projection), and then converting the raster coordinates into pixel numbers.

Converting geographic projections

Converting from latitude-longitude pairs (or any other representation) into a different projection can be accomplished by GDAL (surprise!). We first need to find a way to defining the projection. There are several ways to do this:

- **EPSG codes** These are numerical codes that have been internationally agreed and fully define a projection
- **Proj4 strings** Proj4 is the library the manages coordinate conversions under the hood in GDAL. It has a method to define a projection as a text string.
- **WKT (Well-known text) format** This is a standard that defines the projection as a text block

Generally speaking, their simplicity of use recommends EPSG, a single number. In some cases, proj4 strings are best (e.g. for some product-specific projections), and WKT is generally used by other GIS software.

In any case, the spatialreference website provides a convenient "Rosetta stone" of projections in these different conventions.

Use spatialreference.org to find out what projection the EPSG code 4326 corresponds to

In Python, using the OSR part of the GDAL library, we define the source and destinations projections using `SpatialReference` objects, which are then populated with e.g. EPSG codes or proj4 strings:

```
import osr
# Define the Lat/Long object
wgs84 = osr.SpatialReference()
# In this case, we use EPSG code
wgs84.ImportFromEPSG(4326)
# Define the MODIS projection object
modis_sinu = osr.SpatialReference()
# In this case, we use the proj4 string
modis_sinu.ImportFromProj4("+proj=sinu +lon_0=0 +x_0=0 +y_0=0 " +
                           "+a=6371007.181 +b=6371007.181 +units=m +no_defs")
```

The previous code snippet defines two `SpatialReference` objects. These can be used to map from MODIS to/and from Latitude Longitude (or "WGS84") coordinates by using the `osr.CoordinateTransformation` object:

```
transformation = osr.CoordinateTransformation(wgs84, modis_sinu)
modis_x, modis_y, modis_z = transformation.TransformPoint(longitude,
                                                          latitude)
```

Clearly, changing the order of the parameters in `osr.CoordinateTransformation` would reverse the transformation.

Write some python code to convert the location of the Pearson Building (latitude: 51.524750 decimal degrees, longitude=-0.134560 decimal degrees) between WGS84 and OSGB 1936/British National Grid and UTM zone 30N/WGS84. Use mygeodata.cloud to test that your results are sensible

```
import osr

lat, lon = 51.524750, -0.134560
wgs84 = osr.SpatialReference()
```

(continues on next page)

(continued from previous page)

```
wgs84.ImportFromEPSG(4326)
osgb = osr.SpatialReference()
osgb.ImportFromEPSG(27700)
utm30n = osr.SpatialReference()
utm30n.ImportFromEPSG(32630)

transformation_osgb = osr.CoordinateTransformation(wgs84, osgb)
transformation_utm = osr.CoordinateTransformation(wgs84, utm30n)
print("GDAL OSGB: ", transformation_osgb.TransformPoint(lon, lat))
print("Expected OSGB: 529510.455794 182297.498731")
print("GDAL UTM30N/WGS84: ", transformation_utm.TransformPoint(lon, lat))
print("Expected UTM30N/WGS84: 698771.632126 5712074.37524")
```

```
GDAL OSGB:  (529510.4529047138, 182297.49865122623, -46.15269147325307)
Expected OSGB: 529510.455794 182297.498731
GDAL UTM30N/WGS84:  (698771.6321257285, 5712074.3752355995, 0.0)
Expected UTM30N/WGS84: 698771.632126 5712074.37524
```

Finding a pixel based on its coordinates

Geospatial data usually contain a definition of how to go from a coordinate to a pixel location. In GDAL, the generic way this is encoded is through the `GeoTransform` element, a six element vector that details the location of the **U**pper **L**eft corner of the raster file (pixel position (0, 0)), the pixel spacing, as well as a possible angular shift. Here are the elements of the geotransform array:

1. The Upper Left easting coordinate (i.e., horizontal)
2. The E-W pixel spacing
3. The rotation (0 degrees if image is “North Up”)
4. The Upper left northing coordinate (i.e., vertical)
5. The rotation (0 degrees)
6. The N-S pixel spacing, negative as we will be counting from the UL corner

With this in mind, and remembering that in Python arrays start at 0, and ignoring the rotation contributions, the pixel numbers can be calculated as follows

```
 pixel_x = (x_location - geo_transform[0])/geo_transform[1] \
     # The difference in distance between the UL corner (geot[0] \
     #and point of interest. Scaled by geot[1] to get pixel number

pixel_y = (y_location - geo_transform[3])/(geo_transform[5]) # Like for pixel_x, \
     #but in vertical direction. Note the different elements of geot \
     #being used
```

Since it’s easy to get this wrong, GDAL provides a couple of methods to do this conversion directly:

```
inv_geoT = gdal.InvGeoTransform(geotransform)
r, c = (gdal.ApplyGeoTransform(inv_geoT, x_location, y_location))
```

Let’s see a whole example of this zooming in the fAPAR map from the MODIS MCD15 product near the fine city of A Coruña in Galicia, NW Spain (latitude: 43.3623, longitude: -8.4115):

```
%%html
<iframe src="https://www.google.com/maps/embed?pb=!1m14!1m12!1m3!1d238659.69294928786!
↪2d-8.664931741126212!3d43.39317238062582!2m3!1f0!2f0!3f0!3m2!1i1024!2i768!4f13.1!
↪5e1!3m2!1sen!2suk!4v1542817273641" width="600" height="450" frameborder="0" style=
↪"border:0" allowfullscreen></iframe>
```

```
#################################################################
# Define transformations and variables. This is like above!
#################################################################

y_location, x_location = 43.3623, -8.4115 # In degs
# Define the Lat/Long object
wgs84 = osr.SpatialReference()
# In this case, we use EPSG code
wgs84.ImportFromEPSG(4326)
# Define the MODIS projection object
modis_sinu = osr.SpatialReference()
# In this case, we use the proj4 string
modis_sinu.ImportFromProj4("+proj=sinu +lon_0=0 +x_0=0 +y_0=0 " +
                           "+a=6371007.181 +b=6371007.181 +units=m +no_defs")

transformation = osr.CoordinateTransformation(wgs84, modis_sinu)
modis_x, modis_y, modis_z = transformation.TransformPoint(x_location,
                                                          y_location)
print("MODIS coordinates: ", modis_x, modis_y)

#################################################################
# We use a random file in the UCL filesystem
#################################################################

fname = "/home/plewis/public_html/geog0111_data/lai_files/" + \
              "MCD15A3H.A2016273.h17v04.006.2016278070708.hdf"
g = gdal.Open('HDF4_EOS:EOS_GRID:"%s":MOD_Grid_MCD15A3H:Fpar_500m' % fname)


#################################################################
# This is where new stuff begins
# Find out the pixel location from the MODIS Easting & Northing
#################################################################

geoT = g.GetGeoTransform()
inv_geoT = gdal.InvGeoTransform(geoT)
r, c = (gdal.ApplyGeoTransform(inv_geoT, modis_x, modis_y))
r = int(r+0.5)
c = int(c+0.5)
print("Pixel location: ", r,c)

#################################################################
# Now, read in the data, and plot it
#################################################################

fapar = g.ReadAsArray()/100
fapar[fapar>1] = np.nan
cmap = plt.cm.inferno
cmap.set_bad("0.6")
plt.figure(figsize=(8, 8))
plt.imshow(fapar, interpolation="nearest", vmin=0, vmax=1, cmap=cmap)
```

(continues on next page)

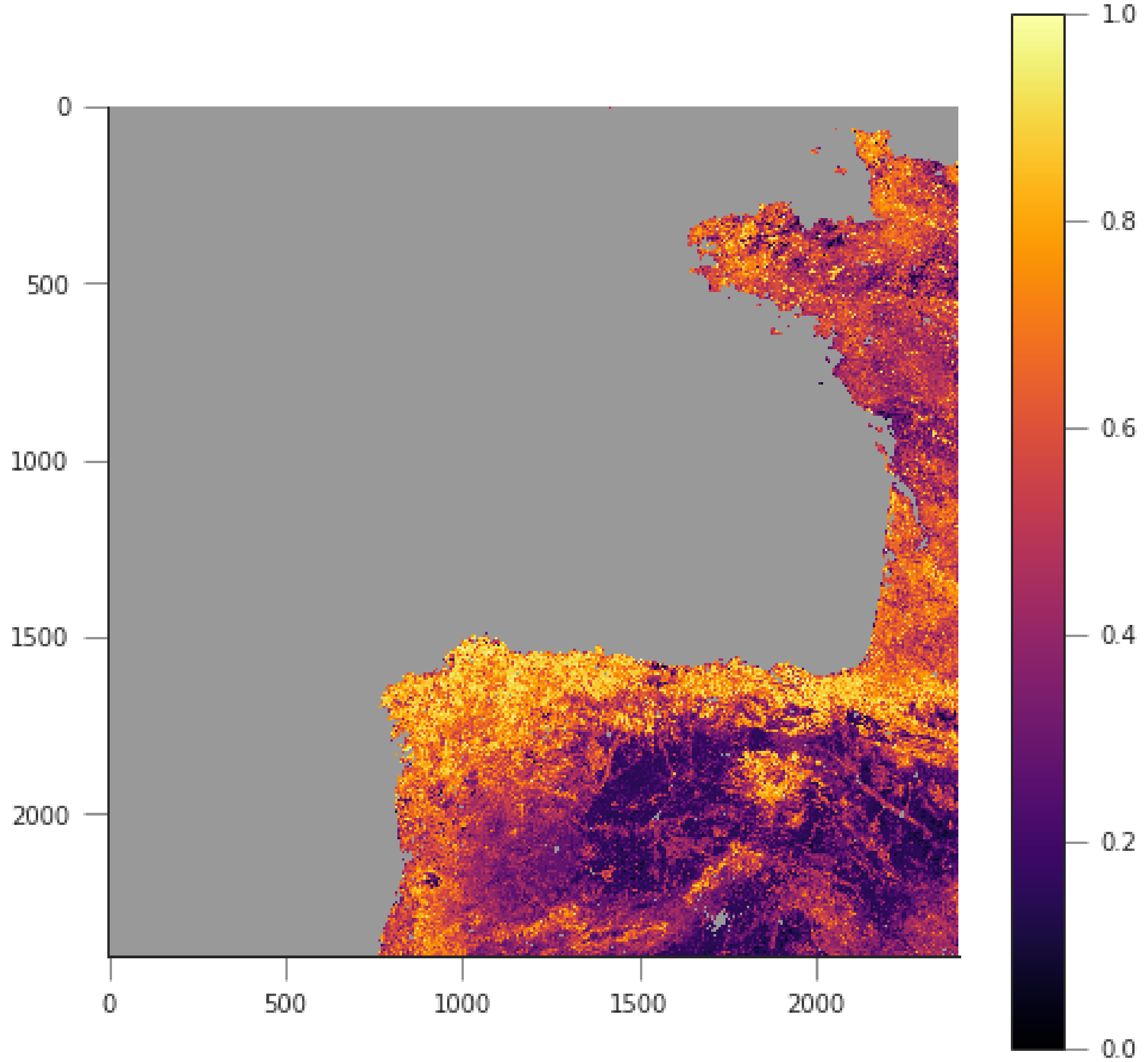
(continued from previous page)

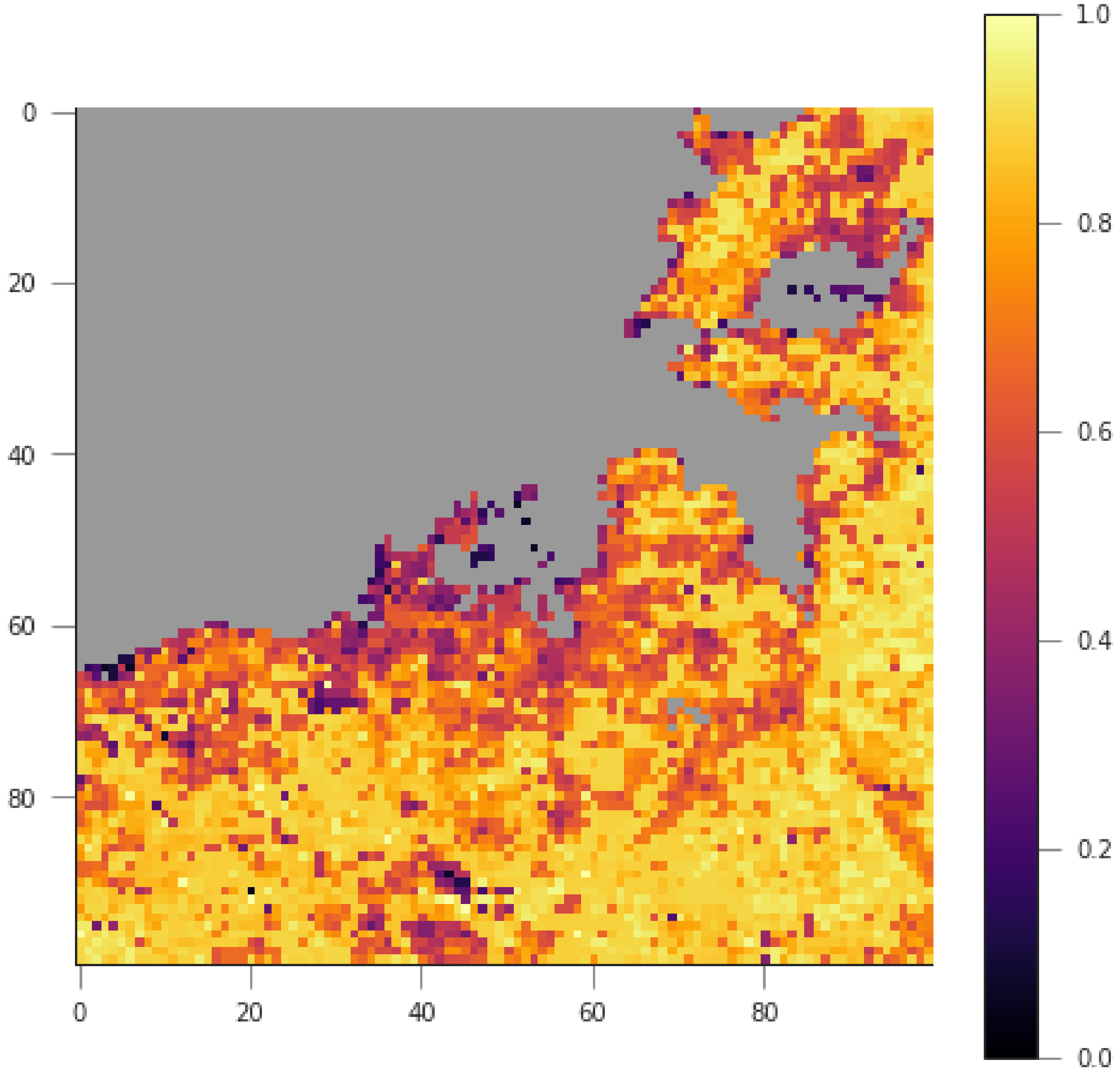
```
plt.colorbar()

##################################################################
# Plot a zoomed-in version
##################################################################


plt.figure(figsize=(8, 8))
plt.imshow(fapar[(c-50):(c+50), (r-50):(r+50)], interpolation="nearest",
           vmin=0, vmax=1, cmap=cmap)
plt.colorbar()
```

```
MODIS coordinates:  -680000.4782137175 4821673.202327191
Pixel location:  932 1593
```

```
<matplotlib.colorbar.Colorbar at 0x7f8ffdcf6390>
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```


0
500
1000
1500
2000
0
500
1000
1500
2000
1.0
0.8
0.6
0.4
0.2
0.0



We can see that in the example above, we're getting the right pixel number. Clearly, the code above is a bit of a mess, and needs to be cleaned up, split into functions and tested. This is an example, and you can take this as a reference of how to document functions etc.

```
def convert_coordinates(x_location, y_location,
                        src_transform={'EPSG':4326},
                        dst_transform={'Proj4':
                                       "+proj=sinu +lon_0=0 +x_0=0 " +
                                       "+y_0=0 +a=6371007.181 " +
                                       "+b=6371007.181 +units=m +no_defs"
                                      }):
    """A function to convert coordinates from one target coordinate
    representation to another. The input an output transformation can be given
    in either EPSG codes or Proj4 strings, by providing the function with a
    dictionary with the desired convention as a key, and with the relevant
```

(continues on next page)

(continued from previous page)

```
    codes as its only element.

    Parameters
    ----------
    x_location: float
        The x location
    y_location: float
        The y location
    src_transform: dict
        A dictionary with keys either "EPSG" or "Proj4" (anything else throws
        an exception) with the description of the **input** projection
    dst_transform: dict
        A dictionary with keys either "EPSG" or "Proj4" (anything else throws
        an exception) with the description of the **output** projection
    Returns
    --------

    The transformed x and y coordinates"""
    input_coords = osr.SpatialReference()
    # In this case, we use EPSG code
    try:
        input_coords.ImportFromEPSG(src_transform["EPSG"])
    except KeyError:
        input_coords.ImportFromProj4(src_transform["Proj4"])
    except KeyError:
        raise ValueError("src_transform not dictionary with EPSG/Proj4 keys!")


    output_coords = osr.SpatialReference()
    try:
        output_coords.ImportFromEPSG(dst_transform["EPSG"])
    except KeyError:
        output_coords.ImportFromProj4(dst_transform["Proj4"])
    except KeyError:
        raise ValueError("src_transform not dictionary with EPSG/Proj4 keys!")


    transformation = osr.CoordinateTransformation(input_coords,
                                                  output_coords)
    output_x, output_y, output_z = transformation.TransformPoint(x_location,
                                                          y_location)
    return output_x, output_y


##################################################################
# Test function
##################################################################

y_location, x_location = 43.3623, -8.4115 # In WGS84
print (convert_coordinates(x_location, y_location))
```

```
(-680000.4782137175, 4821673.202327191)
```

```
def get_pixel(raster, point_x, point_y):
    """Get the pixel for given coordinates (in the raster's convention, not
    checked!) for a raster file.
```

(continues on next page)

(continued from previous page)

```
    Parameters
    ----------
    raster: string
        A GDAL-friendly raster filename
    point_x: float
        The Easting in the same coordinates as the raster (not checked!)
    point_y: float
        The Northing in the same coordinates as the raster (not checked!)

    Returns
    -------
    The row/column (or column/row, depending on how you define it)
    """
    g = gdal.Open(raster)
    if g is None:
        raise ValueError(f"{raster:s} cannot be opened!")
    geoT = g.GetGeoTransform()
    inv_geoT = gdal.InvGeoTransform(geoT)
    r, c = (gdal.ApplyGeoTransform(inv_geoT, point_x, point_y))
    return int(r + 0.5), int(c + 0.5)

###################################################################
# Test function
###################################################################


fname = "/home/plewis/public_html/geog0111_data/lai_files/" + \
            "MCD15A3H.A2016273.h17v04.006.2016278070708.hdf"
gdal_fname = 'HDF4_EOS:EOS_GRID:"%s":MOD_Grid_MCD15A3H:Fpar_500m' % fname
print (get_pixel(gdal_fname, -680000.4782137175, 4821673.202327191))
```

```
(932, 1593)
```

Retrieving a time series from a multi-band raster

We have produced 4 rasters, with the LAI value for 2016 and 2017, as well as the correspodingn `FparLai_QC` layer. They're avaialable in `data/euro_lai`. Let's quickly have a look at the data:

```
success = procure_dataset("euro_lai", destination_folder="data/euro_lai/",
                         verbose=True)
if not success:
    print("Something happened copying files across to data/euro_lai")

print(gdal.Info("data/euro_lai/Europe_mosaic_Lai_500m_2017.tif").split("\n")[:10])
```

```
Running on UCL's Geography computers
trying /archive/rsu_raid_0/plewis/public_html/geog0111_data
trying /data/selene/ucfajlg/geog0111_data/lai_data/
trying /data/selene/ucfajlg/geog0111_data/
Linking /data/selene/ucfajlg/geog0111_data/euro_lai/Europe_mosaic_FparLai_QC_2016.tif␣
↪to data/euro_lai/Europe_mosaic_FparLai_QC_2016.tif
Linking /data/selene/ucfajlg/geog0111_data/euro_lai/Europe_mosaic_FparLai_QC_2017.tif␣
↪to data/euro_lai/Europe_mosaic_FparLai_QC_2017.tif
```

(continues on next page)

(continued from previous page)

```
Linking /data/selene/ucfajlg/geog0111_data/euro_lai/Europe_mosaic_Lai_500m_2016.tif␣
↪to data/euro_lai/Europe_mosaic_Lai_500m_2016.tif
Linking /data/selene/ucfajlg/geog0111_data/euro_lai/Europe_mosaic_Lai_500m_2017.tif␣
↪to data/euro_lai/Europe_mosaic_Lai_500m_2017.tif
['Driver: GTiff/GeoTIFF', 'Files: data/euro_lai/Europe_mosaic_Lai_500m_2017.tif',
↪'Size is 4800, 4800', 'Coordinate System is:', 'PROJCS["unnamed",', '    GEOGCS[
↪"Unknown datum based upon the custom spheroid",', '        DATUM["Not_specified_
↪based_on_custom_spheroid",', '            SPHEROID["Custom spheroid",6371007.181,
↪0]],', '        PRIMEM["Greenwich",0],', '        UNIT["degree",0.
↪0174532925199433]],']
```

We have 90 (or 91) layers, from day 1 to day 360/364 in the year. While we could read all the data in memory, it's wasteful of resources, and we might as well try to read in all the bands for a given pixel.

We can do this with the `read_tseries` function below. Basically, we this function calls the previous pixel-location functions, and then reads the entire time series for a pixel in one go. The function is defined below:

```python
#def get_pixel(raster, point_x, point_y):
#def convert_coordinates(x_location, y_location,


def read_tseries(raster, lat, long):
    """Read a time series (or all bands) for a raster file given latitude and
    longitude coordinates.
    **NOTE** Only works with Byte/UInt8 data types!
    """
    g = gdal.Open(raster)
#   px, py = get_pixel(raster, *convert_coordinates(x_location,
#                                                    y_location))
#    tmpx, tmpy =convert_coordinates(long, lat)
#    px, py = get_pixel(raster, tmpx, tmpy)

    px, py = get_pixel(raster, *convert_coordinates(long, lat))
    print(px, py)
    if 0 <= px >= g.RasterXSize:
        raise ValueError(f"Point outside of raster ({px:d}/{g.RasterXSize:d})")
    if 0 <= py >= g.RasterYSize:
        raise ValueError(f"Point outside of raster ({py:d}/{g.RasterYSize:d})")

    xbuf = 1
    ybuf = 1
    n_doys = g.RasterCount
    buf = g.ReadRaster (px, py,
                xbuf, ybuf, buf_xsize=xbuf, buf_ysize=ybuf,
                band_list=np.arange (1, n_doys+1))
    data = np.frombuffer ( buf, dtype=np.uint8)
    return data
```

The Nature reserve of Muniellos (43.0156, -6.7038) is mostly populated by *Quercus Robur*, which shows a strong phenology. It should be a good test point to see whether the data we have is sensible or not.

Using the provided function, plot the time series of LAI over the Muniellos Reserve for 2016 and 2017. Extra points for using QA flags to filter the data

```
%%html

<div>
```

(continues on next page)

(continued from previous page)

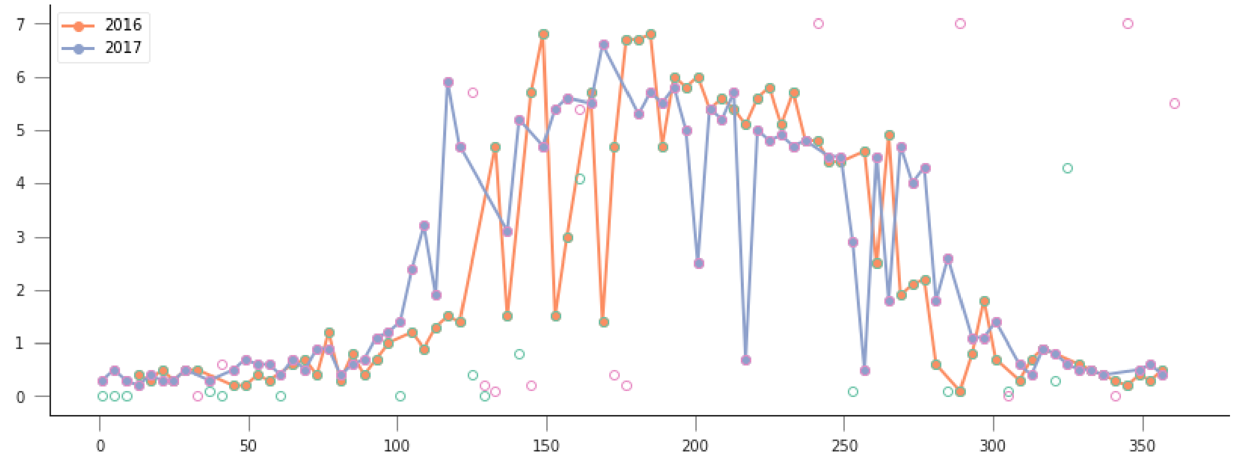
```
     <iframe width="500" height="400" frameborder="0" src="https://www.bing.com/maps/
↪embed?h=400&w=500&cp=43.029897999999996~-6.734863999999996&lvl=11&typ=d&sty=h&
↪src=SHELL&FORM=MBEDV8" scrolling="no">
     </iframe>
     <div style="white-space: nowrap; text-align: center; width: 500px; padding: 6px␣
↪0;">
        <a id="largeMapLink" target="_blank" href="https://www.bing.com/maps?cp=43.
↪029897999999996~-6.734863999999996&amp;sty=h&amp;lvl=11&amp;FORM=MBEDLD">View␣
↪Larger Map</a> &nbsp; | &nbsp;
        <a id="dirMapLink" target="_blank" href="https://www.bing.com/maps/directions?
↪cp=43.029897999999996~-6.734863999999996&amp;sty=h&amp;lvl=11&amp;rtp=~pos.43.
↪029897999999996_-6.734863999999996____&amp;FORM=MBEDLD">Get Directions</a>
    </div>
</div>~
```

```
y_location, x_location = 43.0156, -6.7038
plt.figure(figsize=(14,5))
for year in [2016, 2017]:
    lai_raster = f"data/euro_lai/Europe_mosaic_Lai_500m_{year:d}.tif"
    qa_raster = f"data/euro_lai/Europe_mosaic_FparLai_QC_{year:d}.tif"

    data = read_tseries(lai_raster, y_location, x_location)
    qa = read_tseries(qa_raster, y_location, x_location)

    qa = get_sfc_qc(qa)
    tx = np.arange(len(qa))*4. + 1
    plt.plot(tx[qa<=1], data[qa<=1]/10., 'o-', label=year)
    plt.plot(tx[qa<=3], data[qa<=3]/10., 'o', mfc="none")
plt.legend(loc="best")
```

```
1224 4076
1224 4076
1224 4076
1224 4076
```

```
<matplotlib.legend.Legend at 0x7f8ffdbb94e0>
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



We can now try to fit our double logistic model to the observations, weighted by their uncertainty. We make use the previously defined functions for the model and the cost function that we defined above. We will start by fitting the data to 2016, but will also try to "eyeball" a good starting point for the optimisation. And obviously, we'll want some plots. . .

