# RailGun Documentation

*Release 0.1.9.dev3*

**Takafumi Arakaki**

**Mar 23, 2017**

# Contents

Write less code in *C*

```c
typedef struct linearode_{
  int num_d, num_s;
  double dt;
  double **a;
  double **x;
} LinearODE;

int LinearODE_run(LinearODE *self)
{
  int s, d1, d2;
  for (s = 1; s < self->num_s; ++s){
    for (d1 = 0; d1 < self->num_d; ++d1){
      self->x[s][d1] = self->x[s-1][d1];
      for (d2 = 0; d2 < self->num_d;␣
↪++d2){
        self->x[s][d1] += self->dt * \
          self->a[d1][d2] * self->x[s-
↪1][d2];
      }
    }
  }
  return 0;
}
```

*... and in Python*!

```python
from railgun import SimObject, relpath


class LinearODE(SimObject):
    _clibname_ = 'liblode.so'
    _clibdir_ = relpath('.', __file__)
    _cmembers_ = [
        'num_d',
        'num_s = 10000',
        'double dt = 0.001',
        'double a[d][d]',
        'double x[s][d]',
        ]
    _cfuncs_ = ["x run()"]

lode = LinearODE(num_d=2)
lode.a = [[0, 1], [-1, 0]]
lode.x[0] = [1, 0]
x = lode.run()

import pylab
pylab.plot(x)
pylab.show()
```

The above pair of C and python code is very short but yet *complete* example!

If you want to write fast program for numerical simulations, you will always end up with writing it in C (or C++ or FORTRAN, these low-level languages). Although we have great python packages (not only) for numerical simulation such as Numpy/Scipy, Pyrex, Cython, Psyco and so on, sometimes these are not enough. On the other hand, writing code in C is stressful, especially things like allocating memory and read/write data which do not require speed so much. So, next thing you want is to call C function from python and let python do all these stuff (which python is good at!). Using ctypes, this is very easy.

On top of what ctypes provides, RailGun adds more features to kill boilerplate that you normally need for simulation coding.

For example, when you write simulation code, you may face situation like this many times:

> I am accessing array like `x[i][j]` and `y[j][k]`, so I want the second axis of the array `x` and the first axis of the array `y` to be of the same length.

RailGun solves this problem by keeping shape of all arrays to be consistent. Memory allocation for these arrays is done automatically.

RailGun also provides some value check before passing it to C function. For example, you may want to pass an index of some array to C function. When you do that, you need to check if the index is in a certain range, to avoid segmentation fault. RailGun provides a short hand notation to check that automatically. Also, you can wrap C function to put any kind of complex value check and pre/post-processing.

With these features and other useful utilities provided by RailGun, you can really focus on guts of computation in C code.

Installation:

```
pip install railgun   # using pip
easy_install railgun  # using setuptools (if you must)
```

Requirements:

- Numpy
- six
- (matplotlib for sample code)

Contents:

CHAPTER 1

# RailGun Tutorial

## How to write C program for RailGun

RailGun requires the following constraints in C library to be loaded.

### Constraints on data structure (C's *struct*)

This is an example of a *struct*:

```
typedef struct linearode_{
  int num_d, num_s;
  double dt;
  double **a;
  double **x;
} LinearODE;
```

Here, variable named as *num_d* has special meaning in RailGun. This is the size of array along index *d*.

You can have temporary variables in your C code, but if you want to get or set C variable from python, you must add that variable to the *struct*.

### Constraints on C functions

If you want to use some functions from python, the function must take a pointer of the *struct* as its first argument, like this:

```
int NameOfStruct_name_of_function(NameOfStruct *self, ...)
```

If this function returns non-zero value, RailGun raises an error.

Here is an example:

```
int LinearODE_run(LinearODE *self, int s_end)
{
  int s, d1, d2;
  for (s = 1; s < s_end; ++s){
    for (d1 = 0; d1 < self->num_d; ++d1){
      self->x[s][d1] = self->x[s-1][d1];
      for (d2 = 0; d2 < self->num_d; ++d2){
        self->x[s][d1] += self->dt * self->a[d1][d2] * self->x[s-1][d2];
      }
    }
  }
  return 0;
}
```

**Note:** You don't need to check if *s_end* above is in the range of *[1, self->num_s]*. We will see how you can leave it to RailGun.

## How to use your C functions from python

All you need to import from RailGun is these two:

```
from railgun import SimObject, relpath
```

To load your c function above, all you need to do is define a class which inherits *railgun.SimObject*:

```
class LinearODE(SimObject):
    _clibname_ = 'liblode.so'
    _clibdir_ = relpath('.', __file__)
    _cmembers_ = [
        'num_d',
        'num_s = 10000'
        'double dt = 0.001',
        'double a[d][d]',
        'double x[s][d]',
        ]
    _cfuncs_ = ["x run(int s_end=num_s)"]
```

**Name of the class** should be same as name of the C struct.

***_clibname_*: a string** Name of your C shared library.

***_clibdir_*: a string** Path of the directory where your C library are. If you want to specify relative path from where this python module file are, you can use `relpath('relative/path', __file__)`.

