Melody Documentation

Release 0.1.1

Rupert Ford

Mar 14, 2017

Contents:

Melody is a lightweight parameter search and optimisation tool originally designed to support searching and optimising over the performance landscape of software. In particular, it is being designed to support code composition options (loop ordering, loop fusion etc), code compilation and link options (compilers and compiler flags) and code configuration options (tunable parameters such as a cut off radius, or preconditioner). However Melody has been written in a relatively generic way so can be used to search over, or optimise any search space.

Melody takes its name from the fact that optimisation is often called tuning and most good tunes rely on a great melody!

CHAPTER 1

Installation

Python version

Melody is currently tested with Python 2.7.

Using pip

You can install and uninstall using pip.

```
$ sudo pip install melody
$ sudo pip uninstall melody
```

If you don't have admin rights you can install and uninstall locally.

```
$ pip install melody --user
$ pip uninstall melody
```

Note: In some systems the resultant installation directory for a local install is not automatically added to the PYTHONPATH so must be done manually. The installation path will be `${HOME}/.local/lib/pythonx.y/site-packages` where `x.y` is the version of Python being used.

Downloading and installing

We recommend using pip for installation but if you would prefer to download and install locally then follow the instructions in this section.

Latest release

First download the source from github. You can download a zip file ...

```
$ wget https://github.com/rupertford/melody/archive/0.1.1.zip
$ unzip 0.1.1.zip
```

or clone the repository and switch to the latest release ...

```
$ git clone https://github.com/rupertford/melody.git
$ git checkout tags/0.1.1
```

Latest stable version

First download the source from github. You can download a zip file ...

```
$ wget https://github.com/rupertford/melody/archive/master.zip
$ unzip master.zip
```

or clone the repository and switch to the latest release ...

```
$ git clone https://github.com/rupertford/melody.git
```

Installation

Using setup.py

Melody includes an installation setup file called setup.py in its top level directory.

```
$ cd <melody_dir>
$ python setup.py install
```

If you do not have appropriate permissions you can perform a local installation instead

```
$ cd <melody_dir>
$ python setup.py install --user
```

Note: In some systems the resultant installation directory is not automatically added to the PYTHONPATH so must be done manually. The installation path will be `${HOME}/.local/lib/pythonx.y/site-packages` where `x.y` is the version of Python being used.

If you would like to uninstall melody (after installing using the setup.py script) you can do so using pip. You may see a number of error messages but the removal should complete successfully.

```
$ pip uninstall melody
```

Local pip install

This installation relies on you not moving or modifying the downloaded source code. In the top level directory type:

```
$ cd <melody_dir>
$ pip install -e .
```

Manual setup

This solution also relies on you not moving or modifying the downloaded source code. In this case you simply set your python path appropriately.

```
$ export PYTHONPATH=<melody_dir>/src
```

Testing

If you have successfully installed melody then you should be able to import it from Python.

```
$ python
>>> import melody
>>>
```

There is also a test suite, written to use pytest, that can be used to test the installation. Note, the tests are not included in the pip installation procedure. If you do not have pytest you can install it using `pip install pytest`.

```
$ py.test
============================= test session starts ==============================
platform linux2 -- Python 2.7.12, pytest-2.8.7, py-1.4.31, pluggy-0.3.1
rootdir: /xxx/melody, inifile:
plugins: cov-2.4.0
collected 33 items

src/melody/tests/inputs/choice_test.py ..
src/melody/tests/inputs/create_input_test.py ....
src/melody/tests/inputs/fixed_test.py .
src/melody/tests/inputs/floatrange_test.py .
src/melody/tests/inputs/input_test.py ..
src/melody/tests/inputs/intrange_test.py .
src/melody/tests/inputs/subsets_test.py .
src/melody/tests/inputs/switch_test.py ....
src/melody/tests/main/melody_test.py ....
src/melody/tests/search/bruteforce_test.py .x..
src/melody/tests/search/searchmethod_test.py .........

==================== 32 passed, 1 xfailed in 0.25 seconds ====================
```

CHAPTER 2

Running

Melody supports 3 main concepts, *inputs*, an *objective function* and a *method*.

Melody *inputs* define the search space over which the user would like to search. A number of pre-existing classes are provided to capture this space. These can also be written by the user. The *Inputs* section discusses inputs in more detail.

The user writes the Melody *objective function* to perform the particular task that requires analysis. Melody calls this function with appropriate inputs and expects to receive two return values once the function has completed. The *Objective Function* section discusses writing a Melody *objective function* in more detail.

The Melody *method* defines how to search the input space. Currently there is only one option here, which is brute force. The user may write their own search/optimisation method if they so choose. The *Method* section discusses Melody methods in more detail.