A nice way to plot points with errorbars is (surprisingly enought) the ``plt.errorbar` method <https://matplotlib.org/api/_as_gen/matplotlib.pyplot.errorbar.html>`__. It's like the `plt.plot` method, but it also takes a `yerr` (or `xerr`) keyword with the extent of the error in the y direction.

```
from scipy.optimize import minimize

plt.figure(figsize=(15, 6))


lat, long = 43.0156, -6.7038
#################################################################
# Start by reading in the data
#################################################################
# Filenames
year = 2016
lai_raster = f"data/euro_lai/Europe_mosaic_Lai_500m_{year:d}.tif"
qa_raster = f"data/euro_lai/Europe_mosaic_FparLai_QC_{year:d}.tif"
# Actually read the data
data = read_tseries(lai_raster, lat, long)/10. # Read LAI
qa = read_tseries(qa_raster, lat, long) # Read QA/QC
# We only want to use QA flags 0 or 1
passer = get_sfc_qc(qa) <= 1
# This is the uncertainty
sigma = get_scaling(get_sfc_qc(qa))[passer]
# This is the time axis: every 4 days
t = np.arange(len(passer))*4 + 1

#################################################################
# Plot the observations of LAI with uncertainty bands
#################################################################

plt.errorbar(t[passer], data[passer], yerr=sigma, fmt="o",
             mfc="none", label=f"Obs {year:d}")

#################################################################
# Plot a first prediction with some random model parameters
```

(continues on next page)

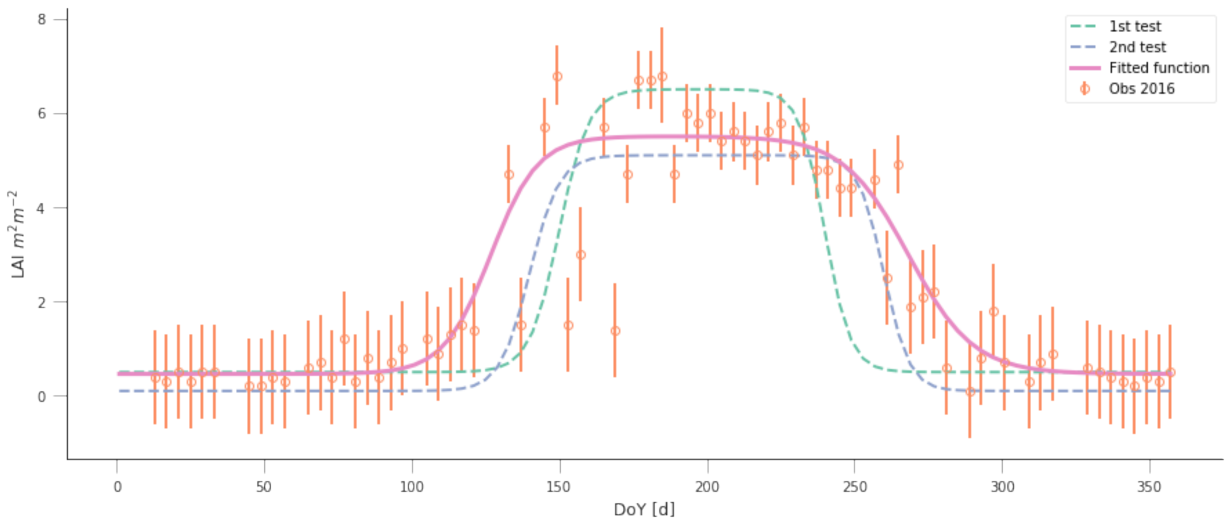
(continued from previous page)

```
#################################################################
# First eyeballing test:
p0 = np.array([0.5, 6, 0.2, 150, 0.23, 240])
plt.plot(t, dbl_sigmoid_function(p0, t), '--', label="1st test")
print("Cost: ",
      cost_function(p0, t, data[passer], passer, sigma))

#################################################################
# Plot a second, more refined prediction
#################################################################

# Second eyeballing test:
p0 = np.array([0.1, 5, 0.2, 140, 0.23, 260])
plt.plot(t, dbl_sigmoid_function(p0, t), '--', label="2nd test")
print("Cost: ",
      cost_function(p0, t, data[passer], passer, sigma))

#################################################################
# Do the minimisation starting from the second prediction
#################################################################

# Now, minimise based on the second test, which appears better

retval2016 = minimize(cost_function, p0, args=(t, data[passer],
                                              passer, sigma))

print(f"Value of the function at the minimum: {retval2016.fun:g}")
print(f"Value of the solution: {str(retval2016.x):s}")

#################################################################
# Plot the fitted model
#################################################################

plt.plot(t, dbl_sigmoid_function(retval2016.x, t), '-', lw=3,
         label="Fitted function")
plt.legend(loc="best")
plt.ylabel("LAI $m^{2}m^{-2}$")
plt.xlabel("DoY [d]")
```

```
1224 4076
1224 4076
Cost:  173.30984000080434
Cost:  89.70477658818041
Value of the function at the minimum: 43.7381
Value of the solution: [4.61179375e-01 5.04359058e+00 1.24979192e-01 1.26813991e+02
 9.04932708e-02 2.68361855e+02]
```

```
Text(0.5,0,'DoY [d]')
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



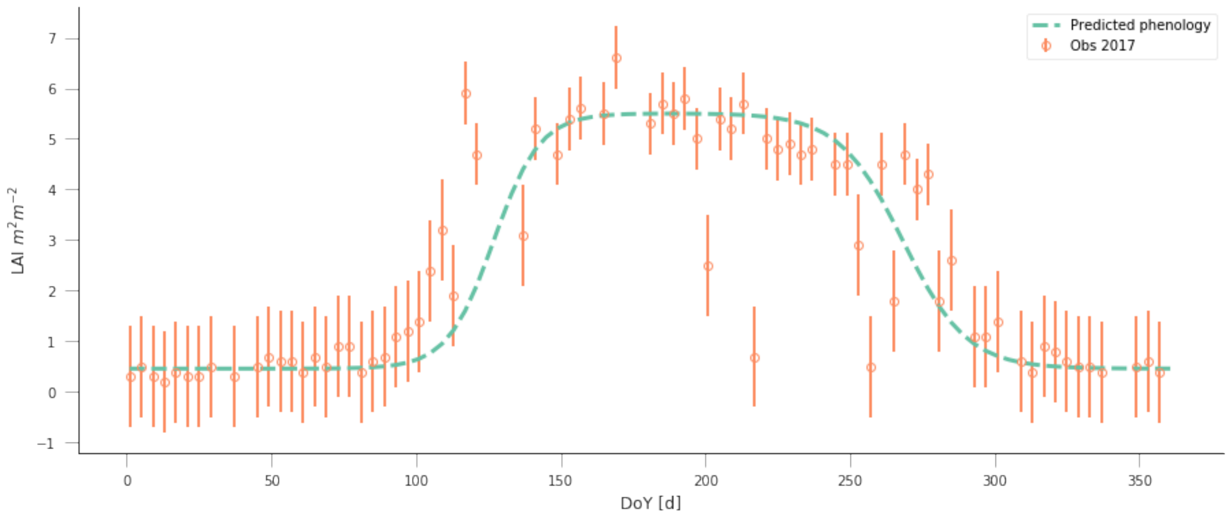
A model isn't very useful if you can't use it to make predictions. So let's just use the optimal solution to predict the LAI for 2017:

```
plt.figure(figsize=(15, 6))

year = 2017
lai_raster = f"data/euro_lai/Europe_mosaic_Lai_500m_{year:d}.tif"
qa_raster = f"data/euro_lai/Europe_mosaic_FparLai_QC_{year:d}.tif"

data = read_tseries(lai_raster, lat, long)/10. # Read LAI
qa = read_tseries(qa_raster, lat, long) # Read QA/QC
passer = get_sfc_qc(qa) <= 1
sigma = get_scaling(get_sfc_qc(qa))[passer]

t = np.arange(len(passer))*4 + 1

# Plot the data with uncertainty bars
plt.errorbar(t[passer], data[passer], yerr=sigma, fmt="o",
             mfc="none", label=f"Obs {year:d}")

# Print the fitted model
plt.plot(t, dbl_sigmoid_function(retval2016.x, t), '--', lw=3,
         label="Predicted phenology")
plt.legend(loc="best")
plt.ylabel("LAI $m^{2}m^{-2}$")
plt.xlabel("DoY [d]")
```

```
1224 4076
1224 4076
```

```
Text(0.5,0,'DoY [d]')
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



The results are very encouraging, but given the large error bars, and the paucity of data in spring and autumn, can we be sure? We could definitely compare the fit from last year to the fit from this year and see whether they're different. But it'll be hard to decide whether they are different or not if we don't have error bars in the parameters!

Fit the phenology model to the 2017 data, and comment on the optimal parameters

```
plt.figure(figsize=(15, 6))

year = 2017
lai_raster = f"data/euro_lai/Europe_mosaic_Lai_500m_{year:d}.tif"
qa_raster = f"data/euro_lai/Europe_mosaic_FparLai_QC_{year:d}.tif"

data = read_tseries(lai_raster, lat, long)/10. # Read LAI
qa = read_tseries(qa_raster, lat, long) # Read QA/QC
passer = get_sfc_qc(qa) <= 1
sigma = get_scaling(get_sfc_qc(qa))[passer]

t = np.arange(len(passer))*4 + 1

# Plot the data with uncertainty bars
plt.errorbar(t[passer], data[passer], yerr=sigma, fmt="o",
             mfc="none", label=f"Obs {year:d}")

# Print the fitted model
plt.plot(t, dbl_sigmoid_function(retval2016.x, t), '--', lw=3,
         label="Predicted phenology")
print(f"Value from previous year: {str(retval2016.x):s}")
retval2017 = minimize(cost_function, p0, args=(t, data[passer], passer,
                                                sigma))


print(f"Value of the function at the minimum: {retval2017.fun:g}")
print(f"Value of the solution: {str(retval2017.x):s}")


# Print the fitted model
plt.plot(t, dbl_sigmoid_function(retval2017.x, t), '-', lw=3,
         label="Fitted function")
```

(continues on next page)

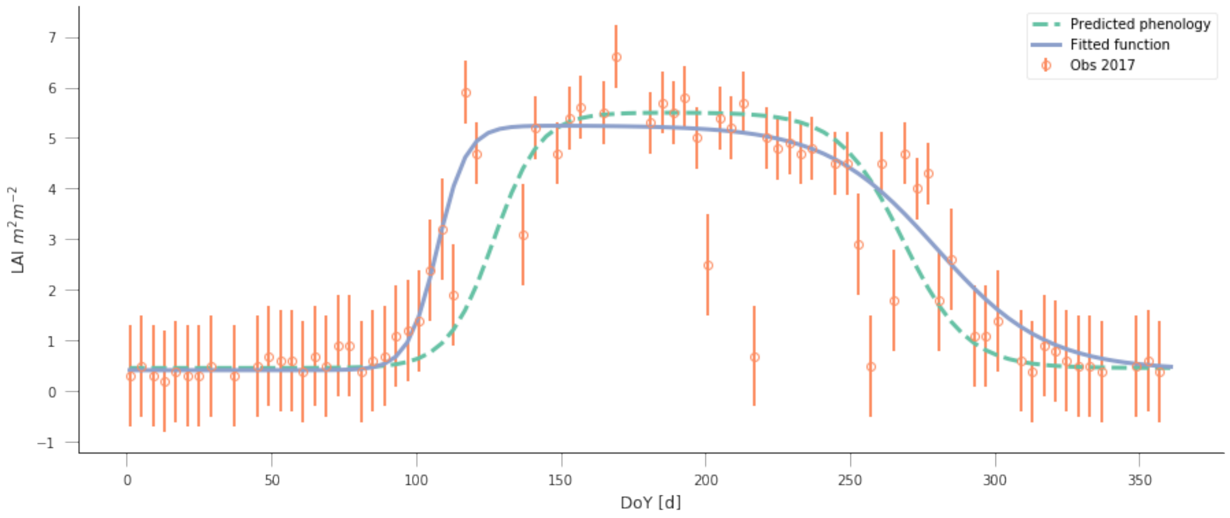
(continued from previous page)

```
plt.legend(loc="best")
plt.ylabel("LAI $m^{2}m^{-2}$")
plt.xlabel("DoY [d]")
```

```
1224 4076
1224 4076
Value from previous year: [4.61179375e-01 5.04359058e+00 1.24979192e-01 1.26813991e+02
 9.04932708e-02 2.68361855e+02]
Value of the function at the minimum: 41.6294
Value of the solution: [4.17809359e-01 4.83020921e+00 1.97608815e-01 1.07388496e+02
 5.23646758e-02 2.79806238e+02]
```

```
Text(0.5,0,'DoY [d]')
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



Do a similar experiment for other sites. In the benefit of efficiency, you could write a set of functions that would allow you to quickly do the entire process and relevant plots for different latitude/longitude points

Some quick notions on how to develop the functions:

You probably want a read data function, that returns LAI, `sigma` and `passer`.

A second function could take the data and a starting guess, and minimise it, returning the optimal parameters.

A third function would be in charge of plotting the data, and the predictions, so it could take the observations and uncertainties, etc., as well as a vector of parameters.

A final function could wrap the previous three and allow the user to select a year to fit to and a location

You can probably come up with interesting sites, but here are some that you **may** want to try

```
sites = [[43.015364, -6.703704],
         [51.775511, -1.336993],
         [43.487178, 1.283292]
        ]
```

I have fished out some suitable location from the paper from Wingate et al (2015), and you can see the results of fitting to the model to (some of) the sites below:

```
def read_data(year, x_location, y_location):
    """A function to read in data from the given LAI and QC maps. The
    function also converts the QA data into a `passer` (on/off) array
    to filter the really bad observations, and also creates `sigma`,
    the pseudo-uncertainty.

    Parameters
    ------------
    x_location: float
        The longitude in decimal degrees.
    y_location: float
        The latitude in decimal degrees.

    Returns
    ---------

    A time arrray, the data (LAI, scaled properly), the uncertainty and the
    `passer` mask
    """
    lai_raster = f"data/euro_lai/Europe_mosaic_Lai_500m_{year:d}.tif"
    qa_raster = f"data/euro_lai/Europe_mosaic_FparLai_QC_{year:d}.tif"
    print(lai_raster)
    data = read_tseries(lai_raster, y_location, x_location)/10.
    qa = read_tseries(qa_raster, y_location, x_location)
    passer = get_sfc_qc(qa) <= 1
    sigma = get_scaling(get_sfc_qc(qa))[passer]

    t = np.arange(len(passer))*4 + 1
    return t, data, sigma, passer

def fit_data(t, data, sigma, passer,
             cost_f=cost_function,
             p0= np.array([0.1, 5, 0.2, 140, 0.23, 260])):
    """Fits the data using scipy.optimize `minimize` function. It's a wrapper
    around a cost function given by `cost_f` (by default, it)"""
    retval = minimize(cost_f, p0, args=(t, data[passer], passer, sigma))
    return retval

def do_plots(t, data, sigma, passer, p):
    """Do some plots"""

    # Plot the data with uncertainty bars
    plt.errorbar(t[passer], data[passer], yerr=sigma, fmt="o",
              mfc="none", label=f"Obs {year:d}")
    # Print the fitted model
    plt.plot(t, dbl_sigmoid_function(p, t), '--', lw=3,
          label="Predicted phenology")
    plt.legend(loc="best")
    plt.ylabel("LAI $m^{2}m^{-2}$")
    plt.xlabel("DoY [d]")

def fit_data_year(fit_year, val_year, x_location, y_location):
    plt.figure(figsize=(12, 5))
    t, data, sigma, passer = read_data(fit_year, x_location,
                                       y_location)
```

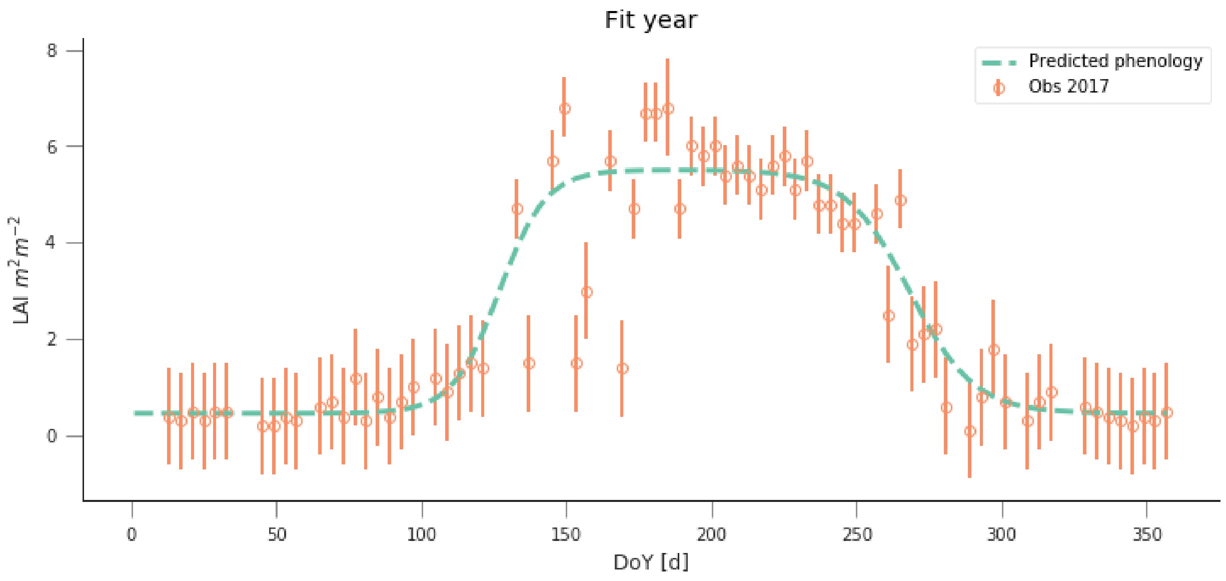
(continues on next page)

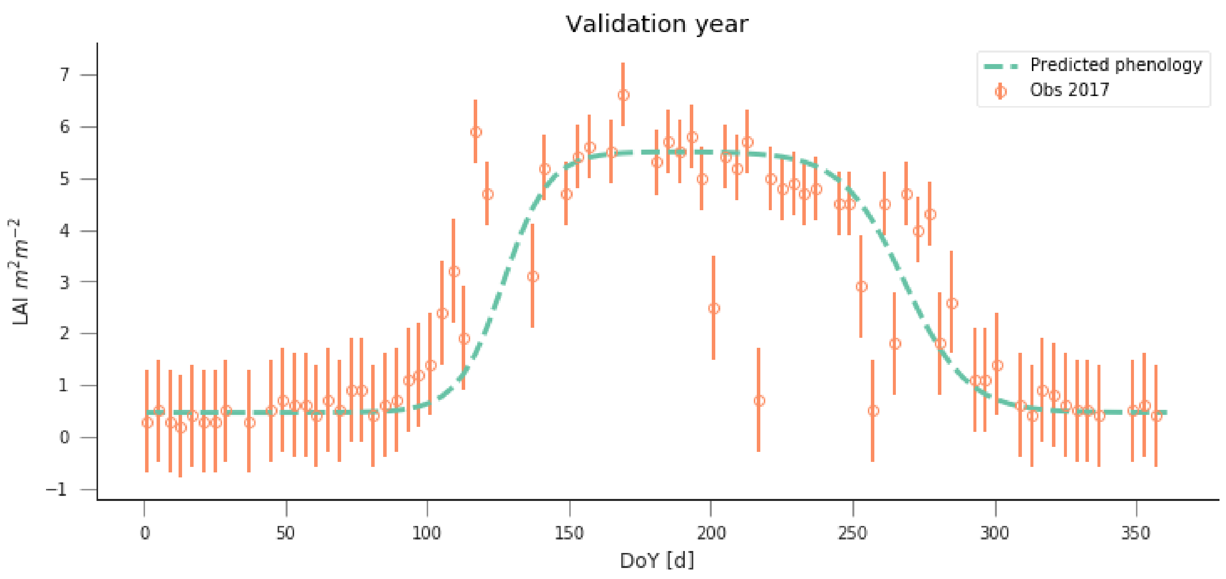
(continued from previous page)

```
    optimal_p = fit_data(t, data, sigma, passer)
    do_plots(t, data, sigma, passer, optimal_p.x)
    t, data, sigma, passer = read_data(val_year, x_location,
                                       y_location)
    plt.title("Fit year")
    plt.figure(figsize=(12, 5))
    do_plots(t, data, sigma, passer, optimal_p.x)
    plt.title("Validation year")


fit_data_year(2016, 2017, x_location, y_location)
```

```
data/euro_lai/Europe_mosaic_Lai_500m_2016.tif
1224 4076
1224 4076
data/euro_lai/Europe_mosaic_Lai_500m_2017.tif
1224 4076
1224 4076
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



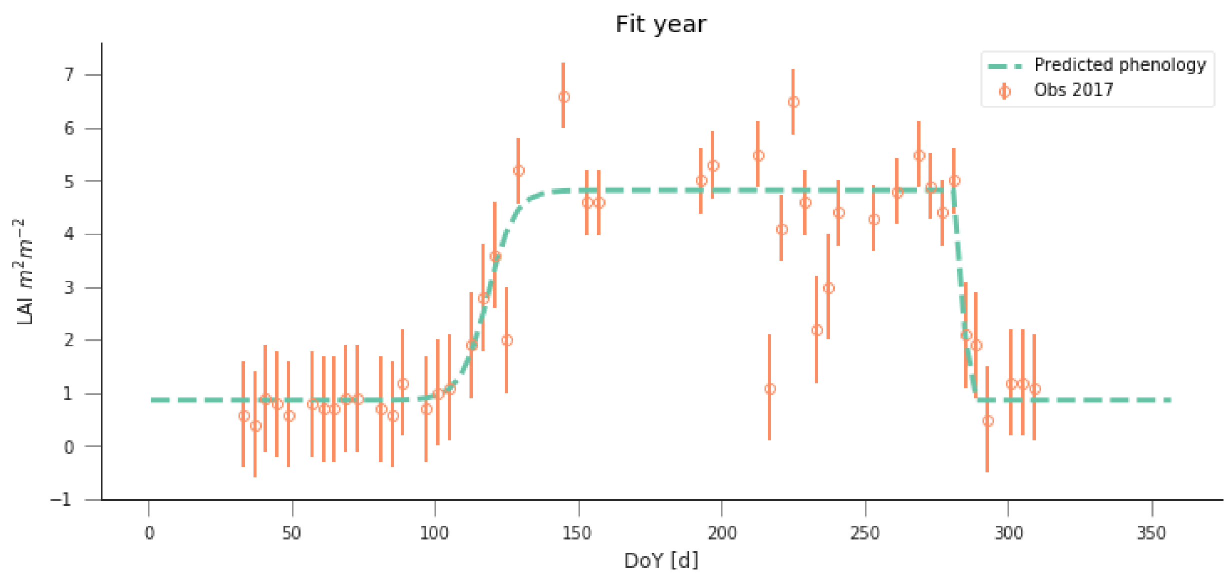


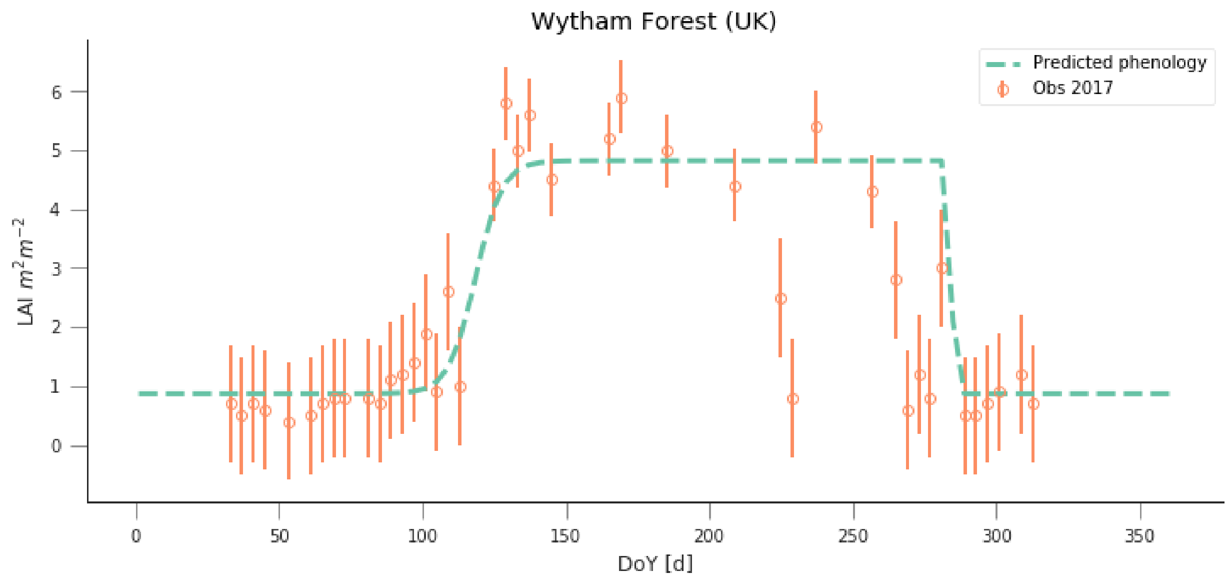
```
y_location, x_location = 51.775511, -1.336993
fit_data_year(2016, 2017, x_location, y_location)
plt.title("Wytham Forest (UK)")
```

```
data/euro_lai/Europe_mosaic_Lai_500m_2016.tif
2201 1974
2201 1974
data/euro_lai/Europe_mosaic_Lai_500m_2017.tif
2201 1974
2201 1974
```

```
Text(0.5,1,'Wytham Forest (UK)')
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```