***_cmembers_*: a list of string** This is the definitions of your C variables. **The order must be the same as in the C struct**. For the member named *num_** you can omit `int`. You can set the default value as `double dt = 0.001`. Indices of C array have meaning. Size of the first axis of `x[s][d]` is `num_s`, and the second is `num_d`.

***_cfuncs_*: a list of string** This is the definitions of your C functions which take the form *ret func_name(arg, ...)* where

- *ret* is the returned value of the function which is a name of C *struct* member. You can leave it empty.
- *func_name* is the name of the C function(s). You don't need to write the name of the *struct*. The name of the *struct* will be automatically added. You can specify several functions using special notations.

- *arg* is the definition of the arguments for the C function. This is essentially same as function declaration of C, but with special features. One of the feature is default value. You can specify default value like python: `int s_end=num_s` or `int s_start=0`.

### Loading several C functions at once: `func_{key|c1,c2}`-notation (choices)

If you have several C functions of *same type* such as:

```
int NameOfStruct_func_method1(NameOfStruct *self, int a, int b)
int NameOfStruct_func_method2(NameOfStruct *self, int a, int b)
int NameOfStruct_func_method3(NameOfStruct *self, int a, int b)
```

You can load all these functions like this:

```
'func_{meth | method1, method2, method3}(int a, int b)'
```

Generated python function will be like this:

```
NameOfStruct.func(a, b, meth='method1')
```

as you see, you can specify "method" by option of the python function. The `func_{key|c1,c2}`-notation is called "choice set".

### Automatically check argument consistency

You can use name of the index as a type of argument like this:

```
"run(s end)"
```

so that *end* is always in the range *[0, num_s)*.

If *end* is an "upper bound" of the index, you want it to be in the range *(0, num_s]*. You can specify this with < like this:

```
"run(s< end)"
```

## Using generated python class

An instance of the simulation class can be created like any other Python classes. Note that *num_** arguments are required:

```
lode = LinearODE(num_d=2)
```

If you specified the default values for *num_**, you can make an instance without passing its value.

Once you create an instance, you can change C variables in various ways:

```
lode.a = [[0, 1], [-1, 0]]
lode.x[0] = [1, 0]
lode.setv(a_0_0=-0.5)
```

Note that *lode.setv(a_0_0=-0.5)* and *lode.a[0,0] = -0.5* are the same.

Calling function is easy. Number of arguments are the number of arguments of C function puls number of the "choice set". The first arguments are used for C function and then the last arguments are used for "choice set". You can also use keyword-style arguments:

```
lode.run()
lode.run(10)
lode.run(s_end=10)
```

# CHAPTER 2

## SimObject — A class loads everything you need from C shared library

**class** yourcode.**YourSimObject**

To load your C shared library, define a class inheriting *railgun.SimObject*. You need to define four attribute: *_clibname_*, *_clibdir_*, *_cmembers_* and *_cfuncs_*.

---

**Note:** In this document, *yourcode.YourSimObject* means the class *you need to define*. This class is not in RailGun.

---

Example:

```
class YourSimObject(SimObject):
    _clibname_ = 'name_of_shared_library.so'
    _clibdir_ = 'path/to/shared/library'

    _cmembers_ = [
        'num_i',
        'num_j',
        'int scalar',
        'int vector[i]',
        'int matrix[i][j]',
        ...
        ]

    _cfuncs_ = [
        'name_of_c_function',
        ...
        ]

    def __init__(self, num_i, num_j, **kwds):
        SimObject.__init__(self, num_i=num_i, num_j=num_j, **kwds)

    def some_function(self, ...):
        ...
```

- You can override `railgun.SimObject.__init__()`.
- You can define additional python methods or members. But be careful with name: *railgun.SimObject* will overwrite the methods or members of your class with name in *_cmembers_* and *_cfuncs_*.

**_clibname_**
Name of your C shared library.

**_clibdir_**
Path to the directory where your C library locates. If you want to specify relative path from where this python module file are, you can use `relpath()`.

**_cmembers_**
This is a list of the definitions of C variables with the following syntax:

```
[CDT] VAR_NAME[INDEX] [= DEFAULT]
```

**VAR_NAME: name of variable** This let you access the C variable by *obj.VAR_NAME*.

Starting the name of the member with `num_` defines an index whose name is what comes after this. For example, `num_i` defines index `i`. Value of `num_i` is the size of array(s) along the index `i`. For the member named `num_*`, you can omit **CDT** (`int`).

**CDT: C Data Type, (optional if VAR_NAME starts with num_)** Choose CDT from the list in *Relationships between C Data Type (CDT), numpy dtype and ctypes*.