Melody

A *Melody* convenience class binds the three concepts (*inputs*, *objective function* and *method*) together and can be used to initiate the search.

Note: Please ignore the state argument in the `Melody` class. This is not used at the moment and is a placeholder for future developments.

Example

This section presents a simple Melody example which iterates over all possible combinations of two types of input, a choice of specified values and a range of integer values. The empty objective function simply returns True and a 0 each time it is called.

```
>>> from melody.inputs import Choice, IntRange
>>> from melody.search import BruteForce
>>> from melody.main import Melody
>>> inputs = [Choice(name="input1", inputs=["a", "b", "c"]),
              IntRange(name="input2", low=1, high=3, step=1)]
>>> def function(values):
>>>     return True, 0
>>> method = BruteForce
>>> melody = Melody(inputs=inputs, function=function, method=method)
>>> melody.search()
[{'input1': 'a'}, {'input2': 1}] True 0
[{'input1': 'a'}, {'input2': 2}] True 0
[{'input1': 'b'}, {'input2': 1}] True 0
[{'input1': 'b'}, {'input2': 2}] True 0
[{'input1': 'c'}, {'input2': 1}] True 0
[{'input1': 'c'}, {'input2': 2}] True 0
```

CHAPTER 3

Inputs

Melody inputs allow the user to specify the space that they would like to search. Inputs are specified as a list of individual input objects.

Supported

The following input classes are currently supported.

Fixed

For example

```
>>> from melody.inputs import Fixed
>>> inputs = [Fixed(name="option1", value="value1")]
```

The above example will generate an input named `option1` with the value `value1`.

Switch

For example

```
>>> from melody.inputs import Switch
>>> inputs = [Switch(name="option1", off="dark", on="light")]
```

The above example will generate an input named `option1` which can take one of two values, `dark` and `light`. If one of the arguments `off` or `on` is not provided then an empty string is returned for that option. If both of the arguments `off` and `on` are not provided then a `Runtime` exception is raised.

Choice

For example

```
>>> from melody.inputs import Choice
>>> inputs = [Choice(name="input2", inputs=["a", "b", "c"])]
```

The above example will generate an input named `input2` which can take one of three values, `a`, `b` and `c`. The list can be arbitrarily long.

The optional `pre` argument will prepend all inputs with the value contained in the string. For example

```
>>> from melody.inputs import Choice
>>> inputs = [Choice(name="in", inputs=["a", "b", "c"], pre="val_")]
```

The above example will generate an input named `in` with three values, `val_a`, `val_b` and `val_c`.

IntRange

For example

```
>>> from melody.inputs import IntRange
>>> inputs = [IntRange(name="range1", low=0, high=3, step=1)]
```

The above example will generate an input named `range1` with three integer values, `0`, `1` and `2`.

Warning: One might expect the integer 3 to appear, however, in keeping with the Python range function, this value is excluded.

IntRange

For example

```
>>> from melody.inputs import FloatRange
>>> inputs = [FloatRange(name="range2", low=0.0, high=0.4, step=0.1)]
```

The above example will generate an input named `range2` with four floating point values, `0.0`, `0.1`, `0.2` and `0.3`.

Warning: One might expect the value 0.4 to appear, however, in keeping with the Python range function, this value is excluded.

Subsets

For example

```
>>> from melody.inputs import Subsets
>>> inputs = [Subsets(name="combinations", inputs=["a", "b", "c"])]
```

The above example will generate an input named `combinations` with 8 combinations of values `[]`, `["a"]`, `["b"]`, `["c"]`, `["a", "b"]`, `["a", "c"]`, `["b", "c"]` and `["a", "b", "c"]`.

This option is useful when you have a set of inputs that are optional and can be used with each other in any combination e.g. compiler flags.

Multiple input objects

So far each of the supported input options has been presented individually. You might naturally be wondering why inputs has been defined as a list of input objects.

The reason for this is that an arbitrary number of input objects can be included in the inputs list. The implication of doing this is that all combinations of options are potentially valid inputs.

If we combine two of the earlier examples into one ...

```
>>> from melody.inputs import Fixed, Choice
>>> inputs = [FloatRange(name="range2", low=0.0, high=0.4, step=0.1),
              Choice(name="input2", inputs=["a", "b", "c"])]
```

we will be specifying the following valid combinations for `range2` and `input2`: `0.0, "a"`, `0.0, "b"`, `0.0, "c"`, `0.1, "a"`, `0.1, "b"`, `0.1, "c"`, `0.2, "a"`, `0.2, "b"`, `0.2, "c"`, `0.3, "a"`, `0.3, "b"` and `0.3, "c"`.