```
y_location, x_location = 45.8444, 7.5781
fit_data_year(2016, 2017, x_location, y_location)
plt.title("Torgnon (IT)")
```

```
data/euro_lai/Europe_mosaic_Lai_500m_2016.tif
3667 3397
3667 3397
data/euro_lai/Europe_mosaic_Lai_500m_2017.tif
3667 3397
3667 3397
```

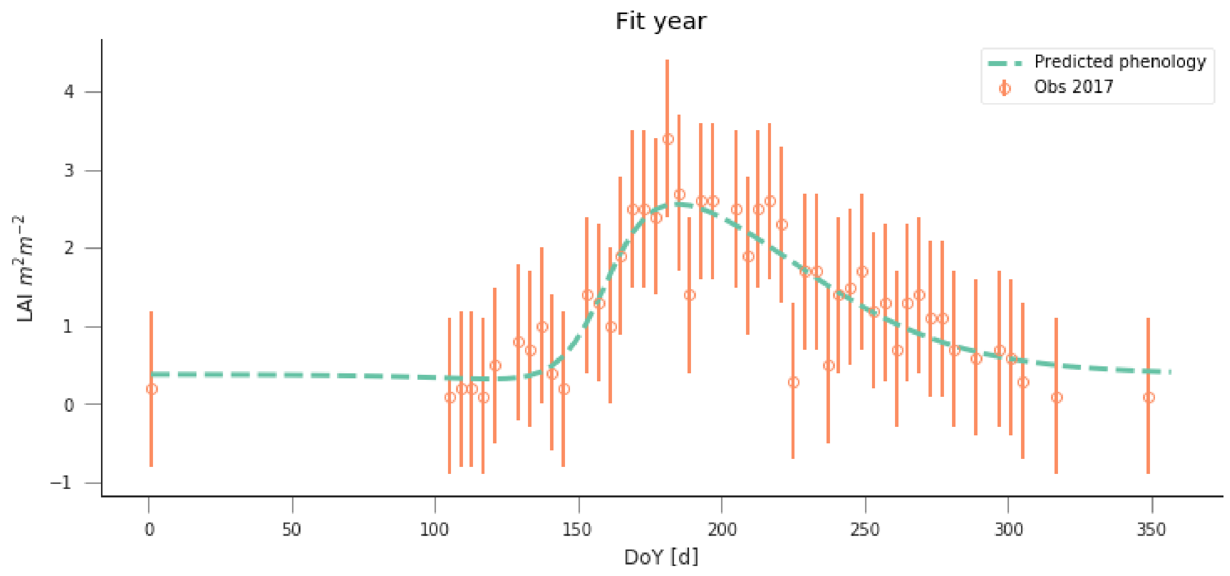
```
Text(0.5,1,'Torgnon (IT)')
```

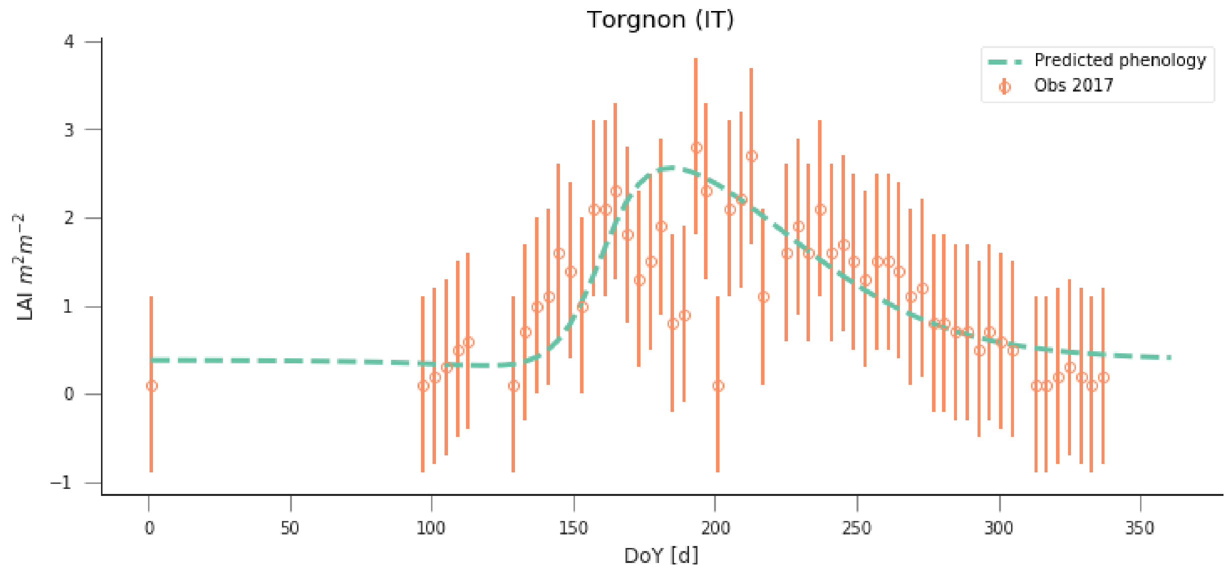
```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
```

(continues on next page)

(continued from previous page)

```
    (prop.get_family(), self.defaultFamily[fontext]))
```



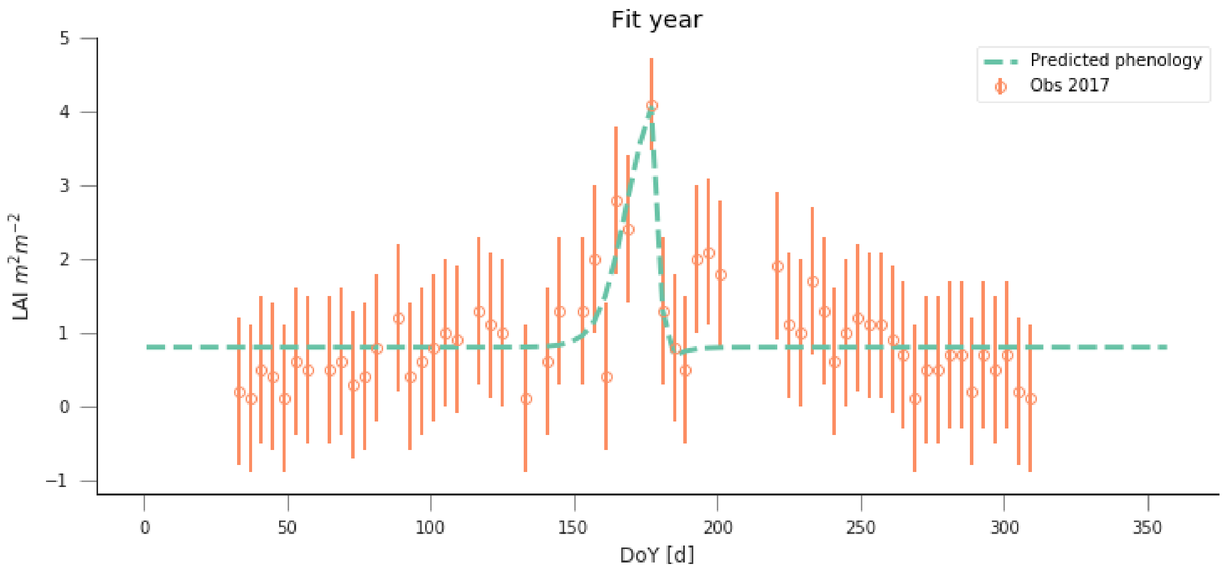


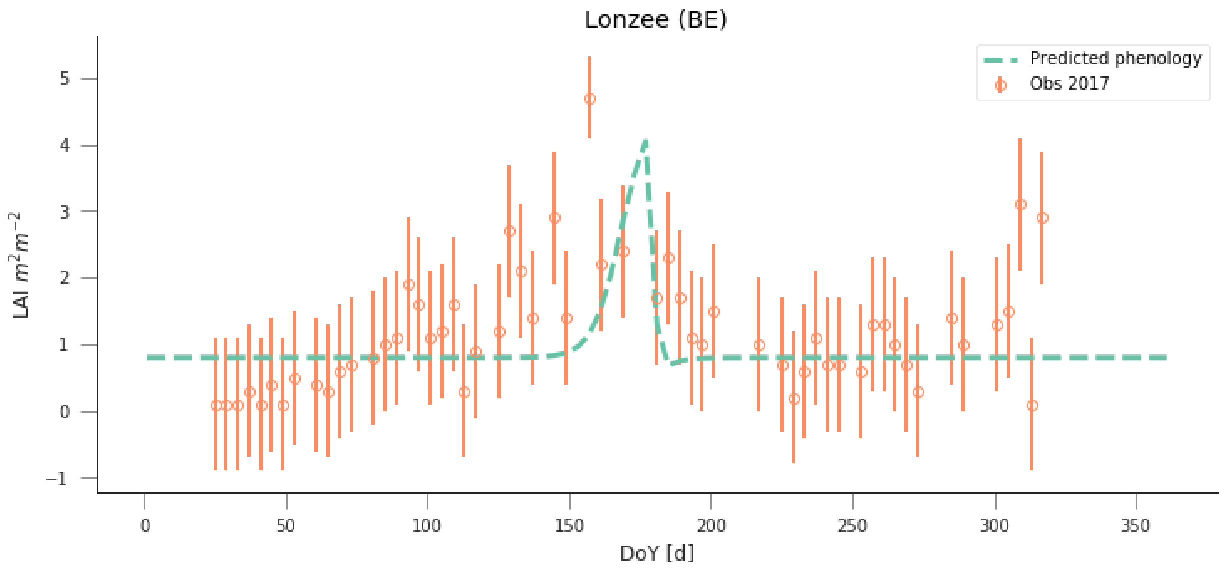
```
y_location, x_location = 50.5516, 4.7461
fit_data_year(2016, 2017, x_location, y_location)
plt.title("Lonzee (BE)")
```

```
data/euro_lai/Europe_mosaic_Lai_500m_2016.tif
3124 2268
3124 2268
data/euro_lai/Europe_mosaic_Lai_500m_2017.tif
3124 2268
3124 2268
```

```
Text(0.5,1,'Lonzee (BE)')
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



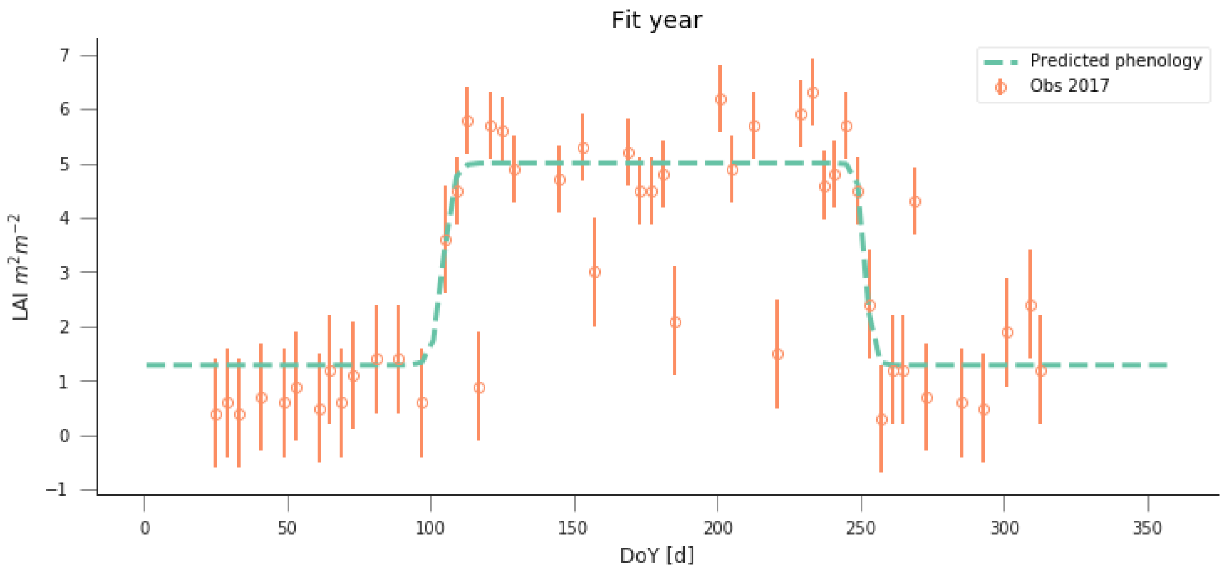


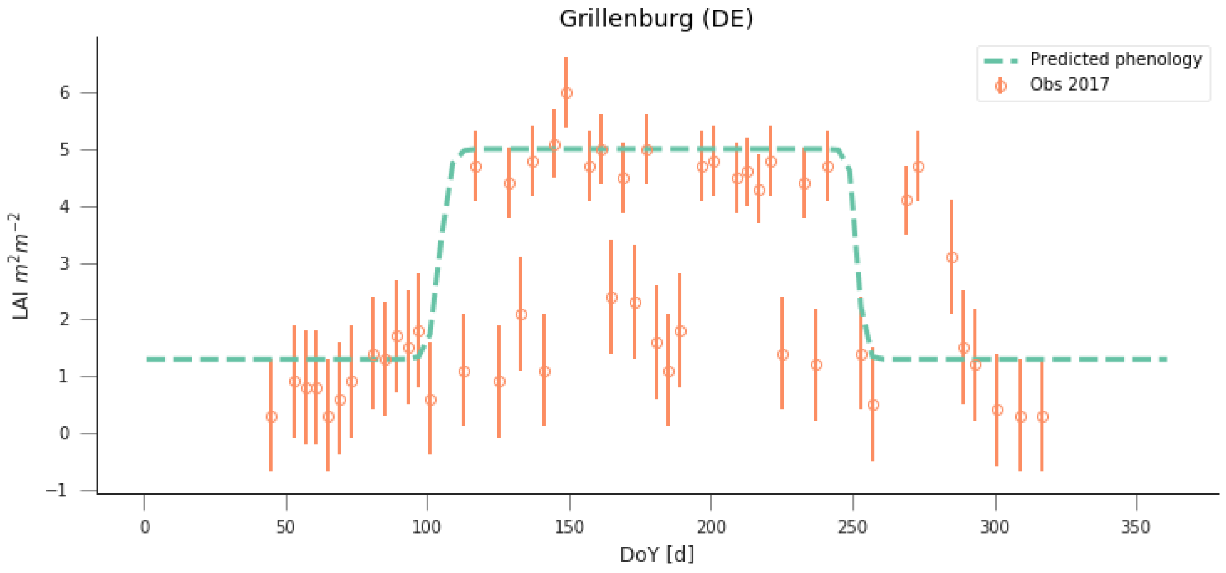
```
y_location, x_location = 50.9500, 13.5126
fit_data_year(2016, 2017, x_location, y_location)
plt.title("Grillenburg (DE)")
```

```
data/euro_lai/Europe_mosaic_Lai_500m_2016.tif
4443 2172
4443 2172
data/euro_lai/Europe_mosaic_Lai_500m_2017.tif
4443 2172
4443 2172
```

```
Text(0.5,1,'Grillenburg (DE)')
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```





```
y_location, x_location = 55.4859,11.6446
fit_data_year(2016, 2017, x_location, y_location)
plt.title("Sorø (DK)")
```

```
data/euro_lai/Europe_mosaic_Lai_500m_2016.tif
3984 1083
3984 1083
data/euro_lai/Europe_mosaic_Lai_500m_2017.tif
```

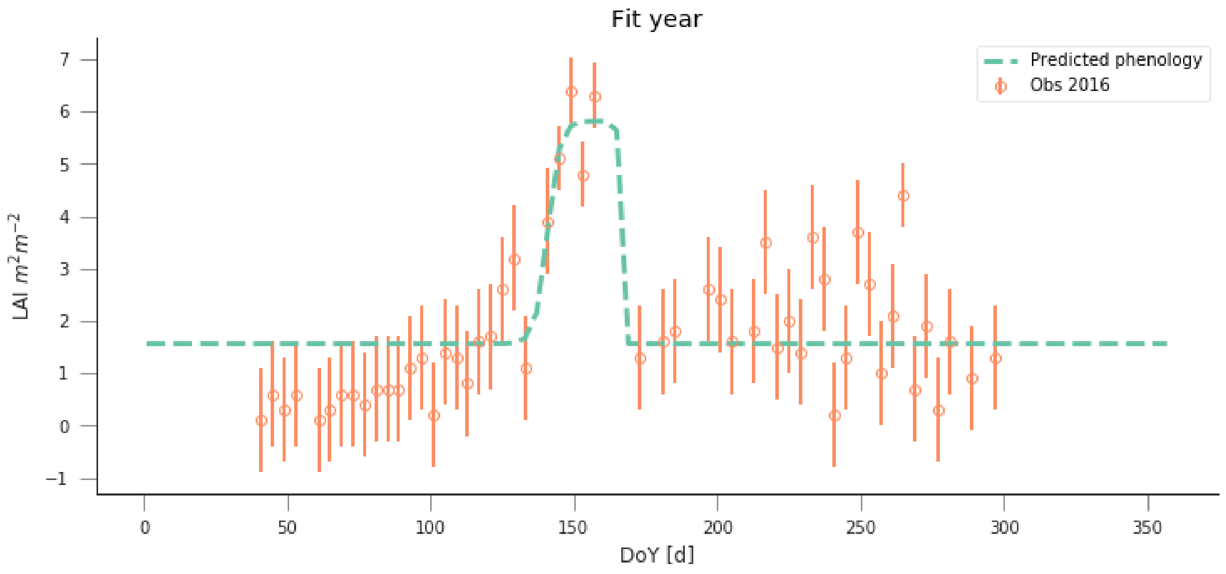
(continues on next page)

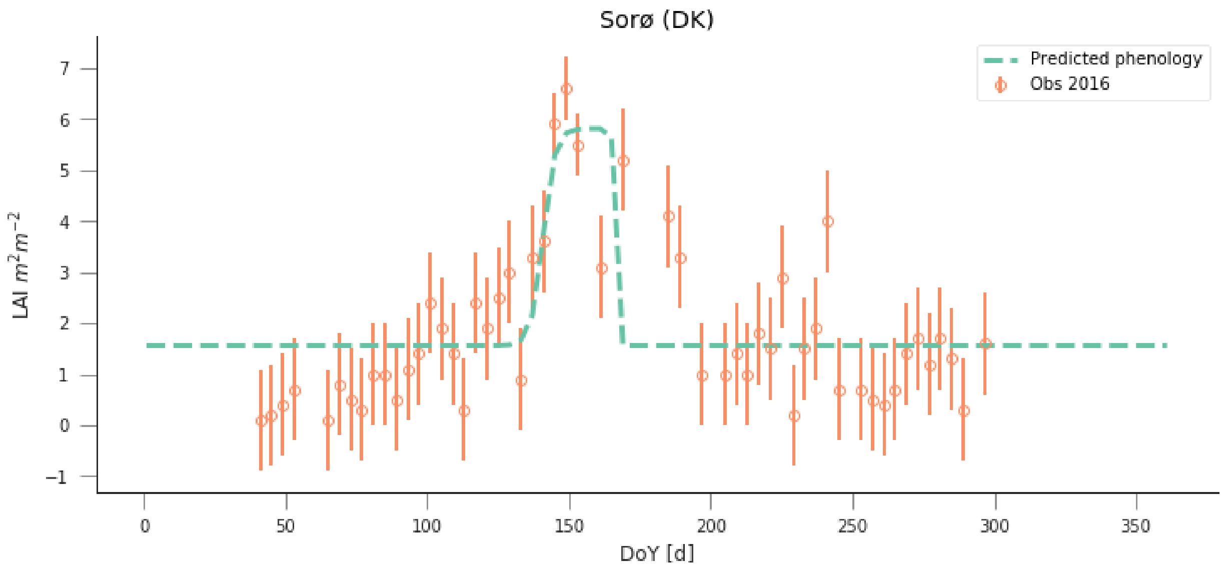
(continued from previous page)

```
3984 1083
3984 1083
```

```
Text(0.5,1,'Sorø (DK)')
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



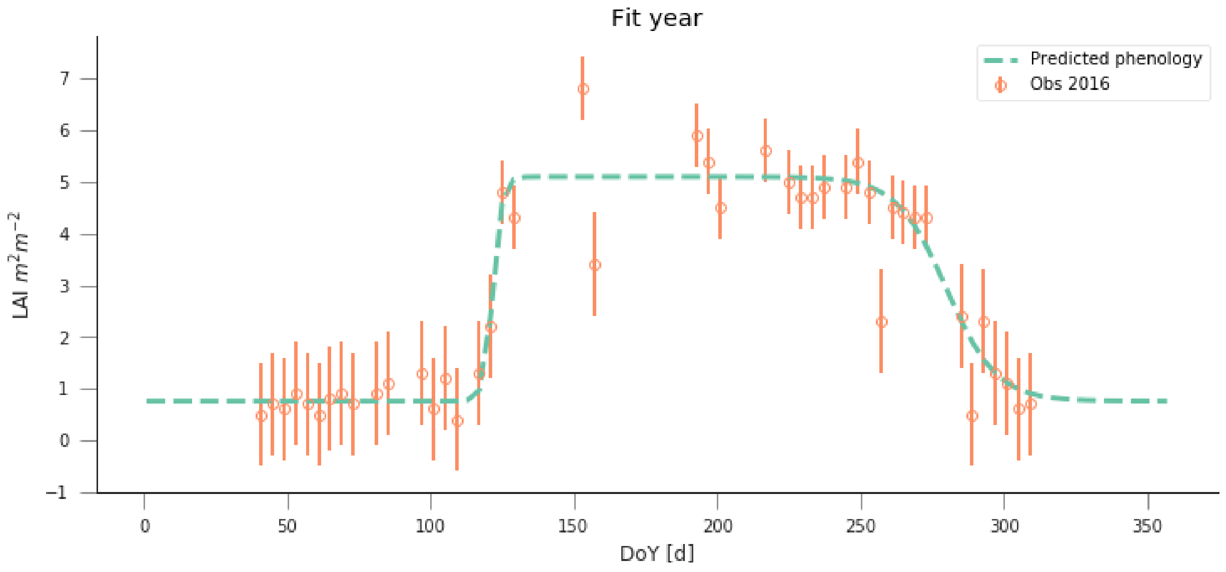


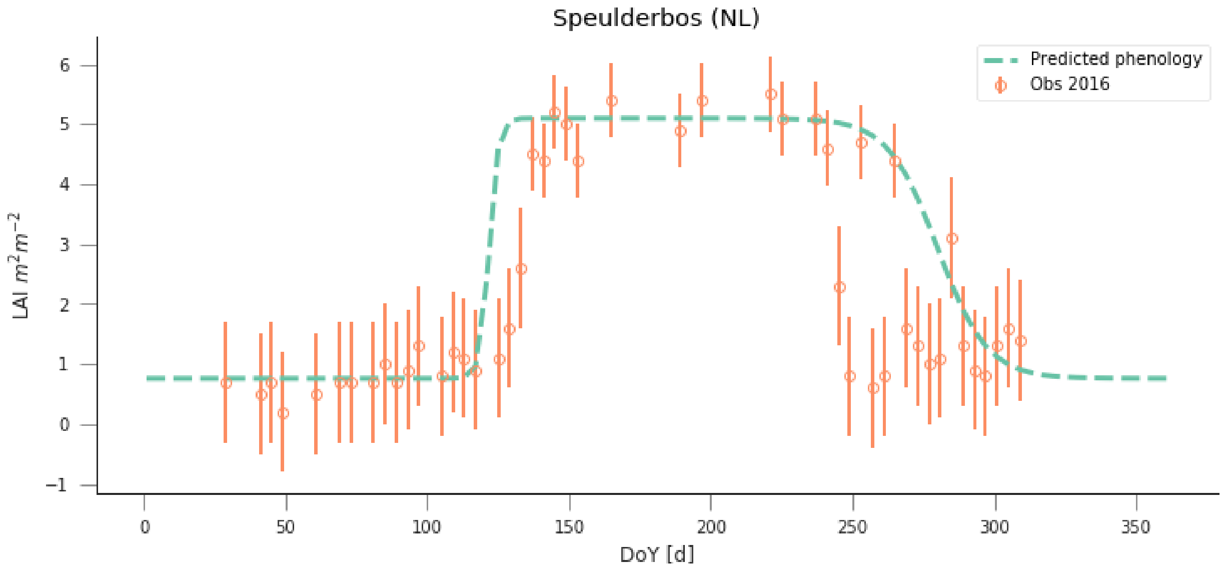
```
y_location, x_location =52.253000000000, 5.702000000000

fit_data_year(2016, 2017, x_location, y_location)
plt.title("Speulderbos (NL)")
```

```
data/euro_lai/Europe_mosaic_Lai_500m_2016.tif
3238 1859
3238 1859
data/euro_lai/Europe_mosaic_Lai_500m_2017.tif
3238 1859
3238 1859
```

```
Text(0.5,1,'Speulderbos (NL)')
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```





Uncertainty

In the previous examples, we have seen that the model can be made to fit observations and used in prediction pretty well, but there's the obvious question of how exactly can we define the different parameters. Ideally, we'd like to have some error bars on e.g. the date and slope of the spring and autumn flanks, so that we can decide whether a shift has ocurred or not. Similarly for the maximum/minimum LAI. Intuitively, we can see that if we change the optimal parameters a bit, we'll get a solution that might still have an acceptable performance (meaning that it will probably go through all the error bars).

One way to test this is to do some Monte Carlo sampling around the optimal solution. We can evaluate the shape of the cost function around the optimum and that will give us an idea of the uncertainty: a cost function that changes very rapidly around the optimal point in one direction suggests that if you change the parameters by a small amount, the goodness of fit changes drastically, so that the parameter is very well and accurately defined. In contrast, if changing the parameter doesn't change the cost function value by much, we have an uncertain parameter.

The Metropolis-Hastings algorithm

A way to do this Monte Carlo sampling is to use the Metropolis-Hastings algorithm. This is a sequential method that proposes and accepts samples based on the likelihood value. Basically, if the cost function improves, the sample gets accepted, if it doesn't improve, then a uniform random number between 0 and 1 is drawn. If the ratio of proposed to previous likelihoods is greater than the random number, the samples gets accepted. This means that for solutions that don't improve the cost function, there's a chance that the algorithm will improve on them, meaning that it doesn't get trapped on local minima, and provides an exploration of the entire problem space.

In a nutshell, here's some **pseudocode** of the MH algorithm