**INDEX: index, optional** If the variable is an array, **INDEX** should be specified. For an array with shape `num_i1 x num_i2 x ... x num_iN`, **INDEX** should be `[i1][i2]...[iN]` or `[i1,i2,...,iN]`.

**`[i1][i2]...[iN]`: multidimensional array** You can access `a[i][j]` as `self->a[i][j]` in C code. This array data structure is called "Iliffe vector" or "display". Strictly speaking, this is not equivalent to multidimensional array, but you can use as if it is.

**`[i1,i2,...,iN]`: flattened array** You can access `a[i][j]` as `self->a[i * self->num_j + j]` in C code. Specifying correct index in C code is up to you. It is recommended to use macro or inline function.

**DEFAULT: a number, optional** A default number for the variable. If **VAR_NAME** is an array, it will be filled with this value when it is created.

> **Warning:** The order and number of the variables in *_cmembers_* must be the same as in the C struct.

Example:

```
_cmembers_ = [
    'num_i', 'num_j', 'num_k',
    'int int_scalar',
    'int int_vector1[i]',
    'int int_vector2[j] = 0',
    'int int_matrix[i][j]',
    'double double_scalar = 0.1',
    'double double_vector[k] = 18.2',
    'double double_matrix[k][i] = -4.1',
    ]
```

See also: *railgun.cmems()*

**_cfuncs_**

This is a list of the definitions of C functions with the following syntax:

```
[RETURN_VAR] FUNC_NAME(ARG, [ARG[, ...]])
```

**FUNC_NAME: string** Name of C function to be loaded. You don't need to write the name of the *struct*. The name of the *struct* will be automatically prepended.

See also: *Loading several C functions at once: func_{key|c1,c2}-notation (choices)*.

**RETURN_VAR: string, optional** Name from C struct members. If specified, python wrapper function named **FUNC_NAME** returns value of **RETURN_VAR**.

**ARG:** Argument of C function, specified by the following syntax:

```
CDT_OR_INDEX ARG_NAME [= DEFAULT]
```

**CDT_OR_INDEX: string** C Data Type or index. If index `i` (`i<`) is used here, error will be rasied if the argument *x* does not satisfy *0* <= *x* < *num_i* (*0* < *x* <= *num_i*).

**ARG_NAME: string** Name of the argument.

**DEFAULT: a number or member of C struct, optional** Default value for the argument.

You don't need to write `self` which will be automatically passed as the first argument of C function.

Example:

```
_cfuncs_ = [
    "func_spam()",
    "bar func_foo()",
    "func_with_args(int a, double b, i start=0, i< end=num_i)",
    "func_{key | c1, c2, c3}()",
    ]
```

See also: *Constraints on C functions*

**_cstructname_**

This is optional. This is used to specify the name of C struct explicitly:

```
class CStructName(SimObject):  # 'CStructName' is name of c-struct
    ...

class OtherNameForPyClass(SimObject):
    ...
    _cstructname_ = 'CStructName'  # this is name of c-struct
```

**_cfuncprefix_**

This is optional. This is used to specify the prefix of C functions explicitly (default is name of C Struct + _):

```
class YourSimObject(SimObject):
    ...
    _cfuncprefix_ = 'MyPrefix'
    _cfuncs_ = [
        "FuncName()",  # 'MyPrefixFuncName' will be loaded
        ...
```

```
        ]

class YourSimObject(SimObject):
    ...
    _cfuncprefix_ = ''
    _cfuncs_ = [
        "FuncName()",  # 'FuncName' will be loaded
        ...
        ]
```

**_cmemsubsets_**
dict of dict of list, optional.

It defines the subset of C functions and struct variables to be accessible. It must be of the following format:

```
{'<SUBSET_KEY_1>': {
    'funcs': ['<FUNCTION_1>', '<FUNCTION_2>', ...],
    'members': ['<MEMBER_1>', '<MEMBER_2>', ...],
    'default': True,  # optional (default is False)
    },
 '<SUBSET_KEY_2>': {...},
 ...}
```

This is useful when some subset of functions needs some subset of struct members. For example, when in "debugging mode", you may want to record all temporal variables. However, allocating temporal variables can be wasteful if you are not debugging. Using *_cmemsubsets_*, you can allocate temporal variables when in the debugging mode and make sure that the functions that requires temporal variables are callable only in the debugging mode. It helps you to avoid segmentation fault due to accessing invalid pointer. *_cmemsubsets_* can be thought as machinery for "access levels".

**SUBSET_KEY** [string] You can pass boolean argument named `_cmemsubsets_SUBSET_KEY` to `railgun.SimObject.__init__()` to enable or disable the corresponding subset.

**FUNCTION** [list of strings] These functions are accessible from Python when the corresponding subset is enabled. You can use short-hand notation `'func_{a, b, c}'` to specify functions `'func_a'`, `'func_b'` and `'func_c'`.