Note: The last input object specified in the list iterates fastest, followed by the penultimate one etc. So, in the above example the values for `input2` are changing more rapidly than the values for `range2`.

Extending

If the supported input classes do not cover your requirements then you can create your own input classes. All of the input classes inherit from the Input base class.

Note: Please ignore the state argument and method in the `Input` class. These are not used at the moment and are placeholders for future developments.

You can subclass the input class. For example, if you wanted to provide all values greater than a tolerance as inputs from a list of values:

```
>>> from melody.inputs import Input
>>> class IntTolerance(Input):
        ''' returns values if they are greater than a tolerance '''
        def __init__(self, name, inputs, tolerance):
            options = []
            for value in inputs:
                if value>tolerance:
                    options.append(value)
        Input.__init__(self, name, options)
>>> inputs = [IntTolerance("tolerance", [8, 9, 2, 4, 10], 7)]
```

Alternatively you can subclass one of the supporting input types if that is simpler. For example, if you wanted to append a string to all Switch values:

```
>>> from melody.inputs import Switch
>>> class SwitchAppend(Switch):
        ''' append a string to all switch values '''
        def __init__(self, name, off, on, append):
            Switch.__init__(self, name, off+append, on+append)
>>> inputs = [SwitchAppend("switch", "a", "b", "_value")]
```

Contributing

If you do create your own subclass and you think it might be a useful addition we ask that you consider contributing your code so that it can be incorporated into Melody for others to use.

CHAPTER 4

Objective Function

Melody objective functions are user-written Python functions that perform the action that the user would like to be optimised and/or investigated. Melody is not aware of what this action is, it simply calls the objective function with a set of inputs and collects the result(s).

If, for example, you wanted to find the time taken to perform a google search for different keywords you would use the Melody inputs to specify the keywords themselves and create a Melody objective function to take a particular keyword as input, perform and time the google search for that keyword and then return the time taken for that particular search.

API

A Melody objective function must contain a single input argument. Melody will call the objective function with particular input values from the inputs specified by the user and will expect results to be provided when the objective function completes.

The input argument is a Python list containing one or more dictionaries, each containing a key/value pair. The number of dictionary entries will correspond to the number of input objects specified by the user. Each dictionary will contain the name of the input as the key and one of its specified input options as the value.

Results are returned as two arguments. The first argument is a boolean value indicating whether the objective function was successful or not. For example, the a code might not compile, or the results might be incorrect. In this case `False` should be returned.

The second argument returns the results that the user would like to be optimised and/or evaluated for the objective function. For example, this might be the time a code took to run. The format of the second argument should be a dictionary of key/value pairs, but the format is not currently enforced.

```
def function(input):
    return success, result
```

Example

A simple example should help explain the API described in the previous section.

```
>>> from melody.inputs import Switch
>>> inputs = [Switch(name="option1", off="dark", on="light")]
>>> def function(input):
        print "function {0}".format(str(input))
        return True, {"value": 10}
>>> from melody.search import BruteForce
>>> from melody.main import Melody
>>> melody = Melody(inputs=inputs, function=function,
                    method=BruteForce)
>>> melody.search()
function [{'option1': 'dark'}]
[{'option1': 'dark'}] True {'value': 10}
function [{'option1': 'light'}]
[{'option1': 'light'}] True {'value': 10}
```

In the above example we have a single input object which can take two values (either `"dark"` or `"light"`. We use the BruteForce method (see Section *BruteForce*) so the objective function will be called twice, once for each value. As the objective function prints out the input values it can be seen that it is called twice and that the input is a list containing a single dictionary (as there is only one input object) and that dictionary contains a single key, the name given to the input object (`"option1"`) and a value which is one of the options provided in the input object (either `"dark"` or `"light"`). The objective function then returns `True` as it is always successful and a dictionary containing a key/value pair with a fixed value (`10`) that (in a useful objective function) would be used to indicate how the objective function performed. By default melody prints out the particular inputs passed to the objective function and the results provided by the objective function. These values can also be seen in the output from the example.

More examples of Melody objective functions can be found in the Melody examples directory.

Support

As explained earlier it is up to the user to write a Melody objective function. A typical objective function might

1. Take the input values for the function call and write those into appropriate input files e.g. a make include file
2. Compile a code using appropriate input files (e.g. a Makefile)
3. Check that the code compiled OK. If not return False.
4. Run a code based on appropriate input files (e.g. Gromacs config files)
5. Check that the code ran OK and gives valid answers. If not return False
6. Extract performance information from the run.
7. Return from the function call specifying success (True) for the job and providing the performance information itself (e.g. time taken).