```
1. Initialise $\vec{x}^0$.
2. For $i=1$ to $i=N_{iterations}$:
    a. Sample a proposed new $\vec{x}^*$ as $\vec{x}^* = \vec{x}^{i-1} + \mathcal{N}(0,\Sigma)$.
    b. Calculate the likelihood associated with $\vec{x}^*$, $L(\vec{x}^*, i)$.
    c. Calculate the likelihood ratio $\alpha = \frac{L(\vec{x}^*, i)}{L(\vec{x}^*, i-1)}$.
    d. Draw a random uniform number between 0 and 1 $u = \mathcal{U}(0,1)$
    e. if $u \leq \min\{1, \alpha\}$:
    f. $\vec{x}^{i+1} = \vec{x}^*$: we accept the new proposal
    g. else:
    h. $\vec{x}^{i+1} = \vec{x}^i$: we reject the new proposal
```

Examine the MH algorithm, and try to see whether you can see how you could go and implement it in Python. The pseudocode above presents you with the barebones recipe, and you will need to add a couple of extra ingredients. You can assume that you have available the **log-likelihood** function that we have described above, and you may need to use the `np.random.normal` and `np.random.rand` functions which respectively provide Gaussian random numbers and uniform random numbers between 0 and 1

```python
def metropolis_hastings(xstart, lklhood, year, x_location, y_location,
                        n_iter=50000):
    """MH algorithm implementation. Takes a starting vector, a log-likelihood
    function, a year and an x and y location (in degrees), as well as a number
    of iterations. We are assuming that a `read_data` function that will read
    data and return a time axis, the LAI, the uncertainties and a mask object
```

(continues on next page)

(continued from previous page)

```
    is available. We also assume that the lklhood function can be called using
    these data as `(x_proposed, t, data[passer], passer,sigma)`
    """
    # We first read in the data. this only happens once
    t, data, sigma, passer = read_data(year, x_location, y_location)
    n_params = len(xstart)
    # We reserve some storage for the posterior samples
    samples = np.zeros((n_iter, n_params))
    # We start. Set up xstart as our current parameter vector
    x_curr = xstart*1.
    # Calculate the initial cost function/log-likelihood
    old_cost = lklhood(x_curr, t, data[passer], passer,
                            sigma)
    # Now iterate...
    for i in range(n_iter):
        # Random displacement
        x_proposed = x_curr + np.random.normal(
            size=n_params)*np.array([0.1,
                                    0.1,
                                    0.01,
                                    5.,
                                    0.01,
                                    5])
        # Now, calculate the cost function for the proposed parameter
        # vector...
        proposed_cost = lklhood(x_proposed, t, data[passer], passer,
                            sigma)
        # This is the Metropolis acceptance: if the proposed cost is greater
        # than the previous one, accept, if not sometimes accept
        if np.random.rand() < np.exp(proposed_cost - old_cost):
            # We have accepted the proposed parameter. We update x_curr
            # to match, and updated proposed_cost to the current cost
            x_curr = x_proposed
            old_cost = proposed_cost
        # if we don't accept (because the proposed cost is less, and the
        # difference is below the random draw) we keep the old cost, and
        # do not update x_curr
        # We store the samples nevertheless!!
        samples[i, :] = x_proposed

    return samples
```

```
y_location, x_location = 43.0156, -6.7038

def read_data(year, x_location, y_location):
    lai_raster = f"data/euro_lai/Europe_mosaic_Lai_500m_{year:d}.tif"
    qa_raster = f"data/euro_lai/Europe_mosaic_FparLai_QC_{year:d}.tif"
    print(lai_raster)
    data = read_tseries(lai_raster, y_location, x_location)/10.
    qa = read_tseries(qa_raster, y_location, x_location)
    passer = get_sfc_qc(qa) <= 1
    sigma = get_scaling(get_sfc_qc(qa))[passer]

    t = np.arange(len(passer))*4 + 1
    return t, data, sigma, passer
```

(continues on next page)

(continued from previous page)

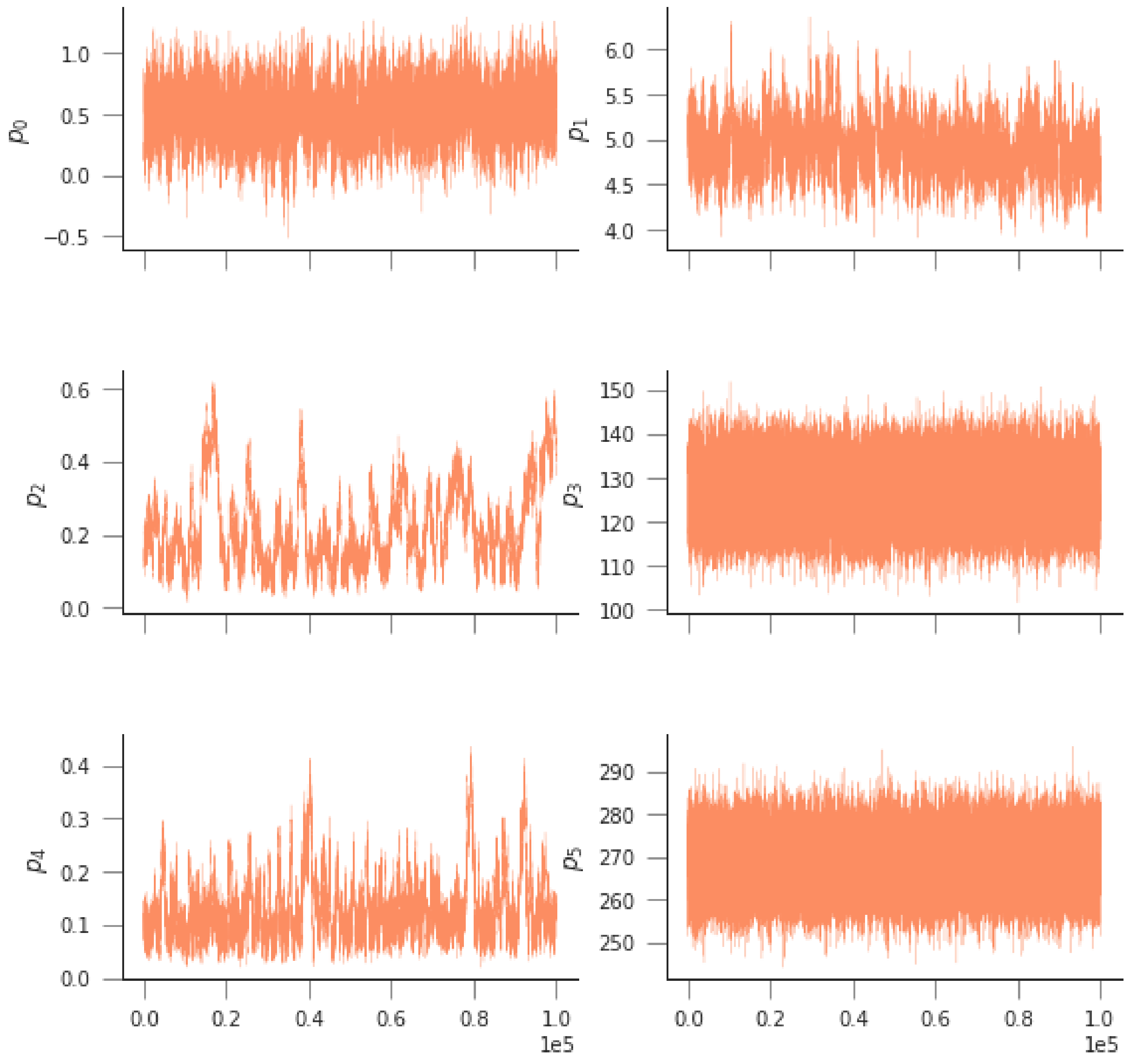
```
def lklhood(p, t, y_obs, passer, sigma_obs, func=dbl_sigmoid_function):
    y_pred = func(p, t)
    n = passer.sum()
    cost = -0.5* (y_pred[passer]-y_obs)**2/sigma_obs**2
    return cost.sum()




samples = metropolis_hastings(retval2016.x, lklhood,
                             2016, x_location, y_location,
                            n_iter=100000)
```

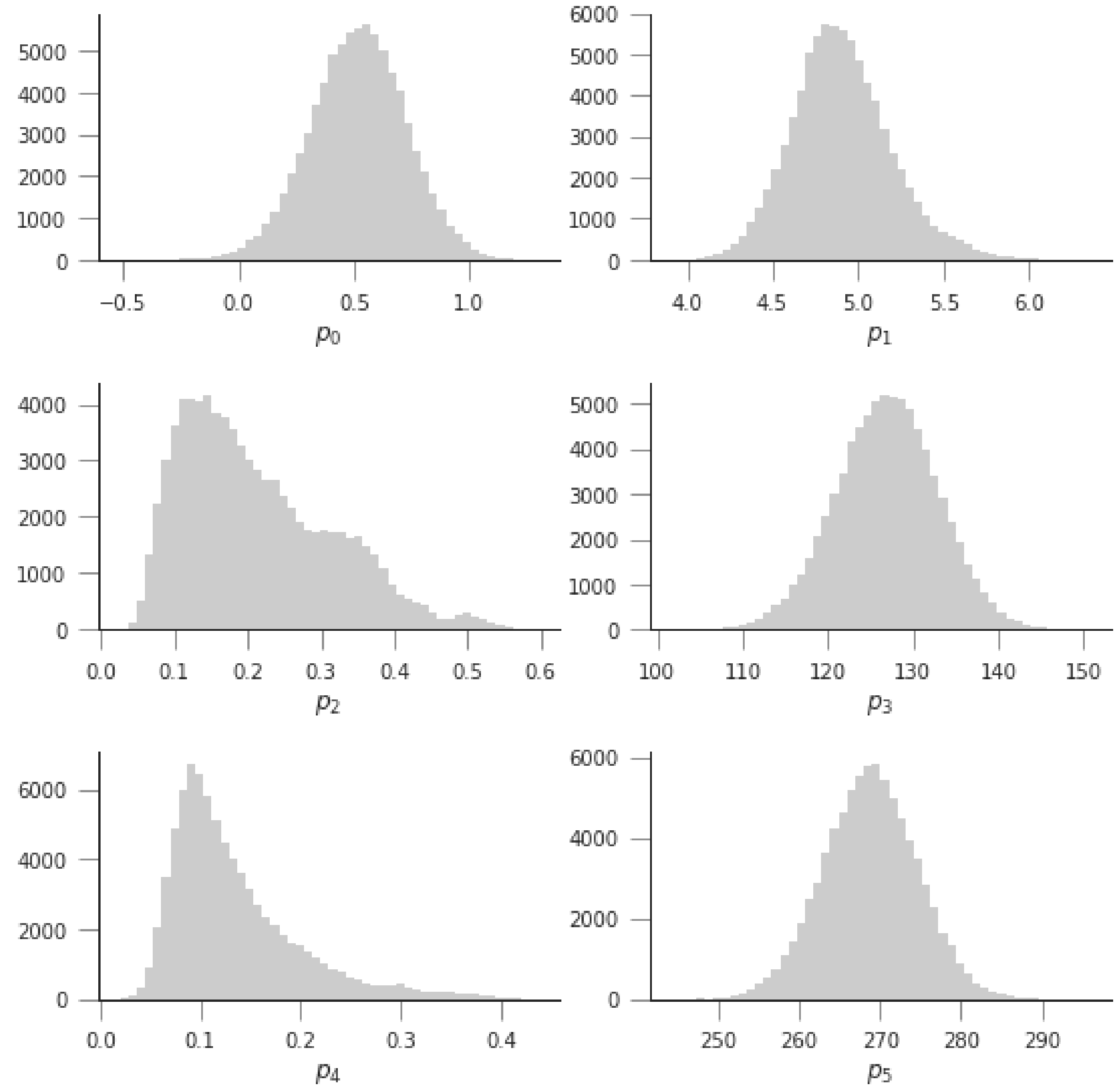
```
data/euro_lai/Europe_mosaic_Lai_500m_2016.tif
1224 4076
1224 4076
```

The previous code has produced samples of parameters that we can now visualise as "traces" as well as histograms. The shape of the histograms gives us some idea of the uncertainty of the parameters in their units. We can also use these samples and propagate them through the phenology model to produce an *ensemble* of model trajectories that define a region of uncertainty.

```
fig, axs = plt.subplots(nrows=3, ncols=2, figsize=(9, 9), sharex=True)
axs = axs.flatten()
for i in range(len(retval2016.x)):
    axs[i].plot(samples[:, i], '-', lw=0.2)
    axs[i].set_ylabel(f"$p_{i}$")

fig, axs = plt.subplots(nrows=3, ncols=2, figsize=(9, 9))
axs = axs.flatten()
for i in range(len(retval2016.x)):
    axs[i].hist(samples[20000:, i], bins=50, color="0.8")
    axs[i].set_xlabel(f"$p_{i}$")
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```


p_0
p_1
p_2
p_3
p_4
p_5
1e5
1e5



```
n_samples = samples.shape[0]
y_location, x_location = 43.0156, -6.7038

t, data, sigma, passer = read_data(2016, x_location, y_location)

pred_lai = np.zeros((n_samples, len(t)))
for i in range(n_samples):
    pred_lai[i, :] = dbl_sigmoid_function(samples[i], t)


plt.figure(figsize=(15, 7))
pcntiles = np.percentile( pred_lai[60000::20], [5, 25, 50, 75, 95], axis=0)
plt.fill_between(t, pcntiles[0], pcntiles[-1], color="0.9")
plt.fill_between(t, pcntiles[1], pcntiles[-2], color="0.7")
plt.plot(t, pcntiles[2], '--', lw=3, label="Median")
```

(continues on next page)

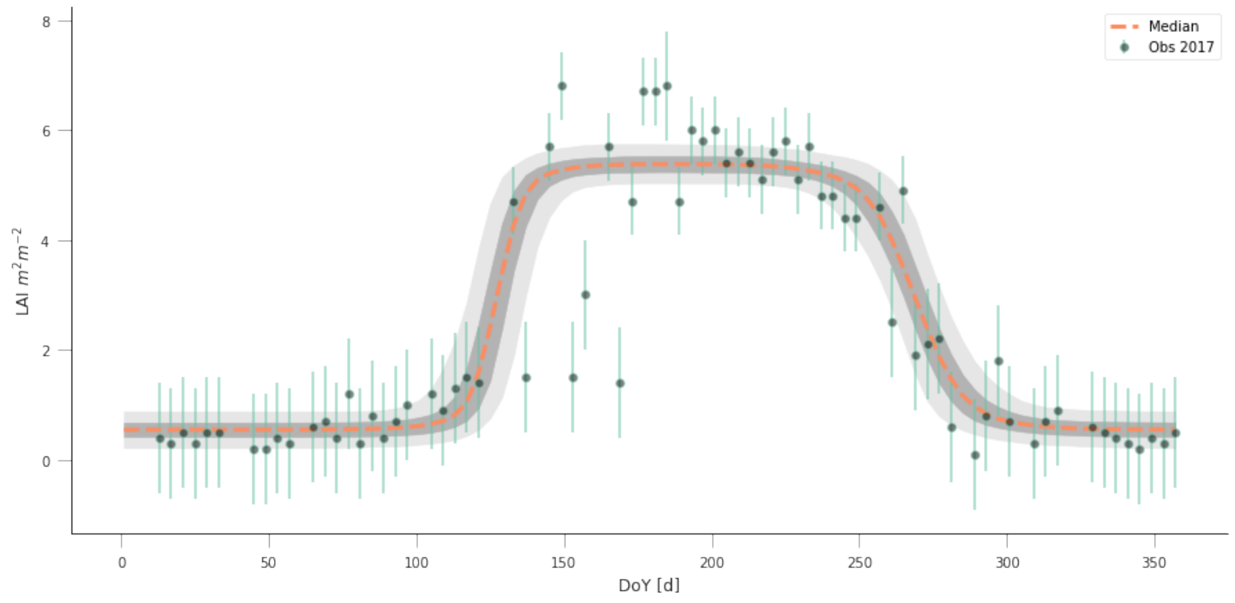
(continued from previous page)

```
plt.errorbar(t[passer], data[passer], yerr=sigma, fmt="o",
             mfc="none", label=f"Obs {year:d}", alpha=0.5)

plt.legend(loc="best")
plt.ylabel("LAI $m^{2}m^{-2}$")
plt.xlabel("DoY [d]")
```

```
data/euro_lai/Europe_mosaic_Lai_500m_2016.tif
1224 4076
1224 4076
```

```
Text(0.5,0,'DoY [d]')
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



```
cov = np.corrcoef(samples[60000::20, :].T)
```

```
def hinton(matrix, max_weight=None, ax=None):
    """Draw Hinton diagram for visualizing a weight matrix."""
    ax = ax if ax is not None else plt.gca()

    if not max_weight:
        max_weight = 2 ** np.ceil(np.log(np.abs(matrix).max()) / np.log(2))

    ax.patch.set_facecolor('gray')
    ax.set_aspect('equal', 'box')
    ax.xaxis.set_major_locator(plt.NullLocator())
    ax.yaxis.set_major_locator(plt.NullLocator())
```

(continues on next page)

(continued from previous page)

```
    for (x, y), w in np.ndenumerate(matrix):
        color = 'white' if w > 0 else 'black'
        size = np.sqrt(np.abs(w) / max_weight)
        rect = plt.Rectangle([x - size / 2, y - size / 2], size, size,
                             facecolor=color, edgecolor=color)
        ax.add_patch(rect)

    ax.autoscale_view()
    ax.invert_yaxis()

plt.figure(figsize=(12,12))
hinton(cov)
```

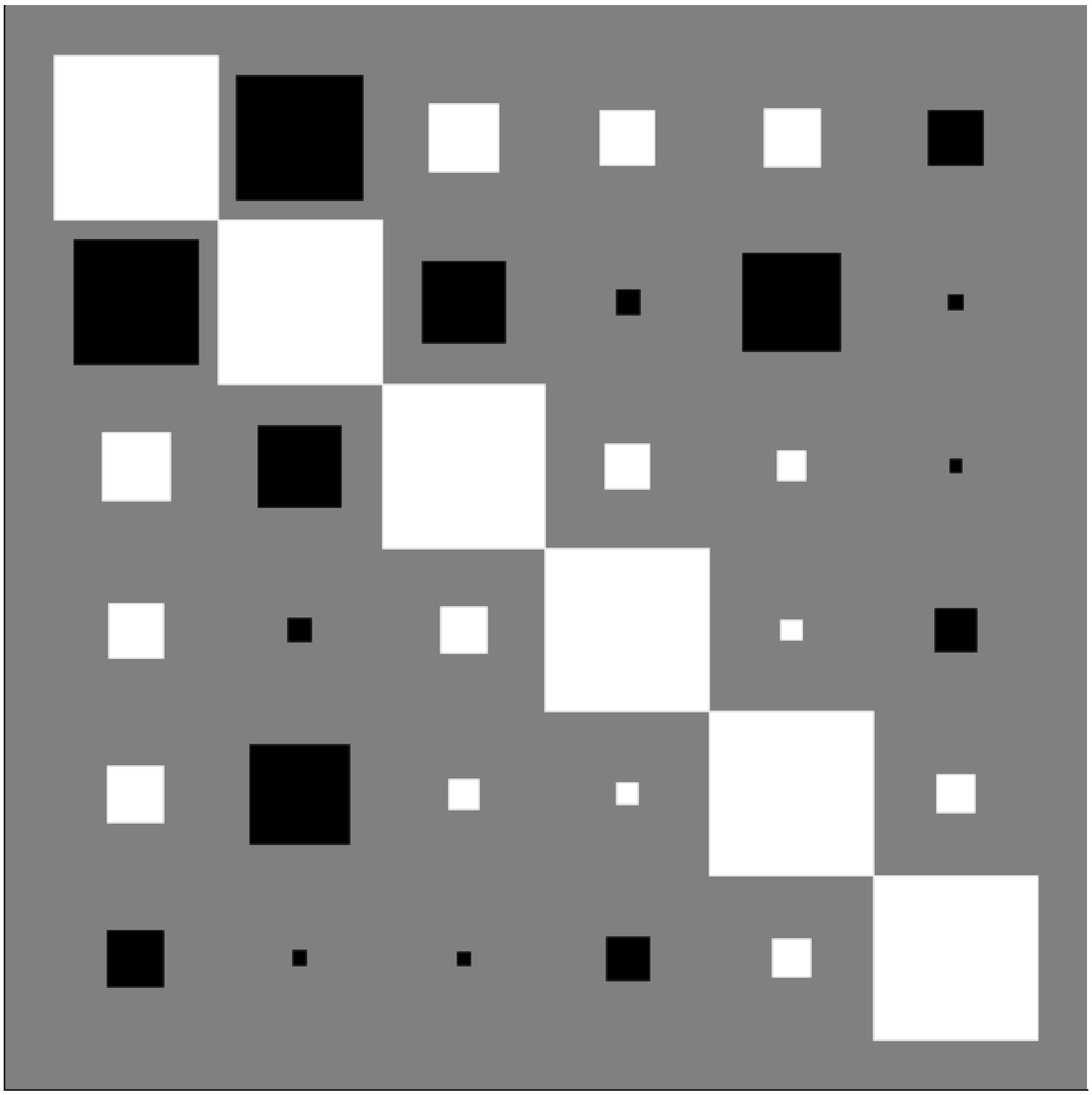


Table of Contents

2.8.27 8. Assessed Practical

8.1 Introduction

8.1.1 Task overview

These notes describe the practical work you must submit for assessment in this course.

The practical comes in two parts: (1) data preparation (50%); (2) modelling (50%).

It is important that you complete both parts of this exercise.

The submission for Part 1 of the coursework (worth 50% of the marks) is the Monday after Reading week (12:00 Noon). That is Monday 12^{th} November 2018.

The submission for Part 2 will be 7^{th} January 2019 (12:00 Noon). Submission is through the usual Turnitin link on the course Moodle page.

- **Part 1: Data Preparation**

  The first task you must complete is to produce a dataset of the proportion of HUC catchment 13010001 (Rio Grande headwaters in Colorado, USA) that is covered by snow for **two years** (not necessarily consecutive), along with associated datasets on temperature (in C) and river discharge at the Del Norte monitoring station. You will use these data in the modelling work in Part 2 of the coursework.

  You **may not** use data from the years 2005 or 2006, as this will be given to you in illustrations of the material.

  The dataset you produce must have a value for **the mean snow cover, temperature and discharge in the catchment for every day over each year.**

  Your write up **must** include fully labelled graph(s) of snow cover, temperature and discharge for the catchment for each year (with units as appropriate), along with some summary statistics (e.g. mean or median, minimum, maximum, and the timing of these).

  You **must** provide evidence of how you got these data (i.e. the code and commands you ran to produce the data).

Checklist:

```
* provide fully commented/documented code for all operations.
* provide two years of **daily** data (not 2005 or 2006)
* Generate datasets of:
    * mean snow cover (0.0 to 1.0) for the catchment for each day of the year
    * temperature (C) at the Del Norte monitoring station for each day of the year
    * river discharge at the Del Norte monitoring station for each day of the year
* Produce a table of summary statistics for each of the 3 datasets (one for easch␣
↪year)
* produce graphs of the 3 datasets for each year (as function of day of year)
* produce an `npz` file containing the 3 datasets, one for each year.
* produce images of snow cover spatial data for the catchment for **13 samples**␣
↪spaced equally through the year, one set of images for each year. You need to do␣
↪this for the data pre-interpolation and aftyer you have done the interpolation.
```

- **Part 2: Modelling**

  You will have prepared two years of data in Part 1 of the work.

  If, for some reason, you have failed to generate an appropriate dataset, you may use datasets that will be provided for you for the years 2005 and 2006. There will be no penalty for that in your Part 2 submission: failure to gernerate the datasets will be accounted for in marks allocated for Part 1.

  You will be given a simple hydrological model of snowmelt.

Use one of these years to calibrate the (snowmelt) hydrological model and one year to test it.

The model parameter estimate *must* be objective (i.e. you can't just arbitrarily choose a set) and optimal in some way you must define (you *must* state the equation of the cost function you will try to minimise and explain the approach used).

You **must** state the values of the model parameters that you have estimated and show evidence for how you went about calculating them. Ideally, you should also state the uncertainty in these parameter estimates (not critical to pass this section though).

You **must** quantify the goodness of fit between your measured flow data and that produced by your model, both for the calibration exercise and the validation.

Checklist:

```
* Provide a site intoduction and an introduction to the purpose of the exercise (
↪'Introduction')
* Provide an introduction to the modelling and calibration/validation ('Method')
* provide code that reads in the datasets and performs the model calibration and␣
↪validation ('Code')
* Provide a table of results on model parameter calibration (and ideally,␣
↪uncertainty) ('Results')
* Provide graphs of the observed and modelled river discharge data for the␣
↪calibration year ('Results')
* Provide graphs of the observed and modelled river discharge data for the validation␣
↪year ('Results')
* Assess the accuracy of the calibration and validation ('Results')
* Discuss the results in the light of the introduction ('Discussion')
* Draw conclusions about issues associated with modelling of this sort ('Conclusion')
```

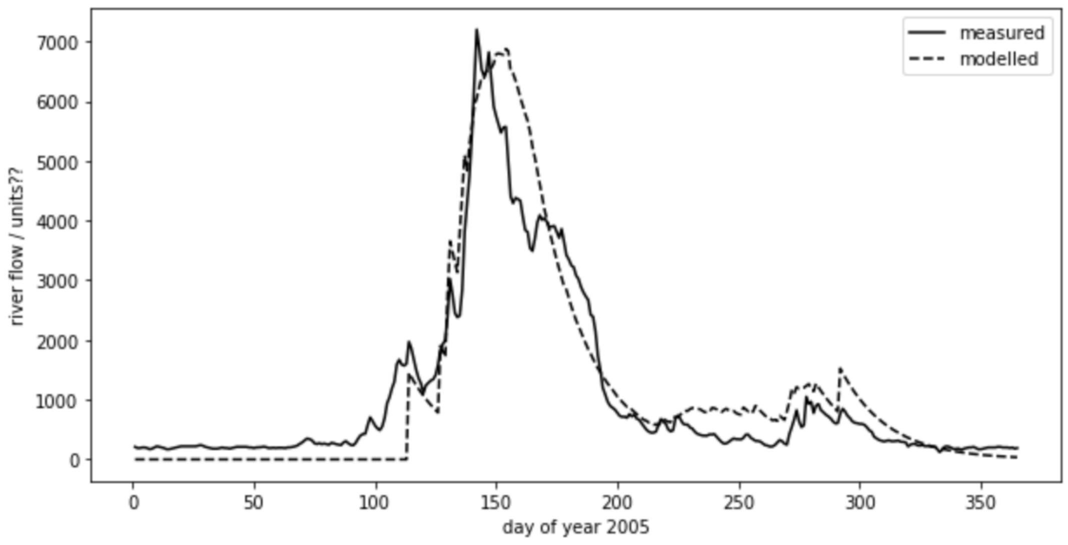
You **must** work individually on this task. If you do not, it will be treated as plagiarism. By reading these instructions for this exercise, we assume that you are aware of the UCL rules on plagiarism. You can find more information on this matter in your student handbook. If in doubt about what might constitute plagiarism, ask one of the course convenors.

8.1.2 Purpose of the work

The hydrology of the Rio Grande Headwaters in Colorado, USA is snowmelt dominated. It varies considerably from year to year and may very further under a changing climate.

We can build a mathemetical ('environmental') model to describe the main physical processes affecting hydrology in the catchment. Such a model could help understand current behaviour and allow some prediction about possible future scenarios.

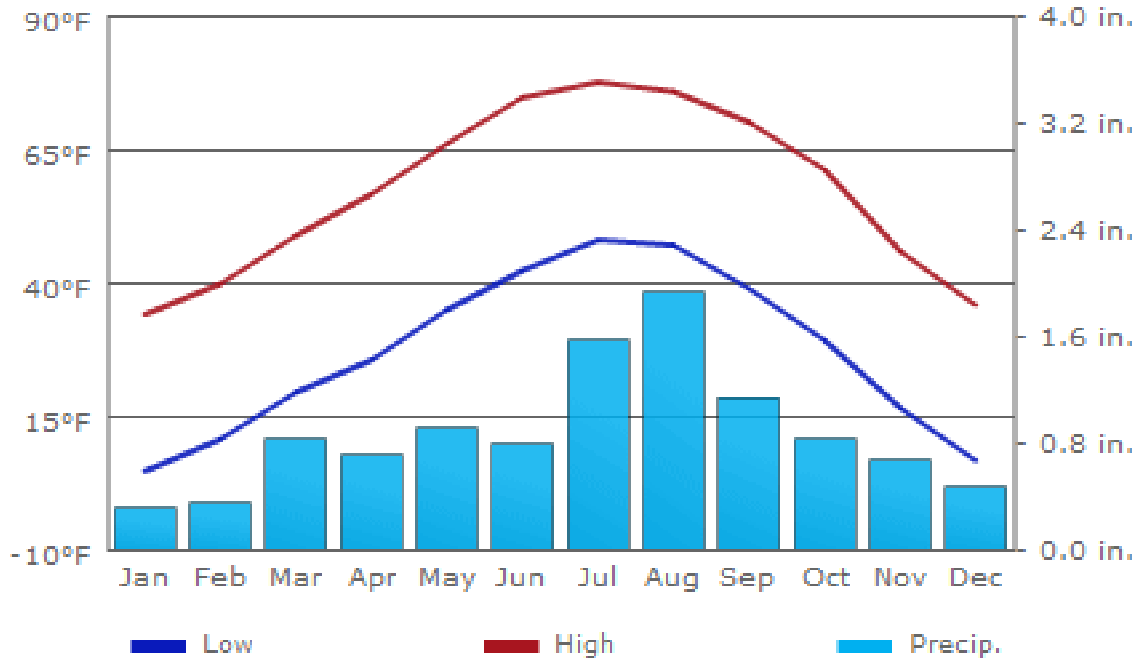
What you are going to do is to build, calibrate and test a (snowmelt) hydrological model, driven by observations in the Rio Grande Headwaters in Colorado, USA



The purpose of the model will be to describe the streamflow at the Del Norte measurement station, just on the edge of the catchment. You will use environmental (temperature) data and snow cover observations to drive the model. You will perform calibration and testing by comparing model output with observed streamflow data.

8.1.2.1 Del Norte

The average climate for Del Norte is:



Further general information is available from various websites, including NOAA.

Fig. 24: www.coloradofishing.net

You can visualise the site Del Norte 2E here.

8.1.2.2 Previous work

In part 1, you should have developed datasets for two years for:

```
* mean snow cover (0.0 to 1.0) for the catchment for each day of the year
* temperature (C) at the Del Norte monitoring station for each day of the year
* river discharge at the Del Norte monitoring station for each day of the year
```

You should use these datasets in part 2 of this work.

If for some reason, you are unable to do that, you may use the datasets provided here:

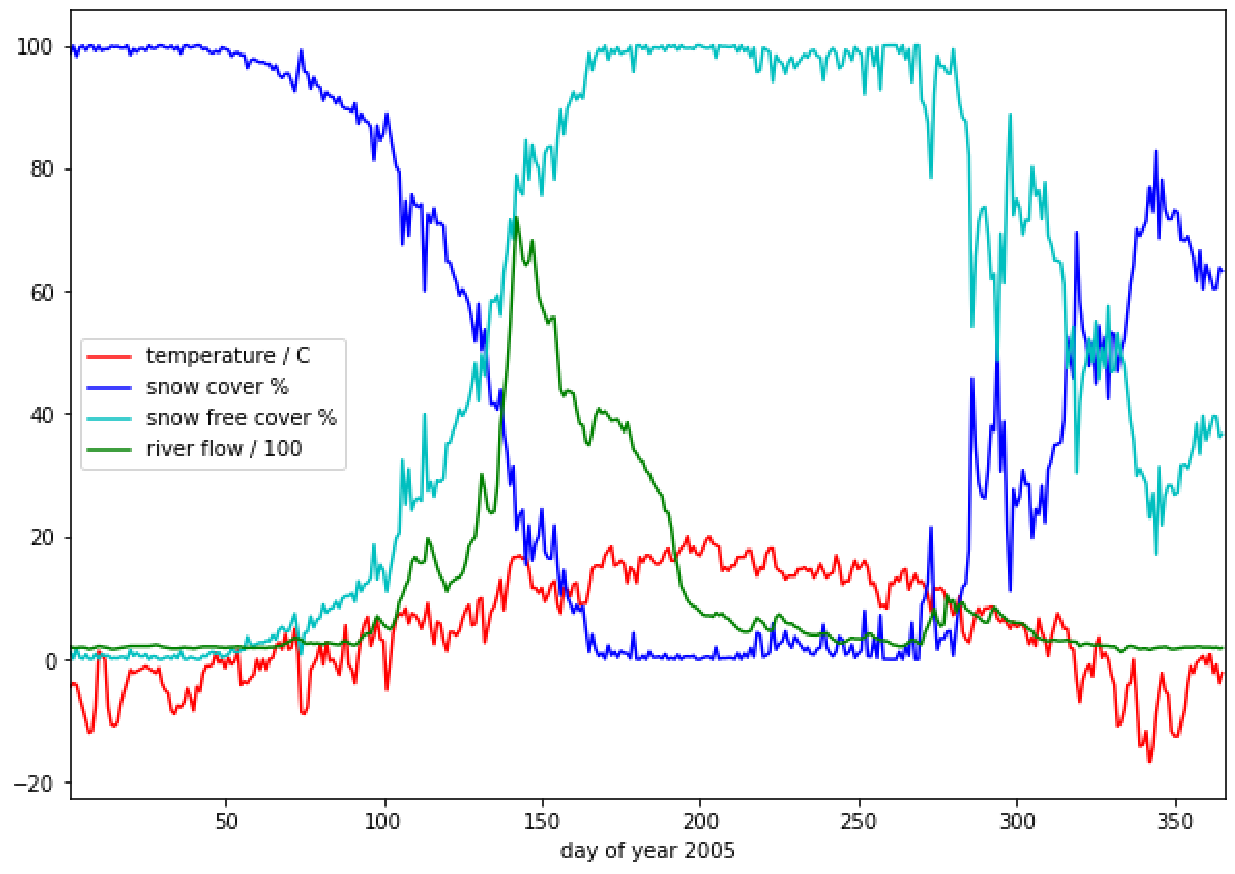
```
# load a pre-cooked version of the data for 2005 (NB -- Dont use this year!!!
# except perhaps for testing)
```

(continues on next page)

(continued from previous page)

```
# load the data from a pickle file
import pickle
import pylab as plt
import numpy as np
%matplotlib inline

pkl_file = open('data/data2005.pkl', 'rb')
# note encoding='latin1' because pickle generated in python2
data = pickle.load(pkl_file, encoding='latin1')
pkl_file.close()

# set up plot
plt.figure(figsize=(10,7))
plt.xlim(data['doy'][0],data['doy'][-1]+1)
plt.xlabel('day of year 2005')

# plot data
plt.plot(data['doy'],data['temp'],'r',label='temperature / C')
plt.plot(data['doy'],data['snowprop']*100,'b',label='snow cover %')
plt.plot(data['doy'],100-data['snowprop']*100,'c',label='snow free cover %')
plt.plot(data['doy'],data['flow']/100.,'g',label='river flow / 100')
plt.legend(loc='best')
```

```
<matplotlib.legend.Legend at 0x10e269be0>
```



we have plotted the streamflow (scaled) in green, the snow cover in blue, and the non snow cover in cyan and the temperature in red. It should be apparent that thge hydrology is snow melt dominated, and to describe this (i.e. to build the simplest possible model) we can probably just apply some time lag function to the snow cover.

8.2 The model

What we need is a model that provides an estimate of flow F (`data['flow']`) as a function of time t (days).

The information we have to help in this is : snow proportion p (`data['snowprop']`) and temperature T (`data['temp']`)

8.2.1 Snow water equivalent

In a snow-melt dominated catchment, we are assuming that the flow can be related to water released from the snow pack. The purpose of any model then is to model that release. The units of this (flow) will be units of volume per unit time.

The water contained in the snow pack is called the Snow Water Equivalent (SWE) and is a measure of water volume. To estimate this, we would need to know the snow equivalent depth d and the snow area. The snow area is the catchment area multipled by p, so:

$$SWE = Apd$$

where A is the catchment area.

We have no information on depth, so will assume that it varies in the same way as snow cover. Letting:

$$d = \frac{k}{A}p$$

Then:

$$SWE = kp^2$$

8.2.2 Snow water melt

From examination of the data plotted above, we can see that the timing of the river discharge corresponds broadly to:

- the presence of snow cover ($p > 0$)
- temperature above some threshold ($T > T_{thresh}$)

where T_{thresh} is the threshold temperature.

This is hardly surprising and exactly what we would expect in a snowmelt-dominated catchment. We might reasonably expect T_{thresh} to be $0C$, but this might not be the case (why?).

A simple model of the SWE entering the system then is to make it equal to some proportion k_p of the SWE on days when melting occurs.

Then:

$$SWE_{melt}(t) = k_p(t)SWE(t)$$

where SWE_{melt} is the proportion of SWE released per unit time (day).

One function we could use for k_p is to make it proportionate to the excess temperature.

We can define this as:

$$k_p(T) = \frac{T - T_{thresh}}{T_{max}}$$

with negative k_p set to zero, and T_{max} the maximum temperature. So, if $T = T_{thresh} + T_{max}$. $k_p = 1$ and the maximum amount of SWE is available as melt water.

8.2.3 Base flow

Close examination of the data suggests there is a base level flow (i.e. a constant flow throughout the year) of around 200 units. We can simply add this to our model as F_{base}.

$$F_{non-base}(t) = F(t) - F_{base}$$

F_{base} is easily estimated from the January mean value of flow.

8.2.4 Total amount of water

We then have modelled flow F_{model} entering the system:

$$F_{model}(t) = F_{base} + SWE_{melt}(t)$$

$$F_{model}(t) = F_{base} + kMAX\left(0, \frac{T - T_{thresh}}{T_{max}} p(t)^2\right)$$

Further, we can assume that the total amount of flow that we model (minus the base flow) should equal that measured (summed over all days).

Then:

$$\Sigma_t F_{model}(t) = \Sigma_t F(t)$$

so

$$\Sigma_t F(t) = \Sigma_t F_{base} + k\Sigma_t MAX\left(0, \frac{T - T_{thresh}}{T_{max}} p(t)^2\right)$$

This allows us to infer the value of k from the data:

$$k = \frac{\Sigma_t (F(t) - F_{base})}{\Sigma_t MAX\left(0, \frac{T - T_{thresh}}{T_{max}} p(t)^2\right)}$$

```
# Guess some values
Tthresh = 6

# we can estimate the base flow from the
# mean January value
F_base = data['flow'][:31].mean()
print(f'F_base = {F_base}')


#
p           = data['snowprop']
k_p         = (data['temp'] - Tthresh)/data['temp'].max()
```

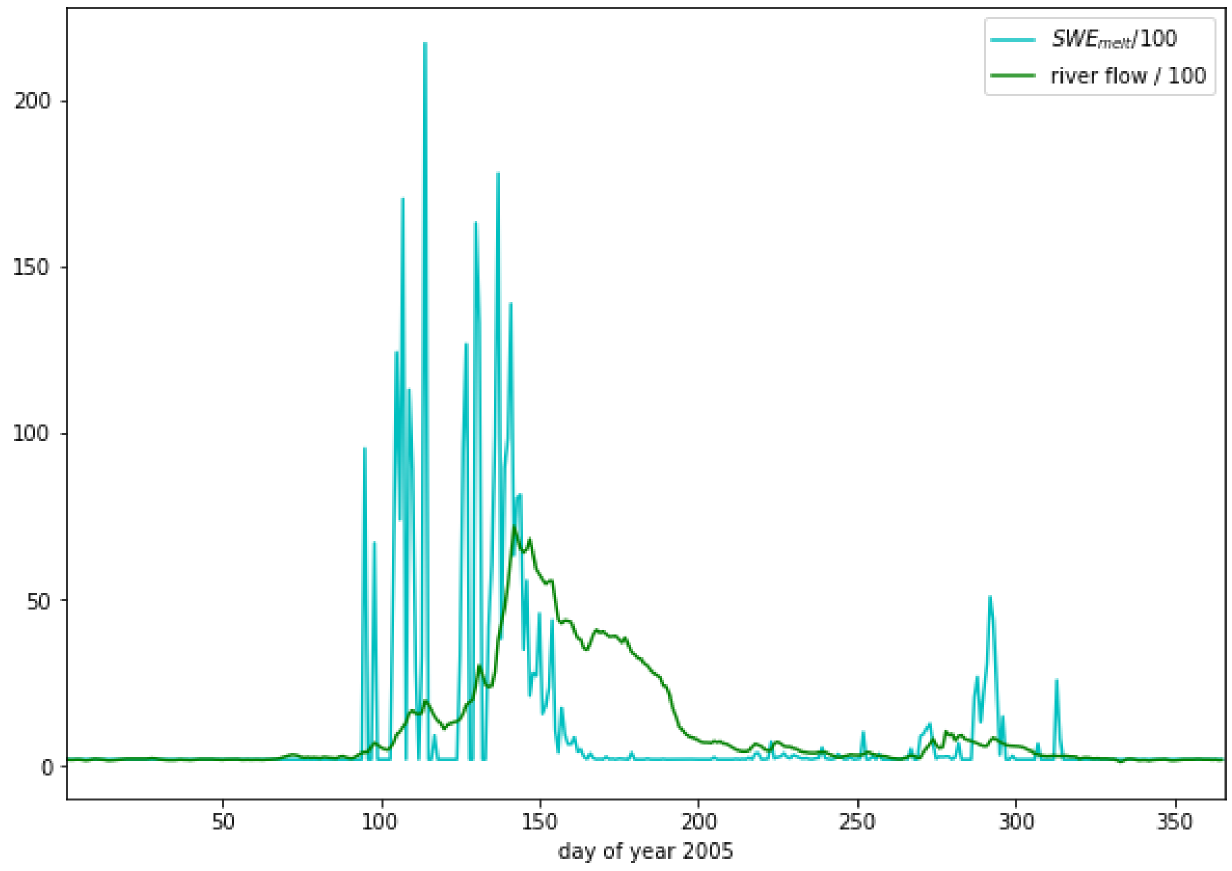
(continues on next page)

(continued from previous page)

```
k_p          = np.max([np.zeros_like(k_p),k_p],axis=0)
swe_melt_k   = p * p * k_p

# take away the base flow from flow
nonbase_flow = data['flow'] - F_base

# k
k = nonbase_flow.sum() / swe_melt_k.sum()

model_flow = swe_melt_k * k + F_base

# set up plot
plt.figure(figsize=(10,7))
plt.xlim(data['doy'][0],data['doy'][-1]+1)
plt.xlabel('day of year 2005')

# plot data
plt.plot(data['doy'],model_flow/100,'c',label='$SWE_{melt}$/100')
plt.plot(data['doy'],data['flow']/100.,'g',label='river flow / 100')
plt.legend(loc='best')
```

```
F_base = 203.8709677419355
```

```
<matplotlib.legend.Legend at 0xb212ef908>
```



This already looks quite reasonable, although:

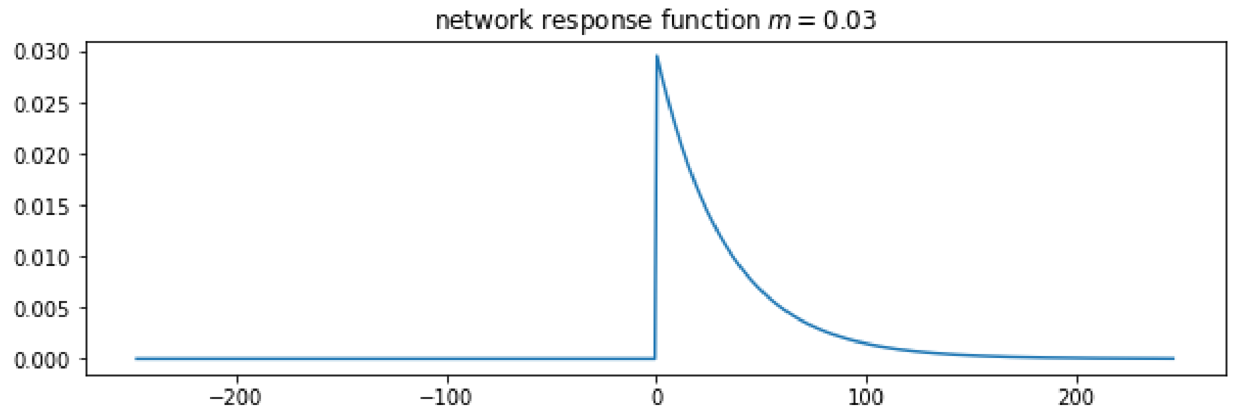
- the flow is smoother than the SWE_{melt} data
- there seems to be a delay between snowmelt occuring and flow appearing in the measurements

The function that describes such delay (and that can cause smoothing) can be called a network response function. It is often modelled as a Laplace function (an exponential). The idea is that for any 'flash' input to the catchment, this network response function will give us what we would measure as a hydrograph at the monitoring station (or elsewhere):

$$nrf = e^{-mt}$$

```
# decay parameter
# nrf = exp(-m x)
m  = 0.03

plt.figure(figsize=(10,3))
# window size
ndays = 15 * int(1/m)
nrf_x = np.arange(ndays) - ndays/2
# function for nrf
nrf = np.exp(-m*nrf_x)
nrf[nrf_x<0] = 0

# normalise so that sum is 1
nrf = nrf/nrf.sum()

# plot
plt.plot(nrf_x,nrf)
plt.title(f'network response function $m={m}$')
```

```
Text(0.5, 1.0, 'network response function $m=0.03$')
```



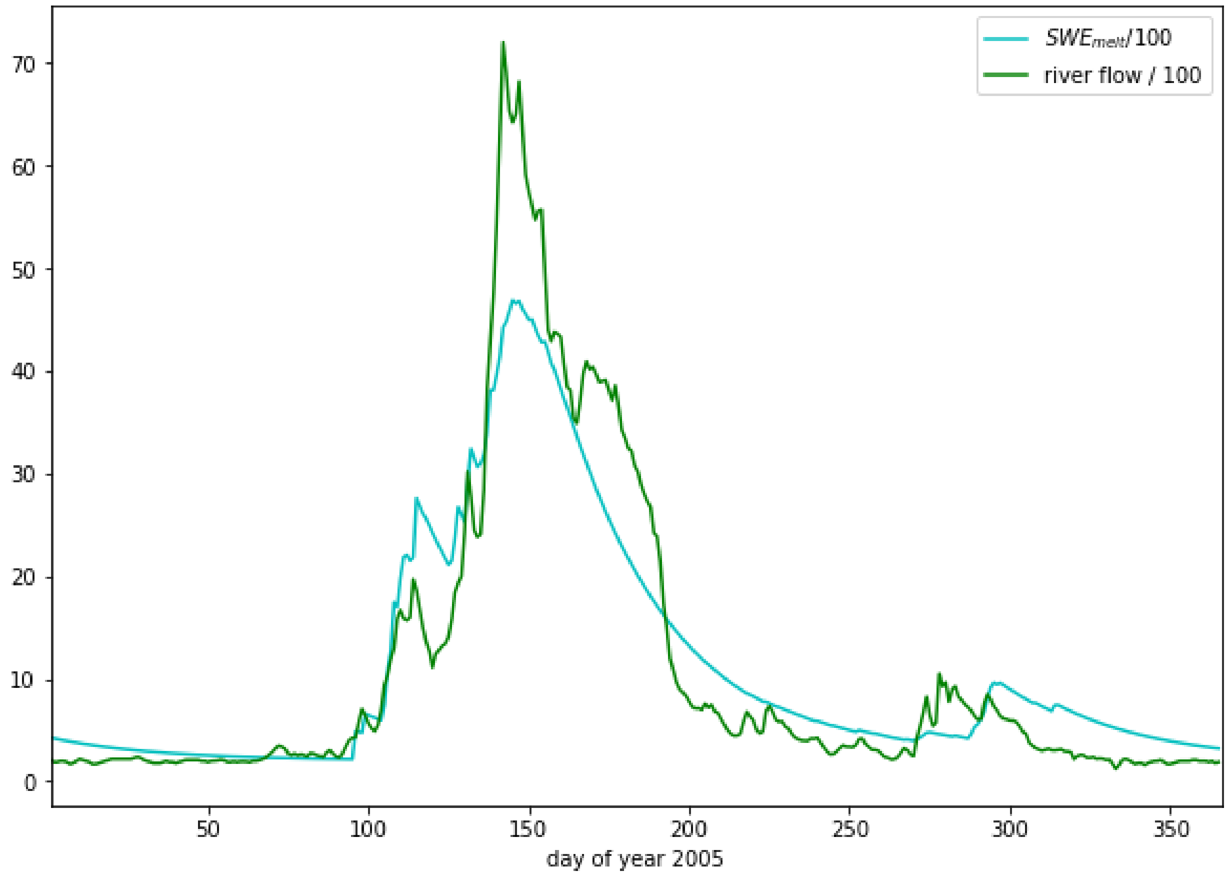
```
import scipy
import scipy.ndimage.filters

# convolve NRF with data
model_flow_nrf = scipy.ndimage.filters.convolve1d(model_flow, nrf)

# set up plot
plt.figure(figsize=(10,7))
```

(continues on next page)

(continued from previous page)

```
plt.xlim(data['doy'][0],data['doy'][-1]+1)
plt.xlabel('day of year 2005')

# plot data
plt.plot(data['doy'],model_flow_nrf/100,'c',label='$SWE_{melt}$/100')
plt.plot(data['doy'],data['flow']/100.,'g',label='river flow / 100')
plt.legend(loc='best')
```

```
<matplotlib.legend.Legend at 0xb215a4358>
```



So, we have defined a simple snow melt model. It has two parameters that we3 can vary:

- T_{thresh}: the lower temperature threshold
- m: the network response function decay factor

The modelled flow is normalised so that the total flow equals the total measured flow. The base flow is estimated from the data (January mean flow value).

Note that the values of the additional model parameters, T_{max}, F_{base} and the scaling term k should be constant for all years modelled. i.e. ytou work out these values from one year of data, then apply them to the modelling of other years.

A quick guess at the parameter values provides a not unreasonable match to the data. As a model that predicts behaviour well for the main spring melt events, this may be quite useful. That said, the model is very simple and might easily be improved.

If we accept that this is a sufficient model for the present, we can attempt to improve on the model performance by optimising the (two) parameters.

To do this, we can use standard optimisation techniques (that you have already covered in the course) to provide imprtoved parameter estimates. There are only two model parameters to optimise, so this should be quite straightforward. It should also be easy to visualise the error function.

Exercise

Write a Python function that takes the `data` dictionary and the model parameters and returns the modelled flow for given T_{thresh} and m.

Vary the two parameters a bit and note the impact on the match between modelled and observed flow data.