**MEMBER** [list of strings] These struct members are allocated and accessible from Python when the corresponding subset is enabled.

**DEFAULT** [bool, optional] It is *False* when not specified, meaning that the C members in this subset is not accessible.

Here is an example:

```
class YourSimObject(SimObject):

    _cmembers_ = [
        # ...
        'temp[j][k]',
        ]
    _cfuncs_ = [
        'run_{ mode | normal, debug }()',
        ]
    _cmemsubsets_ = {
        'debug': {
            'funcs': ['run_debug'],
            'members': ['temp'],
            },
```

```
        }

# Run simulation in debugging mode:
sim = YourSimObject(..., _cmemsubsets_debug=True)
sim.run(mode='debug')
```

**_cwrap_C_FUNC_NAME** (*func*)

This is optional. If you want to wrap C function `C_FUNC_NAME`, define this wrapper function.

Example:

```
class YourSimObject(SimObject):

    _clibname_ = '...'
    _clibdir_ = '...'
    _cmembers_ = [
        'num_i',
        'int vec[i]',
        ]
    _cfuncs_ = [
        'your_c_function',
        ]

    def _cwrap_your_c_function(old_c_function):
        def your_c_function(self, *args, **kwds):
            old_c_function(self, *args, **kwds)
            return self.vec[:]  # return copy
        return your_c_function
```

After *your_c_function* is loaded from C library, your wrapper function will be called like this:

```
your_c_function = _cwrap_your_c_function(your_c_function)
```

**_cerrors_**

This is optional. When C function returns non-zero value, RailGun raises error which just tells the value returned (error code). To make the error message readable, or to handle the error better, you may want to use this attribute.

If C function returns the non-zero value `error_code`, and it is found `_cerrors_`, RailGun will raise the error `_cerrors_[error_code]`.

Examples:

```
class YourSimObject(SimObject):

    _clibname_ = '...'
    _clibdir_ = '...'
    _cmembers_ = [...]
    _cfuncs_ = [...]

    class YourExceptionClass(Exception):
        pass

    _cerrors_ = {
        # set exception
        1: RuntimeError('error code 1 is raised'),
        # you can use your own exception class
        2: YourExceptionClass('your error message'),
        }
```

New in version 0.1.7.

**class** `railgun.`**`SimObject`**(***kwds*)

Base class for wrapping simulator code written in C.

**`cinfo`**

Instance of `CInfo`.

**static `array_alias`**(*alias*)

**Parameters** **`alias`** (*str*) – string such as `'a_1_2'`, which is an alias of `a[1][2]`.

**Return type** 2-tuple or None

**Returns** Tuple of strings `(name, index)`. `name` is a name of the C member (e.g., `'a'`) specified by *`_cmembers_`*. `index` is a index of ints (e.g., `(1, 2)`).

**`setv`**(***kwds*)

This is used for setting values of C struct members or any other Python attributes.

The following two lines have same effects:

```
obj.setv(scalar=1, array=[1, 2, 3])
obj.scalar = 1; obj.array = [1, 2, 3]
```

You can use alias for elements of array. The following lines have same effect:

```
obj.setv(var_0_1=1)
obj.var[0][1] = 1
```

**`getv`**(**args*)

Get the C variable by specifying the name or any other Python attributes.

The following lines have same effect:

```
var = obj.var
var = obj.getv('var')
```

This is useful when you want to load multiple variables to local variable at once. The Following lines have same effect:

```
(a, b, c) = (obj.a, obj.b, obj.c)
(a, b, c) = obj.getv('a', 'b', 'c')
(a, c, c) = obj.getv('a, b, c')
```

**`num`**(**args*)

Get the size along index. The Following lines have same effect:

```
num_i = obj.num_i
num_i = obj.num('i')
```

You can specify multiple indices. The Following lines have same effect:

```
(num_i, num_j, num_k) = (obj.num_i, obj.num_j, obj.num_k)
(num_i, num_j, num_k) = obj.num('i', 'j', 'k')
(num_i, num_j, num_k) = obj.num('i, j, k')
```

**`reallocate`**(***nums*)

Reallocate arrays consistently.

Usage:

```
>>> obj.reallocate(i=10, j=20)
>>> obj.num_i
10
>>> obj.num_j
20
```

## Relationships between C Data Type (CDT), numpy dtype and ctypes

To specify C-language type of C struct members and C function arguments, the following C Data Types (**CDTs**) are available.

| CDT | C-language type | numpy dtype | ctypes |
|---|---|---|---|
| `char` | `char` | *character* | *c_char* |
| `short` | `short` | *short* | *c_short* |
| `ushort` | `unsigned short` | *ushort* | *c_ushort* |
| `int` | `int` | *int32* | *c_int* |
| `uint` | `unsigned int` | *uint32* | *c_uint* |
| `long` | `long` | *int32* or *int64* | *c_long* |
| `ulong` | `unsigned long` | *uint32* or *uint64* | *c_ulong* |
| `longlong` | `long long` | *longlong* | *c_longlong* |
| `ulonglong` | `unsigned long long` | *ulonglong* | *c_ulonglong* |
| `float` | `float` | *float32* | *c_float* |
| `double` | `double` | *float* | *c_double* |
| `longdouble` | `long double` | *longdouble* | *c_longdouble* |
| `bool` | `bool` | *bool* | *c_bool* |

**Note:** Numpy dtypes corresponding to CDTs `long` and `ulong` are chosen based on the variable returned by `platform.architecture()`.