For example:

```
def function(input):
    ''' user written objective function '''
    # use input to set compiler flag in Makefile
    # build the code with the Makefile
    # check it built OK. If not return False
    # run the code
```

```
    # check it ran OK. If not return False
    # extract the timing results
    return success, time
```

As many Melody objective functions are likely to follow a similar path to the one described above it is expected that a set of utility routines can be built up to support the process.

At this point one utility is provided. This utility is useful when setting up configuration files from the input data supplied to the objective function.

The utility takes the inputs to the objective function (or another equivalent data-structure created by the user) and matches any keys in the data-structure with keys within a template (using jinja2) replacing any matching key with its corresponding value.

For example

```
> cat template.txt
Hello {{name}}.
> python
>>> from melody.inputs import create_input
>>> input = [{"name": "fred"}]
>>> result = create_input(input, "template.txt",
                          template_location=".")
>>> print result
Hello fred.
```

Examples of the `create_input` function being used in Melody objective functions can be found in the Melody examples directory.

CHAPTER 5

Method

Melody methods determine how the search/optimisation space is going to be traversed. In the first instance melody is being used to search the optimisation space rather than optimise over it.

Methods

Melody currently only supports a single method called `BruteForce`.

BruteForce

The BruteForce method iterates over all possible combinations specified in the inputs irrespective of the return values. Thus it is a parameter search method rather than an optimisation method. It has proven useful to investigate relatively small optimisation-space landscapes if they are not known. For example, the performance of a code for a set of input parameters.

For example, if we specify three input objects, the associated function will be called for all combinations of their values.

```
>>> from melody.inputs import Fixed, Switch, IntRange
>>> inputs = [Fixed(name="opt1", value="grey"),
              Switch(name="opt2", off="dark", on="light"),
              IntRange("opt3", low=1, high=3, step=1)]
>>> def function(input):
        print "function {0}".format(str(input))
        return True, {"value": 10}
>>> from melody.search import BruteForce
>>> from melody.main import Melody
>>> melody = Melody(inputs=inputs, function=function,
                    method=BruteForce)
>>> melody.search()
function [{'opt1': 'grey'}, {'opt2': 'dark'}, {'opt3': 1}]
[{'opt1': 'grey'}, {'opt2': 'dark'}, {'opt3': 1}] True {'value': 10}
```

```
function [{'opt1': 'grey'}, {'opt2': 'dark'}, {'opt3': 2}]
[{'opt1': 'grey'}, {'opt2': 'dark'}, {'opt3': 2}] True {'value': 10}
function [{'opt1': 'grey'}, {'opt2': 'light'}, {'opt3': 1}]
[{'opt1': 'grey'}, {'opt2': 'light'}, {'opt3': 1}] True {'value': 10}
function [{'opt1': 'grey'}, {'opt2': 'light'}, {'opt3': 2}]
[{'opt1': 'grey'}, {'opt2': 'light'}, {'opt3': 2}] True {'value': 10}
```

In the above example, the first input object has 1 option, the second 2 options and the third 2 options. Therefore for a brute force combination one would expect to have 1*2*2 combinations in total equaling 4. As you can see, 4 options are output.

Extending

The user can create new Melody methods if they so wish. All methods should inherit from the `SearchMethod` base class.

Note: Please ignore the state argument and method in the `SearchMethod` class. These are not used at the moment and are placeholders for future developments.

The user can subclass the `SearchMethod` base class. The `run` method must be implemented as this is called by the `Melody` class (see the *Melody* section). The `run` method should take all of the supplied `inputs` and call the function appropriately.

In the example below an illustrative `PrintInputs` class is created which simply prints out the input objects supplied (it does not call the function)

```
>>> from melody.inputs import Fixed, Switch, IntRange
>>> inputs = [Fixed(name="option1", value="grey"),
              Switch(name="option2", off="dark", on="light"),
              IntRange("option3", low=1, high=3, step=1)]
>>> def function(input):
        return True, {"value": 10}
>>> from melody.search import SearchMethod
>>> class PrintInputs(SearchMethod):
        ''' example searchmethod subclass '''
        def __init__(self, inputs, function, state=None):
            SearchMethod.__init__(self, inputs, function, state)
        def run(self):
            ''' example run method '''
            for input in inputs:
                print input
>>> from melody.main import Melody
>>> melody = Melody(inputs=inputs, function=function,
                    method=PrintInputs)
>>> melody.search()
<melody.inputs.Fixed object at 0x7f7c1136a490>
<melody.inputs.Switch object at 0x7f7c1136a650>
<melody.inputs.IntRange object at 0x7f7c1136a990>
```