```
# Exercise
```

8.3 Coursework

You need to submit you coursework in the usual manner by the usual submission date.

You **must** work individually on this task. If you do not, it will be treated as plagiarism. By reading these instructions for this exercise, we assume that you are aware of the UCL rules on plagiarism. You can find more information on this matter in your student handbook. If in doubt about what might constitute plagiarism, ask one of the course convenors.

8.3.1 Summary of coursework requirements

- **Part 2: Modelling**

  ```
  You will have prepared two years of data in Part 1 of the work.
  ```

  If, for some reason, you have failed to generate an appropriate dataset, you may use datasets that will be provided for you for the years 2005 and 2006. There will be no penalty for that in your Part 2 submission: failure to gernerate the datasets will be accounted for in marks allocated for Part 1.

  You have been given a simple (two parameter) hydrological model of snowmelt.

  Use one of these years to calibrate the (snowmelt) hydrological model and one year to test (validate) it.

  The model parameter estimate *must* be objective (i.e. you can't just arbitrarily choose a set) and optimal in some way you must define (you *must* state the equation of the cost function you will try to minimise and explain the approach used). You should state why you have taken any particular approach for the optimisation, and make consideration of what options you might have.

  You **must** state the values of the model parameters that you start with and those youy have estimated. ~TYou must show evidence for how you went about calculating the optimised pareameters. Ideally, you should also state the uncertainty in these parameter estimates (not critical to pass this section though). You should try to at least make some comment on the likely uncertainty.

  You **must** quantify the goodness of fit between your measured flow data and that produced by your model, both for the calibration exercise and the validation.

Checklist:

```
* Provide a site intoduction and an introduction to the purpose of the exercise (
↪'Introduction')
* Provide an introduction to the modelling and calibration/validation ('Method')
* provide code that reads in the datasets and performs the model calibration and␣
↪validation ('Code')
* Provide a table of results on model parameter calibration (and ideally,␣
↪uncertainty) ('Results')
```

(continues on next page)

(continued from previous page)

```
* Provide graphs of the observed and modelled river discharge data for the␣
↪calibration year ('Results')
* Provide graphs of the observed and modelled river discharge data for the validation␣
↪year ('Results')
* Assess the accuracy of the calibration and validation ('Results')
* Discuss the results in the light of the introduction ('Discussion')
* Draw conclusions about issues associated with modelling of this sort ('Conclusion')
```

8.3.2 Summary of Advice

The second task revolves around using the model that we have developed above. You should probably put the elements of the model together in a function. You have been through previous examples in Python where you attempt to estimate some model parameters given an initial estimate of the parameters and some cost function to be minimised. Solving the model calibration part of problem should follow those same lines then. Testing (validation) should be easy enough. Don't forget to include the estimated parameters (and other relevant information, e.g. your initial estimate, uncertainties if available) in your write up.

There is quite a lot of data presentation here, and you need to provide *evidence* that you have done the task. Make sure you use graphs (e.g. one of the inoputs used for calibration and validation years, and one for modelled and predicted flow, etc.), and tables (e.g. model parameter estimates) throughout, as appropriate.

8.3.3 Further advice

There is plenty of scope here for going beyond the basic requirements (e.g. improving the model), if you get time and are interested (and/or want a higher mark!).

You will be given credit for all additional work included in the write up, **once you have achieved the basic requirements**. So, there is no point (i.e. you will not get credit for) going off on all sorts of interesting lines of exploration here *unless* you have first completed the core task.

Be aware that if you decide to develop a new model, and this has more parameters than the existing model, the fact that your model fit is better than that for the excisting model *does not* imply that your model is an impriovement. This is because you will have increased the degrees of freedom in the model, so an improved fit is not longer adequate proof of model improvement.

8|.3.4 Structure of the Report

The required elements of the report are:

```
1. Provide a site intoduction and an introduction to the purpose of the exercise (
↪'Introduction') [5]
2. Provide a context and introduction to the modelling and calibration/validation and␣
↪a visualisation of the input data you use ('Method') [10]
3. provide code that reads in the datasets and performs the model calibration and␣
↪validation ('Code') [10]
4. A Results section with appropriate graphs and tables [10]
5. Discuss the results in the light of the introduction ('Discussion') [10]
6. Draw conclusions about issues associated with modelling of this sort ('Conclusion
↪') [5]
```

The figures in brackets indicate the percentage of marks that we will award for each section of the report.

8.3.5 Computer Code

General requirements

You will obviously need to submit computer codes as part of this assessment. Some flexibility in the style of these codes is to be expected. For example, some might write a class that encompasses the functionality for all tasks. Some poeple might have multiple versions of codes with different functionality. All of these, and other reasonable variations are allowed.

All codes needed to demonstrate that you have performed the core tasks are required to be included in the submission. You should include all codes that you make use of in the main body of the text in the main body. Any other codes that you want to refer to (e.g. something you tried out as an enhancement and didn't quite get there) you can include in appendices.

All codes should be well-commented. Part of the marks you get for code will depend on the adequacy of the commenting.

Degree of original work required and plagiarism

If you use a piece of code verbatim that you have taken from the course pages or any other source, **you must acknowledge this** in comments in your text. **Not to do so is plagiarism**. Where you have taken some part (e.g. a few lines) of someone else's code, **you should also indicate this**. If some of your code is heavily based on code from elsewhere, **you must also indicate that**.

Some examples.

The first example is guilty of strong plagiarism, it does not seek to acknowledge the source of this code, even though it is just a direct copy, pasted into a method called `model()`:

```python
def model(tempThresh=9.0,K=2000.0,p=0.96):
    '''need to comment this further ...

    '''
    import numpy as np
    meltDays = np.where(temperature > tempThresh)[0]
    accum = snowProportion*0.
    for d in meltDays:
        water = K * snowProportion[d]
        n = np.arange(len(snowProportion)) - d
        m = p ** n
        m[np.where(n<0)]=0
        accum += m * water
    return accum
```

This is **not** acceptable.

This should probably be something along the lines of:

```python
def model(tempThresh=9.0,K=2000.0,p=0.96):
    '''need to comment this further ...

    This code is taken directly from
    "Modelling delay in a hydrological network"
    by P. Lewis http://www2.geog.ucl.ac.uk/~plewis/geogg122/DelNorte.html
    and wrapped into a method.
    '''
```

(continues on next page)

(continued from previous page)

```
    # my code: make sure numpy is imported
    import numpy as np

    # code below verbatim from Lewis
    meltDays = np.where(temperature > tempThresh)[0]
    accum = snowProportion*0.
    for d in meltDays:
        water = K * snowProportion[d]
        n = np.arange(len(snowProportion)) - d
        m = p ** n
        m[np.where(n<0)]=0
        accum += m * water
    # my code: return accumulator
    return accum
```

Now, we acknowledge that this is in essence a direct copy of someone else's code, and clearly state this. We do also show that we have added some new lines to the code, and that we have wrapped this into a method.

In the next example, we have seen that the way m is generated is in fact rather inefficient, and have re-structured the code. It is partially developed from the original code, and acknowledges this:

```
def model(tempThresh=9.0,K=2000.0,p=0.96):
    '''need to comment this further ...

    This code after the model developed in
    "Modelling delay in a hydrological network"
    by P. Lewis
    http://www2.geog.ucl.ac.uk/~plewis/geogg122/DelNorte.html

    My modifications have been to make the filtering more efficient.
    '''
    # my code: make sure numpy is imported
    import numpy as np

    # code below verbatim from Lewis unless otherwise indicated
    meltDays = np.where(temperature > tempThresh)[0]
    accum = snowProportion*0.

    # my code: pull the filter block out of the loop
    n = np.arange(len(snowProportion))
    m = p ** n

    for d in meltDays:
        water = K * snowProportion[d]

        # my code: shift the filter on by one day
        # ...do something clever to shift it on by one day

        accum += m * water
    # my code: return accumulator
    return accum
```

This example makes it clear that significant modifications have been made to the code structure (and probably to its efficiency) although the basic model and looping comes from an existing piece of code. It clearly highlights what the actual modifications have been. Note that this is not a working example!!

Although you are supposed to do this piece of work on your own, there might be some circumstances under which someone has significantly helped you to develop the code (e.g. written the main part of it for you & you've just copied that with some minor modifications). You **must** acknowledge in your code comments if this has happened. On the whole though, this should not occur, as you **must** complete this work on your own.

If you take a piece of code from somewhere else and all you do is change the variable names and/or other cosmetic changes, you **must** acknowledge the source of the original code (with a URL if available).

Plagiarism in coding is a tricky issue. One reason for that is that often the best way to learn something like this is to find an example that someone else has written and adapt that to your purposes. Equally, if someone has written some tool/library to do what you want to do, it would generally not be worthwhile for you to write your own but to concentrate on using that to achieve something new. Even in general code writing (i.e. when not submitting it as part of your assessment) you and anyone else who ever has to read your code would find it of value to make reference to where you found the material to base what you did on. The key issue to bear in mind in this work, as it is submitted 'as your own work' is that, to avoid being accused of plagiarism and to allow a fair assessment of what you have done, you must clearly acknowledge which parts of it are your own, and the degree to which you could claim them to be your own.

For example, based on . . . is absolutely fine, and you would certainly be given credit for what you have done. In many circumstances 'taken verbatim from . . . ' would also be fine (provided it is acknowledged) but then you would be given credit for what you had done with the code that you had taken from elsewhere (e.g. you find some elegant way of doing the graphs that someone has written and you make use of it for presenting your results).

The difference between what you submit here and the code you might write if this were not a piece submitted for assessment is that you the vast majority of the credit you will gain for the code will be based on the degree to which you demonstrate that you can write code to achieve the required tasks. There would obviously be some credit for taking codes from the coursenotes and bolting them together into something that achieves the overall aim: provided that worked, and you had commented it adequately and acknowledge what the extent of your efforts had been, you should be able to achieve a pass in that component of the work. If there was no original input other than vbolting pieces of existing code together though, you be unlikely to achieve more than a pass. If you get less than a pass in another component of the coursework, that then puts you in danger of an overall fail.

Provided you achieve the core tasks, the more original work that you do/show (that is of good quality), the higher the mark you will get. Once you have achieved the core tasks, even if you try something and don't quite achieve it, is is probably worth including, as you may get marks for what you have done (or that fact that it was a good or interesting thing to try to do).

Documentation

Note: All methods/functions and classes must be documented for the code to be adequate. Generally, this will contain:

- some text on the purpose of the method (/function/class)
- some text describing the inputs and outputs, including reference to any relevant details such as datatype, shape etc where such things are of relevance to understanding the code.
- some text on keywords, e.g.:

```python
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)

    Example taken verbatim from:
    http://www.python.org/dev/peps/pep-0257/
    """
    if imag == 0.0 and real == 0.0: return complex_zero
```

You should look at the document on good docstring conventions when considering how to document methods, classes etc.

To demonstrate your documentation, you **must** include the help text generated by your code after you include the code. e.g.:

```
def print_something(this,stderr=False):
    '''This does something.

    Keyword arguments:
    stderr -- set to True to print to stderr (default False)
    '''

    if stderr:
        # import sys.stderr
        from sys import stderr

        # print to stderr channel, converting this to str
        print >> stderr,str(this)

        # job done, return
        return

    # print to stdout, converting this to str
    print (str(this))

    return
```

Then the help text would be:

```
help(print_something)
```

```
Help on function print_something in module __main__:

print_something(this, stderr=False)
    This does something.

    Keyword arguments:
    stderr -- set to True to print to stderr (default False)
```

The above example represents a 'good' level of commenting as the code broadly adheres to the style suggestions and most of the major features are covered. It is not quite 'very good/excellent' as the description of the purpose of the method (rather important) is trivial and it fails to describe the input this in any way. An excellent piece would do all of these things, and might well tell us about any dependencies (e.g. requires sys if stderr set to True).

An inadequate example would be:

```
def print_something(this,stderr=False):
    '''This prints something'''
    if stderr:
        from sys import stderr
        print >> stderr,str(this)
        return
    print (str(this))
```

It is inadequate because it still only has a trivial description of the purpose of the method, it tells us nothing about inputs/outputs and there is no commenting inside the method.

Word limit

There is no word limit per se on the computer codes, though as with all writing, you should try to be succint rather than overly verbose.

Code style

A good to excellent piece of code would take into account issues raised in the style guide. The 'degree of excellence' would depend on how well you take those points on board.

```
# All imports go here. Run me first!
import datetime
from pathlib import Path  # Checks for files and so on
import numpy as np  # Numpy for arrays and so on
import pandas as pd
import sys
import matplotlib.pyplot as plt  # Matplotlib for plotting
# Ensure the plots are shown in the notebook
%matplotlib inline

import gdal
import osr
import numpy as np

from geog0111.geog_data import procure_dataset

if not Path("data/mod14_data").exists():
    _ = procure_dataset("mod14_data", destination_folder="data/mod14_data")
else:
    print("Data already available")
```

```
Data already available
```

2.8.28 Group project: Fire and teleconnections

There is much public and scientific interest in monitoring and predicting the activity of wildfires and such topics are often in the media, or here for a more recent event.

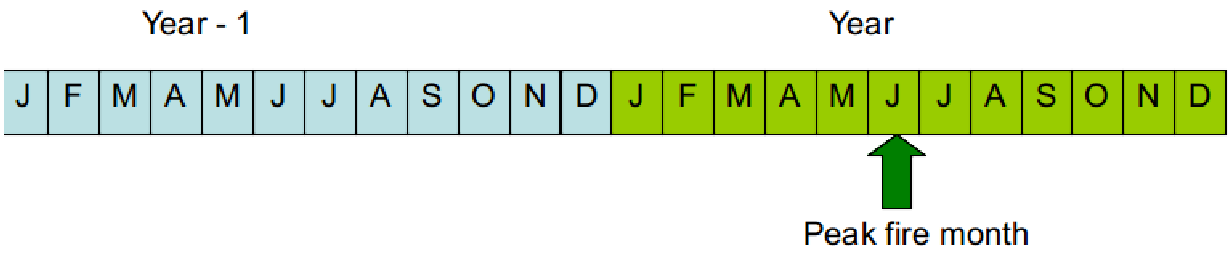
Part of this interest stems from the role fire plays in issues such as land cover change, deforestation and forest degradation and Carbon emissions from the land surface to the atmosphere, but also of concern are human health impacts, effects on soil, erosion, etc. The impacts of fire should not however be considered as wholy negative, as it plays a significant role in natural ecosystem processes.

For many regions of the Earth, there are large inter-annual variations in the timing, frequency and severity of wildfires. Whilst anthropogenic activity accounts for a large proportion of fires started, this is not in itself a new phenomenon, and fire has been and is used by humans to manage their environment.

Fires spread where: (i) there is an ignition source (lightning or man, mostly); (ii) sufficient combustible fuel to maintain the fire. The latter is strongly dependent on fuel loads and moisture content, as well as meteorological conditions. Generally then, when conditions are drier (and there is sufficient fuel and appropriate weather conditions), we would expect fire spread to increase. If the number of ignitions remained approximately constant, this would mean more fires. Many models of fire activity predict increases in fire frequency in the coming decades, although there may well be different behaviours in different parts of the world.

Here's Richard Feynman talking about fire and the carbon cycle.

```
from IPython.display import YouTubeVideo
YouTubeVideo('ITpDrdtGAmo')
```



Satellite data has been able to provide us with increasingly useful tools for monitoring wildfire activity, particularly since 2000 with the MODIS instruments on the NASA Terra and Aqua (2002) satellites. A suite of 'fire' products have been generated from these data that have been used in a large number of publications and practical/management projects.

There is growing evidence of 'teleconnection' links between fire occurence and large scale climate patterns, such as ENSO.

The proposed mechanisms are essentially that such climatic patterns are linked to local water status and temperature and thus affect the ability of fires to spread. For some regions of the Earth, empirical models built from such considerations have quite reasonable predictive skill, meaning that fire season severity might be predicted some months ahead of time.

In this Session..

In this session, you will be working in groups (of 3 or 4) to build a computer code in Python to explore links between fire activity and Sea Surface Temperature anomalies.

This is a team exercise, but does not form part of your formal assessment for this course. You should be able to complete the exercise in a 3-4 hour session, if you work effectively as a team. Staff will be on hand to provide pointers.

You should be able to complete the exercise using coding skills and python modules that you have previously experience of, though we will also provide some pointers to get you started.

In a nutshell, the **goal** of this exercise is

Using monthly fire count data from MODIS Terra, develop and test a predictive model for the number of fires per unit area per year driven by Sea Surface Temperature anomaly data.

The datasets should be created at 5 degree resolution on a latitude/longitude grid, as climate patterns will probably show some sort of response at broader spatial scales.

You should concentrate on building the model that predicts *peak fire count* in a particular year at a particular location, i.e. derive your model for annual peak fire count.

Datasets

We suggest that the datasets you use of this analysis, following Chen at al. (2011), are:

- MODIS Terra fire counts (2001-2011) (MOD14CMH). The particular dataset you will want from the file is 'SUBDATASET_2 [360x720] CloudCorrFirePix (16-bit integer)'.
- Climate index data from NOAA (e.g. see this list)

If you ever wish to take this study further, you can find various other useful datasets such as these.

Fire Data

The MOD14CMH CMG data are available from the UMD ftp server but have also been packaged for you and can be imported using the following code (this has already been done in the first imports cell above):

```
from geog0111.geog_data import procure_dataset

_ = procure_dataset("mod14_data",
                destination_folder="data/mod14_data")
```

The data are in HDF format, and you ought to be able to read them nto numpy arrays an operate with them. Note that there is data for MODIS TERRA and AQUA sensors, and if you want to use them together, you need to figure out the overlap period (AQUA only started providing data halfway through 2002).

The teleconnections data

Teleconnections data are available from a large number of places on the internet. You can find some sources of inspiration here. The data can be processed in two different ways: either as it is, or as anomalies (where you define a baseline temporal period, calculate some average value, and look at the residual between the actual index and the historical average). It's up to you what index you may want to use, and whether you want to use anomalies or directly the index value.

The predictive model

The model is very simple: we assume that the there is a linear relationship between the teleconnection at some given lag, and the recorded number of thermal anomalies. Bear in mind that the aim is to **predict** fire counts some months in advance using a teleconnection. As pseudo-code, for a pixel location `i, j`, you'd have something like this:

```
i, j # this is the pixel value
# Read in the peak fire month
peak_fire_month = get_peak_fire_month(i, j)
# Read in peak fire counts for all years for the pixel of interest
y = get_all_fire_counts_for_all_years(i, j)
# Loop over some lags
for lag in 0, ..., 12
do
    # Get the lagged teleconnection
    x = get_teleconnection_for_all_years(peak_fire_month - lag)
    # Perform linear regression and store the results
    m[lag], c[lag], r2[lag] = linear_regression(x, y)
done
best_lag = argmax(r2) # Select best lag
store_model(i, j, best_lag, m[best_lag], c[best_lag], r2[best_lag])
```

Splitting the tasks

You may want to assign tasks to individual members of the group. A reasonable split might be

- One person is responsible for the **satellite data**. This includes creating a 5 degree global resolution monthly dataset, and from it, derive for each grid cell, a peak fire month dataset, as well as a dataset with the fire counts at each peak fire month for all available years (more hints below)
- Another person might be in charge for getting hold of the **teleconnections dataset**, and process it into a suitable array
- Finally, some other person could be in charge of combining both fire counts and teleconnections datasets together and developing and testing a **linear model** to predict fire counts.

The satellite fire counts data

- You should probably start using the TERRA data. Files are named `MOD14CMH.YYYYMM.005.01.hdf`. By now, you should be familiar with the naming convention, and figure out that `YYYYMM` is the year and month the dataset refers to.
- Remember that this is a dataset of **fire counts**. You may want to check for weird data (e.g. can fire counts be less than zero?)
- The satellite data need to be aggregated to a coarser resolution of 5 degrees. This means that you have to **sum** the fire counts for every 10x10 original pixels, discarding missing values and so on. There's two ways of doing this: a loop and an array-based one.
  - **Loop based aggregation**

```
for i in range(int(nx/size)):
   for j in range(int(ny/size)):
       aggregated[i,j] = aggregate_function(original_data[i*size:((i+1)*size),
                                                          j*size:((j+1)*size)])
```

  - **Array based aggregation**: If the window size fits neatly in the data, you can just reshape the array and aggregate over as

```
aggregated = aggregate_function(original_data.reshape(
                                (nx/nsize, nsize, ny/nsize, nsize)),
                               axis=(1,3))
```

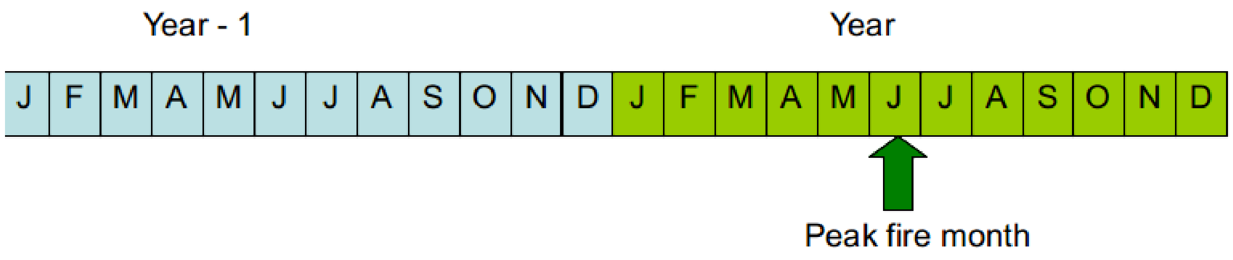
- You ought to discard 2000 as there are only two months of data available for that year.
- A reasonable data model for this is a numpy array of size `n_months*n_years, nn, mm`. You may also want to store the months and years as a 2D array (e.g. `n_months*n_years,2`)
- Once you have this, you can loop over your fire array, selecting all the fire counts for each year (e.g. 12 numbers) for each pixel, and finding the location of the maximum (using e.g. `np.argmax`). You'll end up with an array of size `n_years, nn, mm`.
- So now you need to decide which month is, on average, the peak fire month. How could you do this? The mean is problematic, as you may end up with something like e.g. 6.5. Are there other statistical metrics that might results useful (e.g. see this)?

At this point, you should end up with one main array of size `nn, mm` (e.g. 36, 72), where every grid cell is populated by the peak fire number defined from the data, as well as the `n_years,nn,nn` array with the fire counts at peak fire month. Make sure it is clear what you mean by month number!!! Note that we also have data available for the AQUA platform, and you may want to use it too.

If you plot them, they should look like this:

The teleconnections

- You can start with one teleconnection, but you may want to explore others.
- A data structure for the telecon data that might be useful and convenient is to stack two consecutive years together. It then makes it easy to loop over different lags (e.g. if your peak fire month for a pixel is February, then examining the 12 previous months can be done by looking at positions 13 (Feb), 12 (Jan), 11 (Dec previous year) and so on. Here's how it looks like graphically



With this in mind, you should aim to have an array with your teleconnection (or a dictionary of teleconnections!) with size `n_years*24`.

You can use the ESRL NOAA webpage to plot time series of your index (. . . indices) of choice, and make sure you have processed the data correctly.

Developing the model

The model is a simple linear model that relates the teleconnection value at some lag l with respect to the peak fire month (tc_l) with the number of fire counts for a given cell, $N_{counts,\,i,j}$

$$m \cdot tc_l + c = N_{counts,\,i,j}$$

- You have to split the data into a testing and training set: select a number of years to fit the model, and another one to test the model.
- The training will produce estimates of the slope m and intercept c for every pixel and time lag l.
- There are different ways to select the best lag, but the simplest one could be in terms of the coefficient of determination r^2: just choose the highest!
- You should store the per grid cell model parameters, as well as probably the r^2 (why?), and the optimal lag.

You can see an example of how this works on a particular grid cell in the following plot

Model testing

The previous plot shows that for a particular grid cell, the model is very succesful in the training phase (open orange circles), but the predictive phase is a bit of a mixed bag, with half the predictions being an important underestimate. However, this is a single cell validation. You ought to think of how to best validate the model. Scatter plots similar to the one above are useful, but maybe you have other ideas.

A useful test of the robustness of the model in the light of limited data is to use bootstrapping: a fraction of the dataset (e.g. a few years) are left out for testing as you've done above, but the procedure is repeated by replacing the training and testing datasets, to give you a distribution of the predictive performance of the model. There is a practical and very readable introduction to bootstrapping here

You may want to pack all the model development, teleconnection gathering and analysis in one top-level function that allows you to quickly repeat the fitting and testing procedure using different sets of years.

Refinements

- You may also want to consider using the AQUA fire counts data, which is also available in the data folder.
- Different teleconnections might explain different variations in weather in different regions. Fire is enhanced with high temperatures, low rainfall and high winds. You may want to check other connections and see how the models.
- Several teleconnections might be used together. For example, Chen et al (2011) use two indices together in a multi-linear model. This can be written as

$$N_{counts\ i,j} = \beta_0 + \beta_1 \cdot tc_{1,L1} + +\beta_2 \cdot tc_{2,L2} + \cdots + +\beta_N \cdot tc_{N,LN}.$$

- Other metrics of fire activity might be more representative, such as the sum of all fire counts around the peak fire month.