# CHAPTER 3

# Utility functions

railgun.**relpath**(*relative_path*, *__file__*)

Get relative path.

Let you have source tree like this:

```
+- your_module.py
`- ext/
   +- build/
   |  `- your_clib.so
   +- src/
   |  `- your_clib.c
   `- Makefile
```

To load `your_clib.so` from `your_module.py`, you can use *`relpath()`* in your class definition in `your_module.py` like this:

```
_clibname_ = 'your_clib.so'
_clibdir_ = relpath('ext/build', __file__)
```

Note that *__file__* is a special attribute of python module which is the pathname of the file from which the module was loaded.

railgun.**cmems**(*cdt*, **args*)

Generate `SimObject._cmembers_` for same C Data Type (CDT) easily

Usage:

```
>>> cmems('int', 'a', 'b', 'c')
['int a', 'int b', 'int c']
>>> cmems('int', 'a, b, c')
['int a', 'int b', 'int c']
>>> cmems('int', 'a[i]', 'b[i][j]') + cmems('double', 'x[i]', 'y[i][j]')
['int a[i]', 'int b[i][j]', 'double x[i]', 'double y[i][j]']
```

CHAPTER 4

# Samples

## Linear Ordinary Differential Equation

### Output

### Python code

```
from railgun import SimObject, relpath

class LinearODE(SimObject):
    """
    Solve D-dimensional linear ordinary differential equations

    Equation::

        dX/dt(t) = A X(t)
        X: D-dimensional vector
        A: DxD matrix

    """

    _clibname_ = 'liblode.so'  # name of shared library
    _clibdir_ = relpath('.', __file__)  # library directory
    _cmembers_ = [  # declaring members of struct
        'num_d',  # num_* as size of array (no need to write `int`)
        'num_s = 10000',  # setting default value
        'double dt = 0.001',
        'double a[d][d]',  # num_d x num_d array
        'double x[s][d]',  # num_s x num_d array
        ]
    _cfuncs_ = [  # declaring functions to load
        "x run(s< s_end=num_s)"
        # argument `s_end` has index `s` type and default is `num_s`
```

```
        # '<' means it is upper bound of the index so the range is [1, num_s]
        # this function returns member x
        ]


lode = LinearODE(num_d=2)  # set num_d
lode.x[0] = [1, 0]  # access c-member "VAR" via lode.VAR
lode.a = [[0, 1], [-1, 0]]
x1 = lode.run().copy()
lode.setv(a_0_0=-0.5)  # set lode.a[i][j]=v via lode.set(a_'i'_'j'=v)
x2 = lode.run().copy()

import pylab
for (i, x) in enumerate([x1, x2]):
    pylab.subplot(2, 2, 1 + i)
    pylab.plot(x[:,0])
    pylab.plot(x[:,1])
    pylab.subplot(2, 2, 3 + i)
    pylab.plot(x[:,0], x[:,1])
pylab.show()
```

## C code

```
typedef struct linearode_{
  int num_d, num_s;
  double dt;
  double **a;
  double **x;
} LinearODE;

int LinearODE_run(LinearODE *self, int s_end)
{
  int s, d1, d2;
  for (s = 1; s < s_end; ++s){
    for (d1 = 0; d1 < self->num_d; ++d1){
      self->x[s][d1] = self->x[s-1][d1];
      for (d2 = 0; d2 < self->num_d; ++d2){
        self->x[s][d1] += self->dt * self->a[d1][d2] * self->x[s-1][d2];
      }
    }
  }
  return 0;
}
```