```
# All imports go here. Run me first!
import datetime
from pathlib import Path  # Checks for files and so on
import numpy as np  # Numpy for arrays and so on
import pandas as pd
import sys
import matplotlib.pyplot as plt  # Matplotlib for plotting
# Ensure the plots are shown in the notebook
%matplotlib inline

import gdal
import osr
import numpy as np

from geog0111.geog_data import procure_dataset

if not Path("data/mod14_data").exists():
    _ = procure_dataset("mod14_data", destination_folder="data/mod14_data")
else:
    print("Data already available")
```

```
Data already available
```

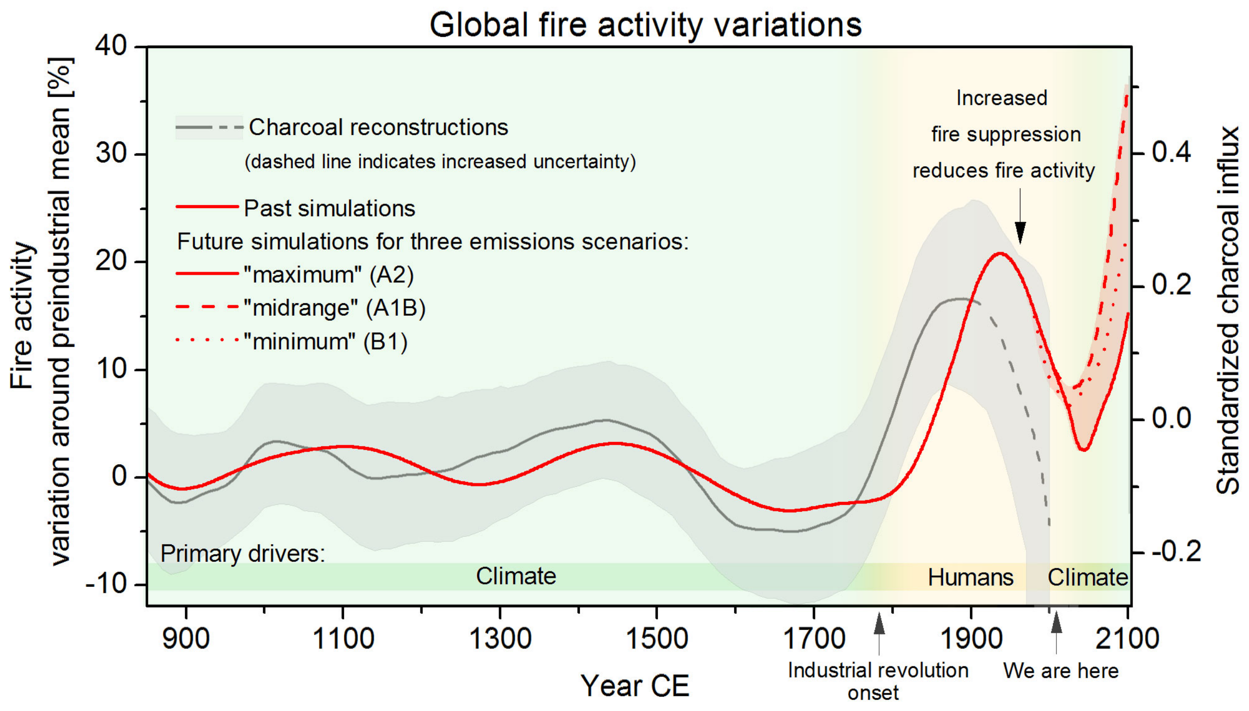
2.8.29 Group project: Fire and teleconnections

There is much public and scientific interest in monitoring and predicting the activity of wildfires and such topics are often in the media, or here for a more recent event.

Part of this interest stems from the role fire plays in issues such as land cover change, deforestation and forest degradation and Carbon emissions from the land surface to the atmosphere, but also of concern are human health impacts, effects on soil, erosion, etc. The impacts of fire should not however be considered as wholy negative, as it plays a significant role in natural ecosystem processes.

For many regions of the Earth, there are large inter-annual variations in the timing, frequency and severity of wildfires. Whilst anthropogenic activity accounts for a large proportion of fires started, this is not in itself a new phenomenon, and fire has been and is used by humans to manage their environment.

Fires spread where: (i) there is an ignition source (lightning or man, mostly); (ii) sufficient combustible fuel to maintain the fire. The latter is strongly dependent on fuel loads and mositure content, as well as meteorological conditions. Generally then, when conditions are drier (and there is sufficient fuel and appropriate weather conditions), we would expect fire spread to increase. If the number of ignitions remained approximately constant, this would mean more fires. Many models of fire activity predict increases in fire frequency in the coming decades, although there may well be different behaviours in different parts of the world.



Satellite data has been able to provide us with increasingly useful tools for monitoring wildfire activity, particularly since 2000 with the MODIS instruments on the NASA Terra and Aqua (2002) satellites. A suite of 'fire' products have been generated from these data that have been used in a large number of publications and practical/management projects.

There is growing evidence of 'teleconnection' links between fire occurence and large scale climate patterns, such as ENSO.

The proposed mechanisms are essentially that such climatic patterns are linked to local water status and temperature and thus affect the ability of fires to spread. For some regions of the Earth, empirical models built from such considerations have quite reasonable predictive skill, meaning that fire season severity might be predicted some months ahead of time.

In this Session..

In this session, you will be working in groups (of 3 or 4) to build a computer code in Python to explore links between fire activity and Sea Surface Temperature anomalies.

This is a team exercise, but does not form part of your formal assessment for this course. You should be able to complete the exercise in a 3-4 hour session, if you work effectively as a team. Staff will be on hand to provide pointers.

You should be able to complete the exercise using coding skills and python modules that you have previously experience of, though we will also provide some pointers to get you started.

In a nutshell, the **goal** of this exercise is

Using monthly fire count data from MODIS Terra, develop and test a predictive model for the number of fires per unit area per year driven by Sea Surface Temperature anomaly data.

The datasets should be created at 5 degree resolution on a latitude/longitude grid, as climate patterns will probably show some sort of response at broader spatial scales.

You should concentrate on building the model that predicts *peak fire count* in a particular year at a particular location, i.e. derive your model for annual peak fire count.

Datasets

We suggest that the datasets you use of this analysis, following Chen at al. (2011), are:

- MODIS Terra fire counts (2001-2011) (MOD14CMH). The particular dataset you will want from the file is 'SUBDATASET_2 [360x720] CloudCorrFirePix (16-bit integer)'.
- Climate index data from NOAA (e.g. see this list)

If you ever wish to take this study further, you can find various other useful datasets such as these.

Fire Data

The MOD14CMH CMG data are available from the UMD ftp server but have also been packaged for you and can be imported using the following code (this has already been done in the first imports cell above):

```
from geog0111.geog_data import procure_dataset

_ = procure_dataset("mod14_data",
                destination_folder="data/mod14_data")
```

The data are in HDF format, and you ought to be able to read them nto numpy arrays an operate with them. Note that there is data for MODIS TERRA and AQUA sensors, and if you want to use them together, you need to figure out the overlap period (AQUA only started providing data halfway through 2002).

The teleconnections data

Teleconnections data are available from a large number of places on the internet. You can find some sources of inspiration here. The data can be processed in two different ways: either as it is, or as anomalies (where you define a baseline temporal period, calculate some average value, and look at the residual between the actual index and the historical average). It's up to you what index you may want to use, and whether you want to use anomalies or directly the index value.

The predictive model

The model is very simple: we assume that the there is a linear relationship between the teleconnection at some given lag, and the recorded number of thermal anomalies. Bear in mind that the aim is to **predict** fire counts some months in advance using a teleconnection. As pseudo-code, for a pixel location `i, j`, you'd have something like this:

```
i, j # this is the pixel value
# Read in the peak fire month
peak_fire_month = get_peak_fire_month(i, j)
```

(continues on next page)

(continued from previous page)

```
# Read in peak fire counts for all years for the pixel of interest
y = get_all_fire_counts_for_all_years(i, j)
# Loop over some lags
for lag in 0, ..., 12
do
    # Get the lagged teleconnection
    x = get_teleconnection_for_all_years(peak_fire_month - lag)
    # Perform linear regression and store the results
    m[lag], c[lag], r2[lag] = linear_regression(x, y)
done
best_lag = argmax(r2) # Select best lag
store_model(i, j, best_lag, m[best_lag], c[best_lag], r2[best_lag])
```

Splitting the tasks

You may want to assign tasks to individual members of the group. A reasonable split might be

- One person is responsible for the **satellite data**. This includes creating a 5 degree global resolution monthly dataset, and from it, derive for each grid cell, a peak fire month dataset, as well as a dataset with the fire counts at each peak fire month for all available years (more hints below)
- Another person might be in charge for getting hold of the **teleconnections dataset**, and process it into a suitable array
- Finally, some other person could be in charge of combining both fire counts and teleconnections datasets together and developing and testing a **linear model** to predict fire counts.

The satellite fire counts data

- The satellite data need to be aggregated to a coarser resolution of 5 degrees. This means that you have to **sum** the fire counts for every 10x10 original pixels, discarding missing values and so on. You ought to discard 2000 as there are only two months of data available for that year.
- A reasonable data model for this is a numpy array of size `n_months*n_years, nn, mm`. You may also want to store the months and years as a 2D array (e.g. `n_months*n_years,2`)
- Once you have this, you can loop over your fire array, selecting all the fire counts for each year (e.g. 12 numbers) for each pixel, and finding the location of the maximum (using e.g. `np.argmax`). You'll end up with an array of size `n_years, nn, mm`.
- So now you need to decide which month is, on average, the peak fire month. How could you do this? The mean is problematic, as you may end up with something like e.g. 6.5. Are there other statistical metrics that might results useful (e.g. see this)?

At this point, you should end up with one main array of size `nn, mm` (e.g. 36, 72), where every grid cell is populated by the peak fire number defined from the data, as well as the `n_years,nn,nn` array with the fire counts at peak fire month. Make sure it is clear what you mean by month number!!! Note that we also have data available for the AQUA platform, and you may want to use it too.

If you plot them, they should look like this:

The teleconnections

- You can start with one teleconnection, but you may want to explore others.

- A data structure for the telecon data that might be useful and convenient is to stack two consecutive years together. It then makes it easy to loop over different lags (e.g. if your peak fire month for a pixel is February, then examining the 12 previous months can be done by looking at positions 13 (Feb), 12 (Jan), 11 (Dec previous year) and so on.

With this in mind, you should aim to have an array with your teleconnection (or a dictionary of teleconnections!) with size `n_years*24`.

You can use the ESRL NOAA webpage to plot time series of your index (. . . indices) of choice, and make sure you have processed the data correctly.

Developing the model

The model is a simple linear model that relates the teleconnection value at some lag l with respect to the peak fire month (tc_l) with the number of fire counts for a given cell, $N_{counts,\,i,j}$

$$m \cdot tc_l + c = N_{counts,\,i,j}$$

- You have to split the data into a testing and training set: select a number of years to fit the model, and another one to test the model.
- The training will produce estimates of the slope m and intercept c for every pixel and time lag l.
- There are different ways to select the best lag, but the simplest one could be in terms of the coefficient of determination r^2: just choose the highest!
- You should store the per grid cell model parameters, as well as probably the r^2 (why?), and the optimal lag.

You can see an example of how this works on a particular grid cell in the following plot

Solution: getting the data

Getting the data needs the completion of a few stages:

1. Searching for all the MODIS fire counts files
2. Opening & reading each file, filtering the ocean grid cells (indicated by `-1`)
3. Aggregate the data to the coarser grid
4. Create a 3D stack for all the months since the beggining of the time series.
5. Find the peak month

These functionality has been implemented in `` `geog0111/fire_practical_satellite.py `` <geog0111/fire_practical_satellite.py>`__, where a bunch of functions are defined. Here are the headings of those functions:

```python
def get_mod14(folder="data/mod14_data", skip_files=2):
    """Gets hold of the MOD14 data. We can skip a couple of files
    from 2000, and just read in the data from 2001 to 2016.

    Parameters
    ------------
    folder: str
        The folder where the files are all located
    skip_files: int
        Number of files to skip at the start of the time series

```

(continues on next page)

(continued from previous page)

```
    Returns
    -------
    REturns a list of pathlib objects with all the MOD14CMH HDF files
    """

def subsample_data(data, size=10, aggr=np.sum):
    """Subsample a 2D dataset by aggregating. Assumes that the input
    image or dataset will be aggregated over chunks of `size`
    by `size` pixels. You can select what aggregation method you
    want to use.

    Parameters
    ----------
    data: ndarray
        A 2D array of fire counts (for example)
    size: int
        The size of the aggregation in pixels. Identical for x and y
    aggr: function
        A function to perform the aggregation. By default, sum

    Returns
    ---------
    A downsampled and aggregated dataset
    """

def read_mod14_data(fich, layer=1):
    """Read the MOD14 data. Uses first layer by default.

    Parameters
    ----------
    fich: str
        A MOD14 HDF file
    layer: int
       The layer in the HDF file.

    Result
    -------
    Returns the data. Pixels with values <0 are set to 0.
    """


def create_subsampled_dataset():
    """Creates a subsampled datasets and extracts the dates.

    Returns
    --------
    Returns two arrays: a dates array (2 columns, years and months), as well
    as a 3D array of months*years, nx, ny cells.
    """
```

`create_subsampled_dataset` is in charge of creating the subsampled dataset. It does so by calling `get_mod14` which provides a list of all the files, each file can then read into numpy arrays using GDAL in `read_mod14_data`. The individual months can be subsampled by `subsample_data`. This structure is very efficient: you can just loop over the filenames that are returned by `get_mod14`, and then read and aggregate before stacking the result.

Once the result is stacked, we can proceed grid by grid to find the peak fire month. Since the peak fire actiivity month might change from year to year, so perhaps using the "most popular value" (e.g the mode) is the best approach. Additionally, we need to store the number of fires for the peak month for all the years. This is done in `find_peak_and_fires`.

Solution: getting hold of the teleconnection data

This is implemented in a single function in `` `geog0111/fire_practical_telecon.py `` <geog0111/fire_practical_telecon.py>'__. We can observe that in the NOAA site, a bunch of teleconnections are available. They are all under the same URL, and have the same format. The format is plain ASCII, with some headers, as well as a "footer" at the end. Missing data are usually encoded by -9999, and each row in the file contains the index for one year, each column storing one month.

We have satellite data from 2001 onwards, so we need teleconections from 2000 (to cover the previous year). The code produces a `n_years, 24` array for easy indexing.

```
def get_telecon_data(
    telecon="nina34.data",
    dest_folder="data/mod14_data/",
    base_url="https://www.esrl.noaa.gov/psd/data/correlation/",
    start_year=2000,
    end_year=2016,
):
    """Downloads and processes the telenconnection data for easy
    model development. It returns a 2D array, where each row
    has 24 elements: element 12 is the teleconnection for January
    of the relevant year, and elements 0 to 11 are the teleconnection
    values for the months of the previous year.

    Parameters
    ------------
    telecon: str
        The name of the teleconnection. You can look it up from
        [this page](https://www.esrl.noaa.gov/psd/data/climateindices/list/)
    dest_folder: str
        The destination folder. It'll save the teleconnection there
    base_url: str
        The base URL for the NOAA server
    start_year: int
        The start year ;-)
    end_year: int
        The end year

    Returns
    ----------
    A 2D array with the teleconnection.
    """
```

Solution: the model fitting part

Once the teleconnection, fire peak month and fire counts for different years are available, we can proceed to fit the model on a grid cell by grid cell basis. The code that does this is available on `` `geog0111/fire_practical_model.py `` <geog0111/fire_practical_model.py>'__ and looks like this:

```
for i in range(nn):
    for j in range(mm):
```

(continues on next page)

(continued from previous page)

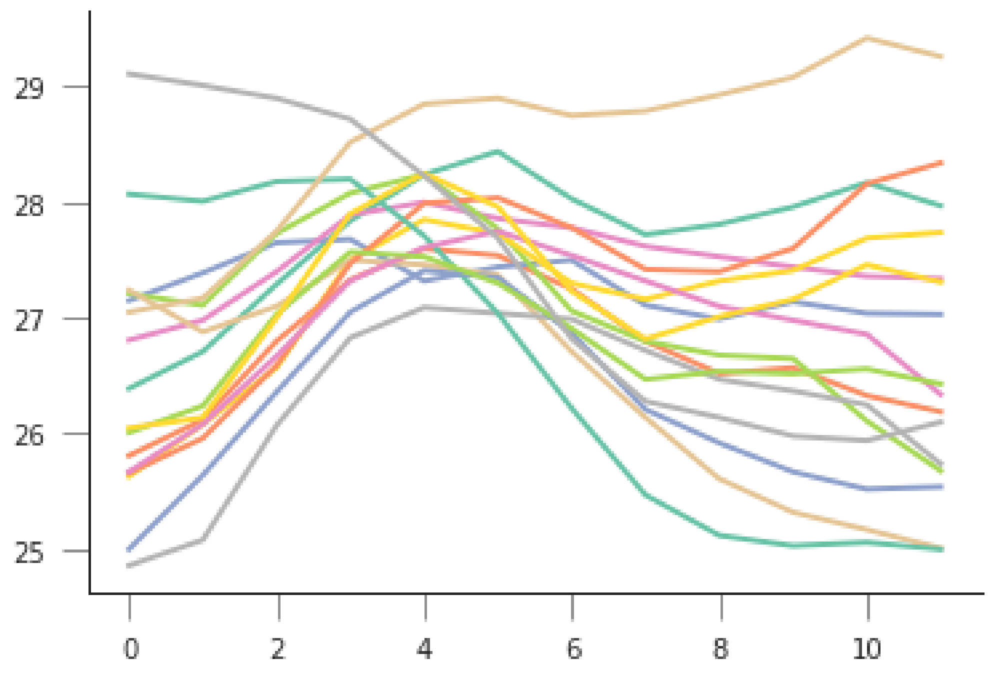
```
        # Now doing grid cell i, j
        # Get the peak fire activity month, and subtract one
        pf_month = peak_fire_month[i, j] - 1  # 1-based month
        # Get the fire counts for all training years
        counts = fire_count_year[:train_years, i, j]
        # The list comprehension sweeps over the telecon,
        # starting on peak fire month and going back 12 months
        reg = [
            scipy.stats.linregress(
                telecon[:train_years, pf_month - lager], counts
            )
            for lager in range(0, -12, -1)
        ]
        # Find the lag with the highest r^2
        iloc = np.argmax([x.rvalue ** 2 for x in reg])
        # etc
```

```
from geog0111.fire_practical_model import *
from geog0111.fire_practical_satellite import *
from geog0111.fire_practical_telecon import *

telecon = get_telecon_data()
mod14_dates, mod14_data = create_subsampled_dataset()
peak_fire_month, fire_count_year = find_peak_and_fires(mod14_dates, mod14_data)
slope, intercept, best_r2, best_lag = fit_model(
    telecon, peak_fire_month, fire_count_year, train_years=12)
```

```
plt.plot(telecon[:, 12:].T, '-')
```

```
[<matplotlib.lines.Line2D at 0x7fb3ceb25d30>,
 <matplotlib.lines.Line2D at 0x7fb3ceb25e80>,
 <matplotlib.lines.Line2D at 0x7fb3ceb25fd0>,
 <matplotlib.lines.Line2D at 0x7fb3ceb31160>,
 <matplotlib.lines.Line2D at 0x7fb3ceb312b0>,
 <matplotlib.lines.Line2D at 0x7fb3ceb31400>,
 <matplotlib.lines.Line2D at 0x7fb3ceb31550>,
 <matplotlib.lines.Line2D at 0x7fb3ceb316a0>,
 <matplotlib.lines.Line2D at 0x7fb3cf3a5a20>,
 <matplotlib.lines.Line2D at 0x7fb3ceb31908>,
 <matplotlib.lines.Line2D at 0x7fb3ceb31a58>,
 <matplotlib.lines.Line2D at 0x7fb3ceb31ba8>,
 <matplotlib.lines.Line2D at 0x7fb3ceb31cf8>,
 <matplotlib.lines.Line2D at 0x7fb3ceb31e48>,
 <matplotlib.lines.Line2D at 0x7fb3ceb31f98>,
 <matplotlib.lines.Line2D at 0x7fb3ceb38128>]
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```

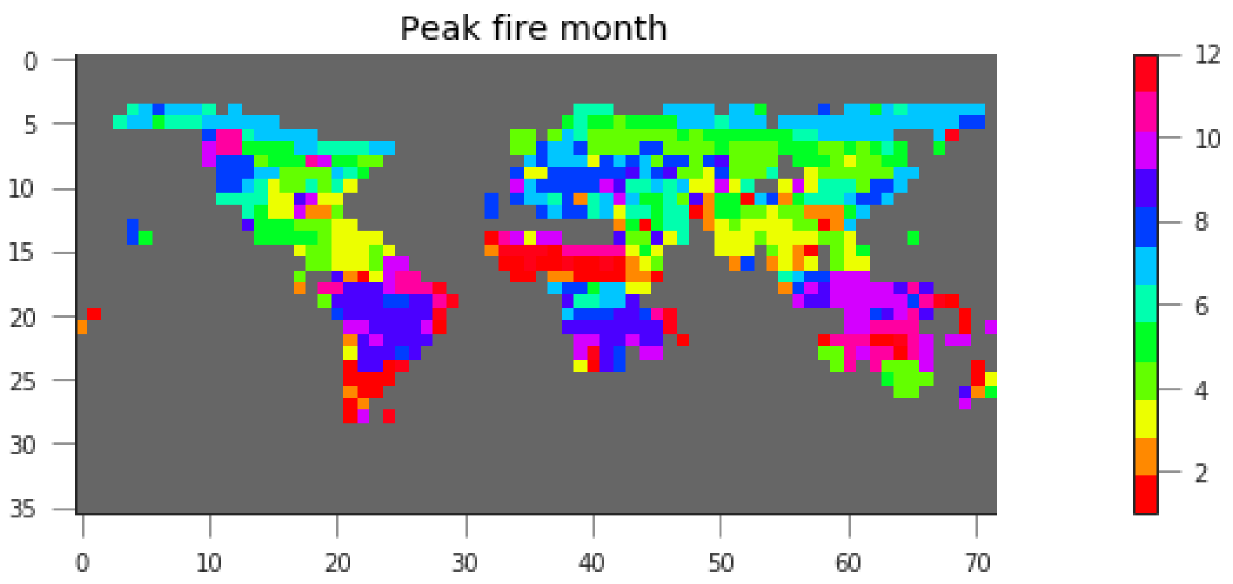


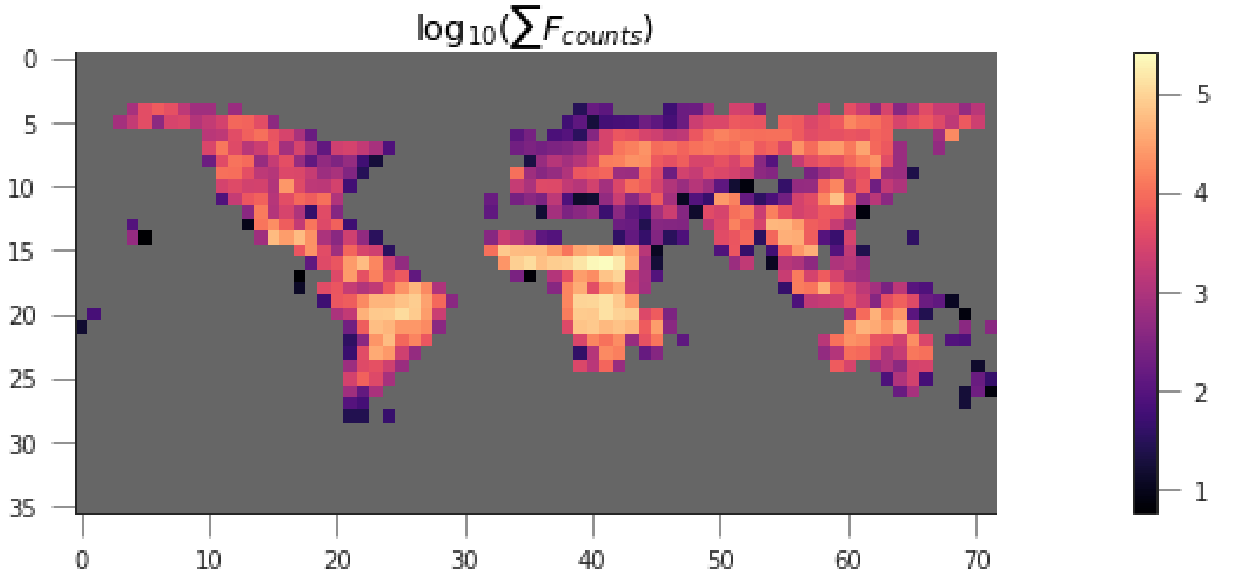
```
fig, axs = plt.subplots(nrows=2, ncols=1, figsize=(21,9))
cmap = plt.cm.get_cmap("hsv", 12)
cmap.set_under("0.4")
im = axs[0].imshow(peak_fire_month, interpolation="nearest", vmin=1, vmax=12,
↪cmap=cmap)
plt.colorbar(im, ax=axs[0], fraction=0.6)
axs[0].set_title("Peak fire month")

cmap = plt.cm.magma
cmap.set_bad('0.4')
im = axs[1].imshow(np.log10(fire_count_year.sum(axis=0)),
                    interpolation="nearest", cmap=cmap)
plt.colorbar(im, ax=axs[1], fraction=0.6)
axs[1].set_title(r"$\log_{10}(\sum F_{counts})$")
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/ipykernel_launcher.
↪py:10: RuntimeWarning: divide by zero encountered in log10
  # Remove the CWD from sys.path while we load stuff.
```

```
Text(0.5,1,'$\log_{10}(\sum F_{counts})$')
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```