## Using `SimObject._cstructname_`

### Python code

```
from railgun import SimObject, relpath

class LinearOrdinaryDifferentialEquation(SimObject):  # != LinearODE
    """
    Solve D-dimensional linear ordinary differential equations
```

```
    Equation::

        dX/dt(t) = A X(t)
        X: D-dimensional vector
        A: DxD matrix

    """

    _clibname_ = 'liblode.so'  # name of shared library
    _clibdir_ = relpath('.', __file__)  # library directory
    _cstructname_ = 'LinearODE'  # specify the C struct name
    _cmembers_ = [  # declaring members of struct
        'num_d',  # num_* as size of array (no need to write `int`)
        'num_s = 10000',  # setting default value
        'double dt = 0.001',
        'double a[d][d]',  # num_d x num_d array
        'double x[s][d]',  # num_s x num_d array
        ]
    _cfuncs_ = [  # declaring functions to load
        "x run(s< s_end=num_s)"
        # argument `s_end` has index `s` type and default is `num_s`
        # '<' means it is upper bound of the index so the range is [1, num_s]
        # this function returns member x
        ]


lode = LinearOrdinaryDifferentialEquation(num_d=2)  # set num_d
lode.x[0] = [1, 0]  # access c-member "VAR" via lode.VAR
lode.a = [[0, 1], [-1, 0]]
x1 = lode.run().copy()
lode.setv(a_0_0=-0.5)  # set lode.a[i][j]=v via lode.set(a_'i'_'j'=v)
x2 = lode.run().copy()

import pylab
for (i, x) in enumerate([x1, x2]):
    pylab.subplot(2, 2, 1 + i)
    pylab.plot(x[:,0])
    pylab.plot(x[:,1])
    pylab.subplot(2, 2, 3 + i)
    pylab.plot(x[:,0], x[:,1])
pylab.show()
```

What is the difference from the normal version? (... just three lines!):

```
--- lode_cstructname.py        2011-04-17 15:39:05.942934998 +0900
+++ lode.py        2011-04-16 21:53:58.707407999 +0900
@@ -1,6 +1,6 @@
 from railgun import SimObject, relpath

-class LinearOrdinaryDifferentialEquation(SimObject):  # != LinearODE
+class LinearODE(SimObject):
     """
     Solve D-dimensional linear ordinary differential equations

@@ -14,7 +14,6 @@

     _clibname_ = 'liblode.so'  # name of shared library
```

```
     _clibdir_ = relpath('.', __file__)  # library directory
-    _cstructname_ = 'LinearODE'  # specify the C struct name
     _cmembers_ = [  # declaring members of struct
         'num_d',  # num_* as size of array (no need to write `int`)
         'num_s = 10000',  # setting default value
@@ -30,7 +29,7 @@
         ]


-lode = LinearOrdinaryDifferentialEquation(num_d=2)  # set num_d
+lode = LinearODE(num_d=2)  # set num_d
 lode.x[0] = [1, 0]  # access c-member "VAR" via lode.VAR
 lode.a = [[0, 1], [-1, 0]]
 x1 = lode.run().copy()
```

## Using `SimObject._cfuncprefix_`

How to omit the name of the struct in the C functions.

### Python code

Please notice that `_cfuncprefix_ = ''` is added to omit the name of the struct name.

```
from railgun import SimObject, relpath

class LinearODE(SimObject):
    """
    Solve D-dimensional linear ordinary differential equations

    Equation::

        dX/dt(t) = A X(t)
        X: D-dimensional vector
        A: DxD matrix

    """

    _clibname_ = 'liblode_cfuncprefix.so'  # name of shared library
    _clibdir_ = relpath('.', __file__)  # library directory
    _cfuncprefix_ = ''  # no prefix for C functions!
    _cmembers_ = [  # declaring members of struct
        'num_d',  # num_* as size of array (no need to write `int`)
        'num_s = 10000',  # setting default value
        'double dt = 0.001',
        'double a[d][d]',  # num_d x num_d array
        'double x[s][d]',  # num_s x num_d array
        ]
    _cfuncs_ = [  # declaring functions to load
        "x run(s< s_end=num_s)"
        # argument `s_end` has index `s` type and default is `num_s`
        # '<' means it is upper bound of the index so the range is [1, num_s]
        # this function returns member x
        ]
```

```
lode = LinearODE(num_d=2)  # set num_d
lode.x[0] = [1, 0]  # access c-member "VAR" via lode.VAR
lode.a = [[0, 1], [-1, 0]]
x1 = lode.run().copy()
lode.setv(a_0_0=-0.5)  # set lode.a[i][j]=v via lode.set(a_'i'_'j'=v)
x2 = lode.run().copy()

import pylab
for (i, x) in enumerate([x1, x2]):
    pylab.subplot(2, 2, 1 + i)
    pylab.plot(x[:,0])
    pylab.plot(x[:,1])
    pylab.subplot(2, 2, 3 + i)
    pylab.plot(x[:,0], x[:,1])
pylab.show()
```

### C code

The function is simply `run` without the prefix now.

```
typedef struct linearode_{
  int num_d, num_s;
  double dt;
  double **a;
  double **x;
} LinearODE;

int run(LinearODE *self, int s_end)
{
  int s, d1, d2;
  for (s = 1; s < s_end; ++s){
    for (d1 = 0; d1 < self->num_d; ++d1){
      self->x[s][d1] = self->x[s-1][d1];
      for (d2 = 0; d2 < self->num_d; ++d2){
        self->x[s][d1] += self->dt * self->a[d1][d2] * self->x[s-1][d2];
      }
    }
  }
  return 0;
}
```

## Runge-Kutta method (usage of `_cmemsubsets_`)

If you are using a lot of intermediate variables and want to check these variables for debugging, you can use *YourSimObject._cmemsubsets_* to allocate memory only when the flag is on.