```
fig, axs = plt.subplots(nrows=2, ncols=1, figsize=(31,12))

cmap = plt.cm.get_cmap("hsv", 12)
cmap.set_under("0.4")
im = axs[0].imshow(best_lag, interpolation="nearest", vmin=1, vmax=12,cmap=cmap)
plt.colorbar(im, ax=axs[0], fraction=0.6)
axs[0].set_title("Best lag (months)")

cmap = plt.cm.magma
cmap.set_bad('0.4')
im = axs[1].imshow(best_r2,
                   interpolation="nearest", vmin=0.1, vmax=1,
                   cmap=cmap)
```

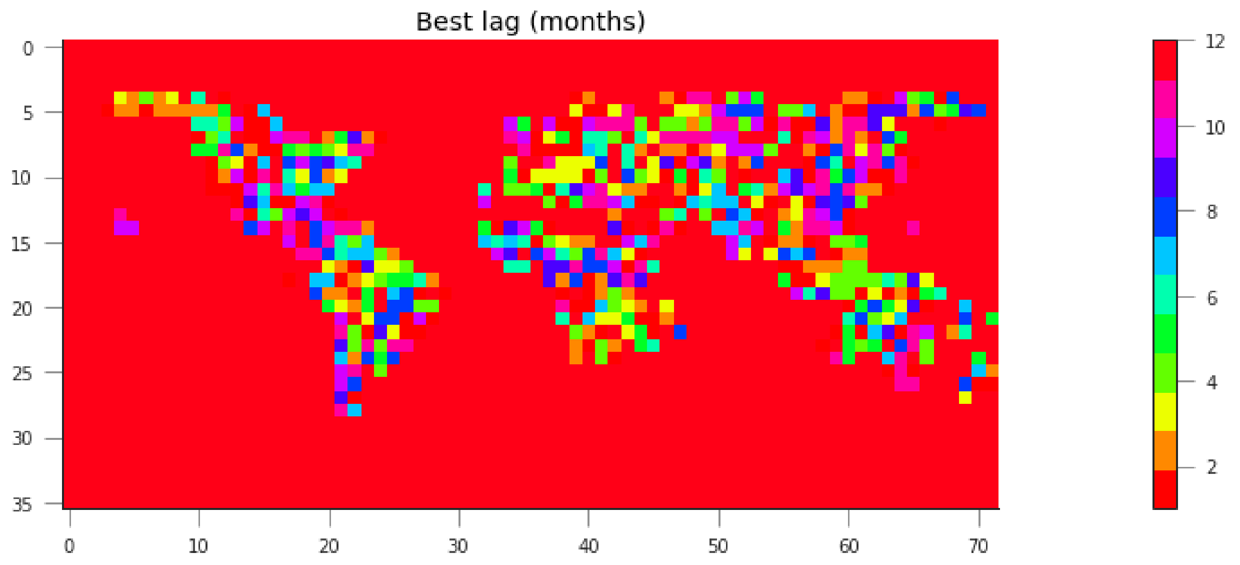
(continues on next page)

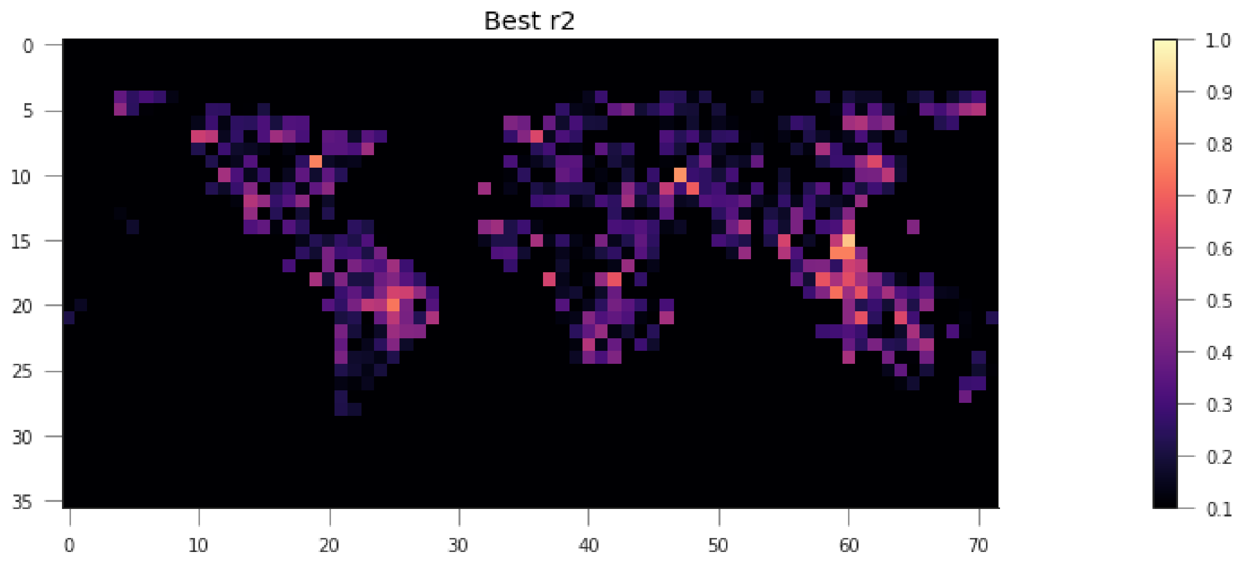
(continued from previous page)

```
plt.colorbar(im, ax=axs[1], fraction=0.6)
axs[1].set_title("Best r2")
```

```
Text(0.5,1,'Best r2')
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```





```
plt.hist(best_r2[best_r2 >= 0.1], bins=np.arange(0.1, 1, 0.1)-0.05, cumulative=True,␣
↪histtype="stepfilled",
```

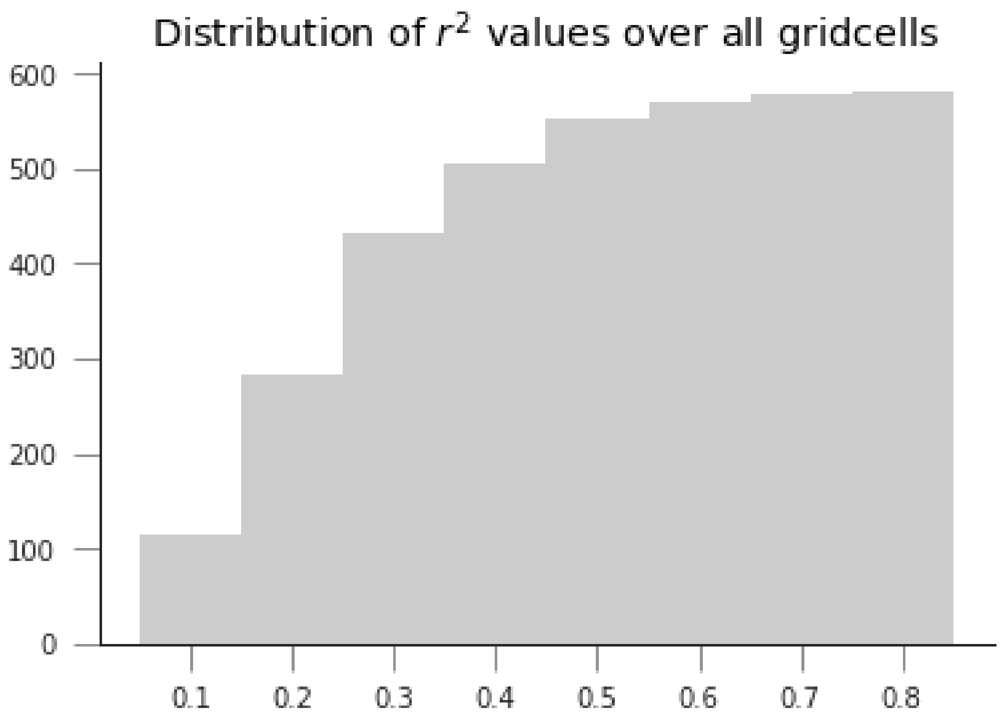
(continues on next page)

(continued from previous page)

```
        color="0.8")
plt.title("Distribution of $r^2$ values over all gridcells")
```

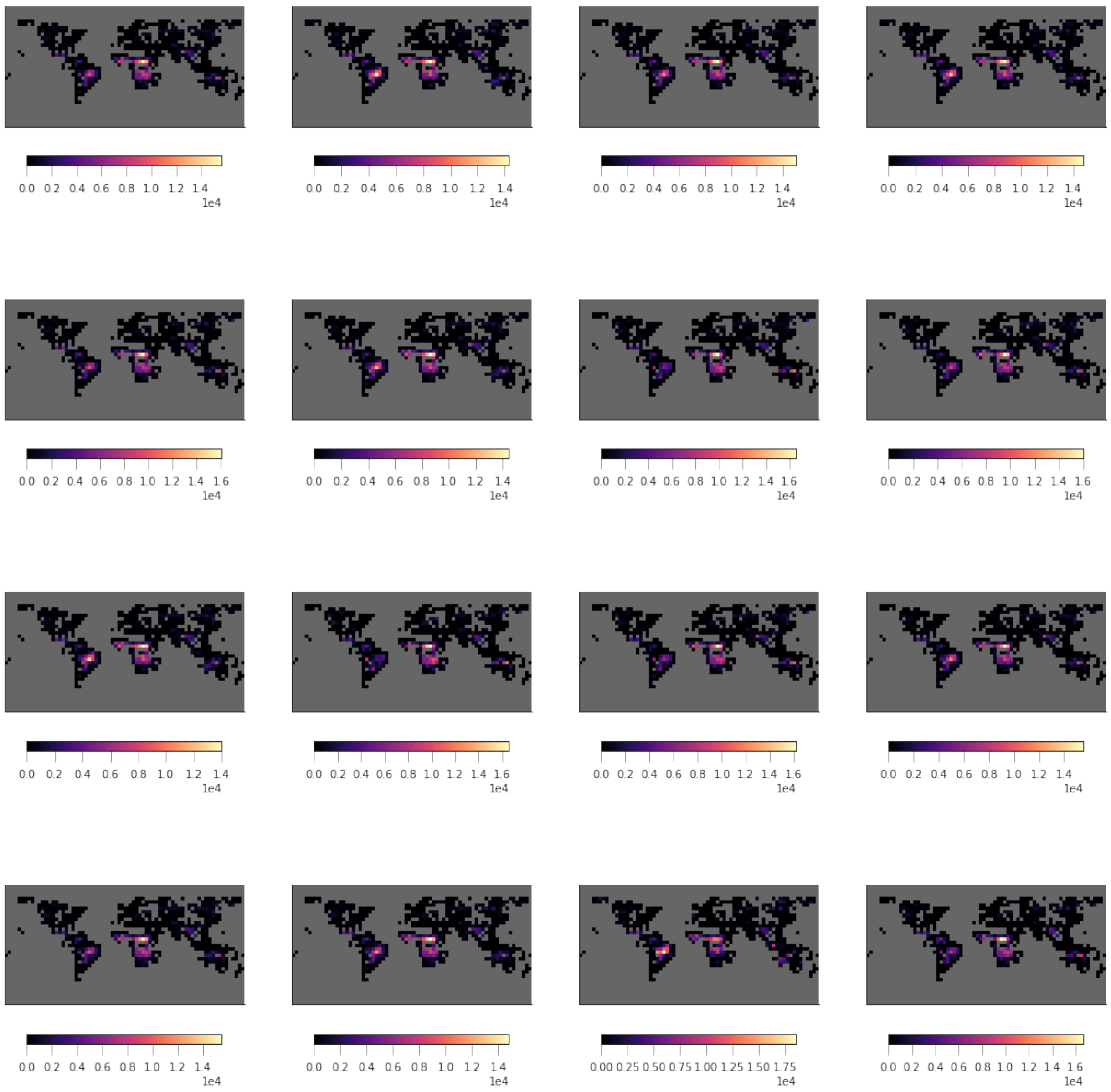
```
Text(0.5,1,'Distribution of $r^2$ values over all gridcells')
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



```
sensible_grids = best_r2 >= 0.1

predicted_fire_counts = np.ones((16, 36, 72))*np.nan

fig, axs = plt.subplots(nrows=4, ncols=4, sharex=True, sharey=True,
                       figsize=(18, 18))
axs = axs.flatten()

for year in range(16):
    x = telecon[year, :][11 + peak_fire_month]
    y = slope*x + intercept
    predicted_fire_counts[year, sensible_grids] = y[sensible_grids]
    im=axs[year].imshow(predicted_fire_counts[year], interpolation="nearest",
                 vmin=0, cmap=plt.cm.magma)
    axs[year].set_xticks([])
    axs[year].set_yticks([])

    plt.colorbar(im, ax=axs[year], fraction=0.05, orientation="horizontal")
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```


0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1e4
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1e4
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1e4
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1e4
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1e4
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1e4
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1e4
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1e4
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1e4
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1e4
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1e4
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1e4
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1e4
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1e4
0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 1e4
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1e4

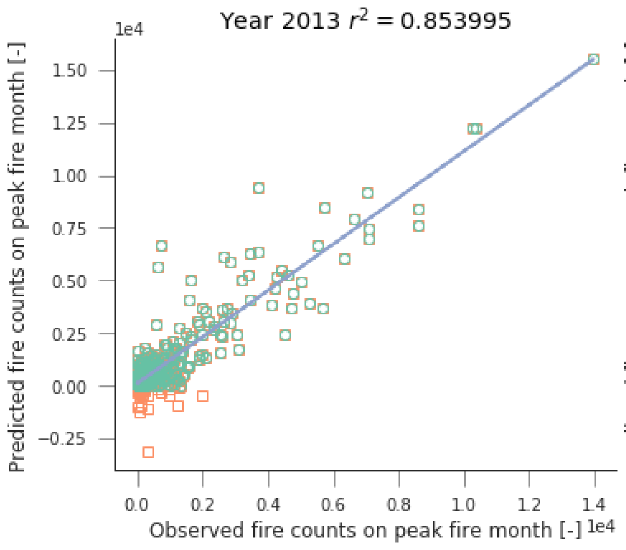
```
fig, axs = plt.subplots(nrows=2, ncols=2, figsize=(12,12))
axs = axs.flatten()
for i,year in enumerate(range(12, 16)):
    x = fire_count_year[year]
    y = predicted_fire_counts[year]
    axs[i].plot(x[sensible_grids], y[sensible_grids], 's', mfc="none")
    axs[i].plot(x[sensible_grids*(y>0)], y[sensible_grids*(y>0)], 'o', mfc="none")



    reg = scipy.stats.linregress(x[sensible_grids*(y>0)], y[sensible_grids*(y>0)])
```

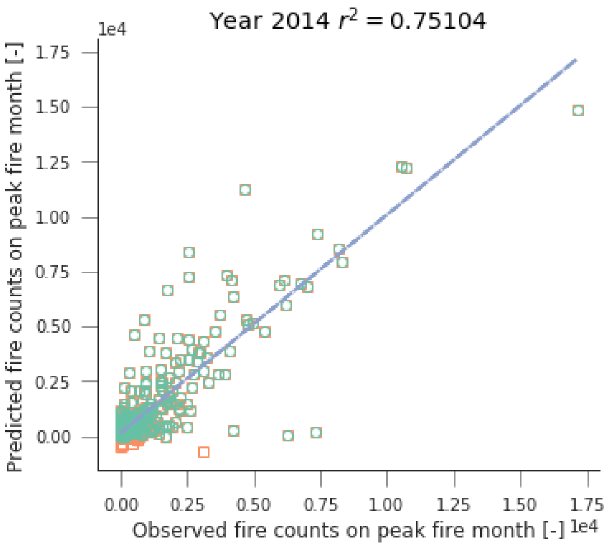
(continues on next page)

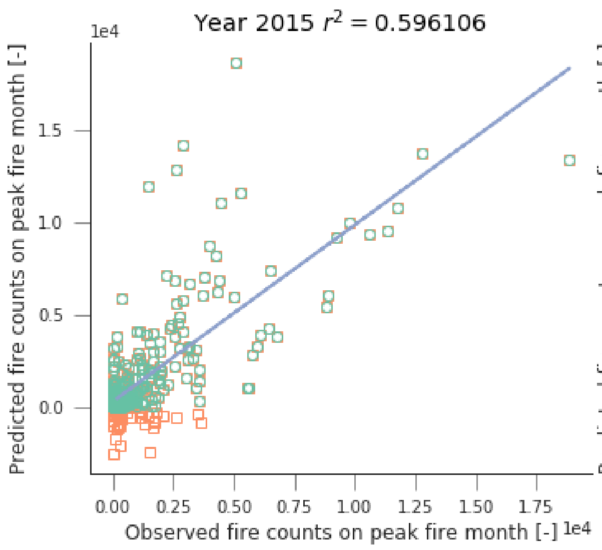
(continued from previous page)

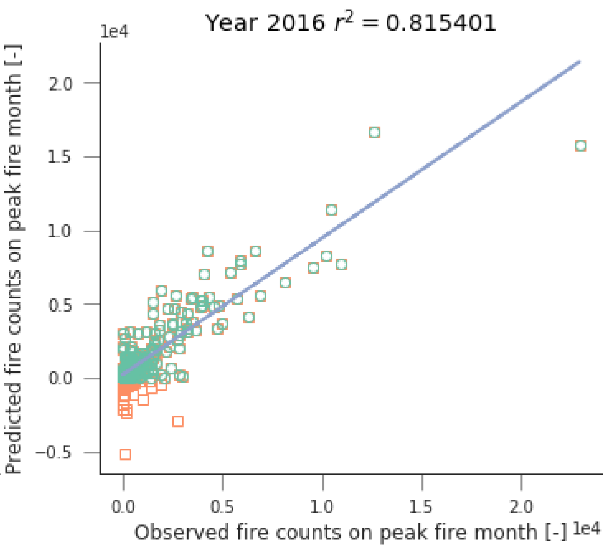
```
    axs[i].plot(x[sensible_grids*(y>0)],
                x[sensible_grids*(y>0)]*reg.slope + reg.intercept, '--')
    axs[i].set_title(f"Year {2001+year:d} " + "$r^2=%g$"%reg.rvalue**2)
    axs[i].set_xlabel("Observed fire counts on peak fire month [-]")
    axs[i].set_ylabel("Predicted fire counts on peak fire month [-]")
```

```
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/ipykernel_launcher.
↪py:7: RuntimeWarning: invalid value encountered in greater
  import sys
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/ipykernel_launcher.
↪py:10: RuntimeWarning: invalid value encountered in greater
  # Remove the CWD from sys.path while we load stuff.
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/ipykernel_launcher.
↪py:11: RuntimeWarning: invalid value encountered in greater
  # This is added back by InteractiveShellApp.init_path()
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/ipykernel_launcher.
↪py:12: RuntimeWarning: invalid value encountered in greater
  if sys.path[0] == '':
/home/ucfajlg/miniconda3/envs/python3/lib/python3.6/site-packages/matplotlib/font_
↪manager.py:1328: UserWarning: findfont: Font family ['sans-serif'] not found.␣
↪Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```









2.8.30 Connecting to notebooks from outside UCL

You can find more information about accessing the Lab computers in this webpage.

Jupyter notebooks are accessed through the browser, and it is possible to run the jupyter notebook on the computers in the UCL Geography lab, but still access the browser remotely (either from a laptop while on UCL, or from anywhere else).

It is also possible to access the UNIX shell remotely, and doing this is a requirement for accessing the Jupyter notebooks remotely. Let's first look at accessing the UNIX shell remotely first.

Remote access to your UCL shell

The UNIX lab computers can be accessed using the `ssh` command. This command is encrypted and allows one to connect to a remote computer through the Internet. On **Linux** and **MacOSX**, `ssh` is installed by default. On **Windows**, it has to be installed. A good option for this is MobaXterm, although there is also Kitty if you like more bare-bones systems (and a less restrictive license).

We will assume that you want to access your home directory. You can do this by "ssh-ing" to one of the bastion computers that are open to the outside internet:

- `arch.geog.ucl.ac.uk`
- `round.geog.ucl.ac.uk`
- `square.geog.ucl.ac.uk`
- `triangle.geog.ucl.ac.uk`

In Linux or MacOSX, you can simply do this using the command

```
ssh -YXC username@triangle.geog.ucl.ac.uk
```

where `username` needs to be substituted by your Geography Linux username. The system will prompt you for a password (note that if you start typing, it will not print anything back, this is normal), and as finish typing the password and pressing "Return", you'll be logged into your home space.

This isn't very exciting: the bastion computers have little software installed, and they're a *gateway* to the other computers in the lab. However, they work fine for transferring files to and from your own computer and Geography's Linux system.

Accessing a lab computer

Once you've logged into to one of the bastion hosts, you can use `ssh` again to log into a lag machine, e.g. `ankara`:

```
ssh -YXC username@ankara.geog.ucl.ac.uk
```

This is equivalent as being sitting down in front of the terminal in the lab.

Actually accessing the Jupyter notebooks remotely

Once you have access, you can launch the Jupyter notebook

jupyter notebook –no-browser –ip=* –port=8889

You will see some text scroll down the screen. This is what I get if I run this command, your result will be slightly different:

```
[W 16:17:05.852 NotebookApp] WARNING: The notebook server is listening on all IP␣
↪addresses and not using encryption. This is not recommended.
[I 16:17:05.878 NotebookApp] [jupyter_nbextensions_configurator] enabled 0.2.7
[I 16:17:05.883 NotebookApp] Serving notebooks from local directory: /home/ucfajlg
[I 16:17:05.883 NotebookApp] 0 active kernels
[I 16:17:05.883 NotebookApp] The Jupyter Notebook is running at:
[I 16:17:05.883 NotebookApp] http://[all ip addresses on your system]:8889/?
↪token=0b856e664152b8fa3a4b6e688945f3e59e490cf5fd0c2f05
[I 16:17:05.884 NotebookApp] Use Control-C to stop this server and shut down all␣
↪kernels (twice to skip confirmation).
[C 16:17:05.884 NotebookApp]
```

(continues on next page)

(continued from previous page)

```

    Copy/paste this URL into your browser when you connect for the first time,
to login with a token:
    http://localhost:8889/?token=0b856e664152b8fa3a4b6e688945f3e59e490cf5fd0c2f05
```

We note that we can connect to the notebook by using port number 8889, and using this complicated token. For the time being, **just note down the port number**, 8889. Sometimes, you might get a different port as this one is already in use. Note the number, and modify the commands below accordingly.

In order to connect, you can use the following command in your local remote machine (open a new shall, and it is all in one line, the \ is just a line continuator)

```
ssh -L 8889:localhost:8889 username@triangle.geog.ucl.ac.uk \
     ssh -L 8889:localhost:8889 -N username@ankara.geog.ucl.ac.uk
```

This should just hang on the shell without much going on. Open your browser, and visit the URL that we had above, and you should see the notebook in all its glory!

CHAPTER 3

Indices and tables

- genindex
- modindex
- search