### Python code (`lode_rk4.py`)

```
from railgun import SimObject, relpath, cm


class LinearODERK4(SimObject):
```

```
    """
    Solve D-dimensional linear ODE using the Runge-Kutta method

    Equation::

        dX/dt(t) = A X(t)
        X: D-dimensional vector
        A: DxD matrix

    """

    _clibname_ = 'liblode_rk4.so'  # name of shared library
    _clibdir_ = relpath('.', __file__)  # library directory
    _cmembers_ = [  # declaring members of struct
        'num_d',  # num_* as size of array (no need to write `int`)
        'num_s',  # setting default value
        'double dt = 0.001',
        'double a[d][d]',  # num_d x num_d array
        'double x[s][d]',  # num_s x num_d array
    ] + (
        cm.double('k%s[d]' % s for s in '1234') +
        cm.double('x%s[d]' % s for s in '123') +
        cm.double('k%s_debug[s][d]' % s for s in '1234') +
        cm.double('x%s_debug[s][d]' % s for s in '123')
    )
    _cfuncs_ = ["x run_{mode | normal, debug}()"]
    _cmemsubsets_ = dict(
        debug=dict(
            funcs=['run_debug'],
            members=(['k%s_debug' % s for s in '1234'] +
                     ['x%s_debug' % s for s in '123'])),
        )

    def plot_x(self):
        import pylab
        t = pylab.arange(self.num_s) * self.dt

        for d in range(self.num_d):
            pylab.plot(t, self.x[:,d], label='x%s'% d)
        pylab.legend()

    def plot_k(self):
        import pylab
        t = pylab.arange(self.num_s) * self.dt
        klist = self.getv(*['k%s_debug' % s for s in '1234'])

        for d in range(self.num_d):
            for (i, k) in enumerate(klist):
                pylab.plot(t, k[:,d], label='k%s_%s'% (i, d))
        pylab.legend()
```

## C code (`lode_rk4.c`)

```
typedef struct linearoderk4_{
  int num_d, num_s;
  double dt;
```

```
  double **a, **x;
  double *k1, *k2, *k3, *k4, *x1, *x2, *x3;
  double **k1_debug, **k2_debug, **k3_debug, **k4_debug,
    **x1_debug, **x2_debug, **x3_debug;
} LinearODERK4;


void calc_f(double *y, double *x, double **a, int num_d)
{
  int d1, d2;

  for (d1 = 0; d1 < num_d; ++d1){
    y[d1] = 0;
    for (d2 = 0; d2 < num_d; ++d2){
      y[d1] += a[d1][d2] * x[d2];
    }
  }
}

void calc_xi(double *xi, double *x0, double *ki, double c, double dt,
             int num_d)
{
  int d;
  for (d = 0; d < num_d; ++d){
    xi[d] = x0[d] + dt * c * ki[d];
  }
}


void iterate_once(LinearODERK4 *self, int s)
{
  int d;
  double *x0 = self->x[s-1];

  calc_f(self->k1, x0, self->a, self->num_d);
  calc_xi(self->x1, x0, self->k1, 0.5, self->dt, self->num_d);
  calc_f(self->k2, self->x1, self->a, self->num_d);
  calc_xi(self->x2, x0, self->k2, 0.5, self->dt, self->num_d);
  calc_f(self->k3, self->x2, self->a, self->num_d);
  calc_xi(self->x3, x0, self->k3, 1, self->dt, self->num_d);
  calc_f(self->k4, self->x3, self->a, self->num_d);

  for (d = 0; d < self->num_d; ++d){
    self->x[s][d] = x0[d] + self->dt / 6 *
      (self->k1[d] + 2 * self->k2[d] + 2 * self->k3[d] + self->k4[d]);
  }
}


int LinearODERK4_run_normal(LinearODERK4 *self)
{
  int s;
  for (s = 1; s < self->num_s; ++s){
    iterate_once(self, s);
  }
  return 0;
}
```

```
int LinearODERK4_run_debug(LinearODERK4 *self)
{
  int s, d;
  for (s = 1; s < self->num_s; ++s){
    iterate_once(self, s);
    for (d = 0; d < self->num_d; ++d){
      self->k1_debug[s][d] = self->k1[d];
      self->k2_debug[s][d] = self->k2[d];
      self->k3_debug[s][d] = self->k3[d];
      self->k4_debug[s][d] = self->k4[d];
      self->x1_debug[s][d] = self->x1[d];
      self->x2_debug[s][d] = self->x2[d];
      self->x3_debug[s][d] = self->x3[d];
    }
  }
  return 0;
}
```

## Running on “normal mode”

```
from lode_rk4 import LinearODERK4

lode = LinearODERK4(num_s=10000, num_d=2)
lode.x[0] = [1, 0]  # access c-member "VAR" via lode.VAR
lode.a = [[-0.5, 1], [-1, 0]]
lode.run()  # mode='normal'

import pylab
lode.plot_x()
pylab.show()
```

## Running on “debug mode”

You can create “debug mode” instance by giving `_cmemsubsets_debug=True` to your class constructor.

```
from lode_rk4 import LinearODERK4

lode = LinearODERK4(num_s=10000, num_d=2, _cmemsubsets_debug=True)
lode.x[0] = [1, 0]  # access c-member "VAR" via lode.VAR
lode.a = [[-0.5, 1], [-1, 0]]
lode.run(mode='debug')

import pylab
lode.plot_k()
pylab.show()
```

## Reference

- Runge–Kutta methods - Wikipedia, the free encyclopedia

## Logistic map with additive noise (usage of `railgun.cmem()`)

You will need the file `gslctypes_rng.py` (see *Using RailGun with GSL (GNU Scientific Library)*).

### Python code

`logistic_map.py`:

```
from railgun import SimObject, relpath, cmem
import numpy
from gslctypes_rng import gsl_rng

class LogisticMap(SimObject):

    _clibname_ = 'liblogistic_map.so'
    _clibdir_ = relpath('.', __file__)
    _cmembers_ = [
        'num_i',
        'double xt[i]',
        'double mu',
        'double sigma',
        cmem(gsl_rng, 'rng'),
        ]
    _cfuncs_ = ["xt gene_seq()"]

    def __init__(self, num_i, seed=None, **kwds):
        SimObject.__init__(self, num_i=num_i, rng=gsl_rng(seed), **kwds)

    def _cwrap_gene_seq(old_gene_seq):
        def gene_seq(self, **kwds):
            # use the previous "last state" as initial state, but setv
            # may overwrite this by `xt_0=SOMEVAL`
            self.xt[0] = self.xt[-1]
            self.setv(**kwds)
            return old_gene_seq(self)
        return gene_seq

    def bifurcation_diagram(self, mu_start, mu_stop, mu_num, **kwds):
        mu_list = numpy.linspace(mu_start, mu_stop, mu_num)
        self.gene_seq(mu=mu_list[0], **kwds)  # through first steps away
        xt_list = [self.gene_seq(mu=mu).copy() for mu in mu_list]
        return (mu_list, xt_list)

    def plot_bifurcation_diagram(self, mu_start, mu_stop, mu_num, **kwds):
        import pylab
        bd = self.bifurcation_diagram(mu_start, mu_stop, mu_num, **kwds)
        ones = numpy.ones(self.num_i)

        for (mu, x) in zip(*bd):
            pylab.plot(ones * mu, x, 'ko', markersize=0.5)
        pylab.ylim(0, 1)
```

`bifurcation_diagram.py`:

```
from logistic_map import LogisticMap
import pylab
```

```
lmap = LogisticMap(1000)

pylab.figure(1)
lmap.plot_bifurcation_diagram(0, 4, 500, sigma=1e-5)
pylab.title(r'$\sigma=10^{-5}$')

pylab.figure(2)
lmap.plot_bifurcation_diagram(0, 4, 500, sigma=1e-3)
pylab.title(r'$\sigma=10^{-3}$')

pylab.figure(3)
lmap.plot_bifurcation_diagram(0, 4, 500, sigma=1e-2)
pylab.title(r'$\sigma=10^{-2}$')

pylab.show()
```

### C code

logistic_map.c:

```
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>

typedef struct logisticmap_{
  int num_i;
  double *xt;
  double mu;
  double sigma;
  gsl_rng *rng;
} LogisticMap;


int LogisticMap_gene_seq(LogisticMap *self)
{
  int i;
  double eta;
  for (i = 1; i < self->num_i; ++i){
    eta = gsl_ran_gaussian_ziggurat(self->rng, self->sigma);
    self->xt[i] = self->mu * self->xt[i-1] * (1 - self->xt[i-1]) + eta;
    if (self->xt[i] < 0){
      self->xt[i] = 0;
    }
  }
  return 0;
}
```

### Output

## Using RailGun with GSL (GNU Scientific Library)

Class `gslctypes_rng.cmem` defined in the following file (`gslctypes_rng.py`) is a class you can give to `railgun.cmem()`. See *Logistic map with additive noise (usage of railgun.cmem())* for usage.

```
import ctypes
from ctypes import (POINTER, c_char_p, c_size_t, c_int, c_long, c_ulong,
                    c_double, c_void_p)
from ctypes.util import find_library

from six import string_types as basestring


class _c_gsl_rng_type(ctypes.Structure):
    _fields_ = [('name', c_char_p),
                ('max', c_long),
                ('min', c_size_t),
                ('__set', c_void_p),
                ('__get', c_void_p),
                ('__get_double', c_void_p),
                ]
_c_gsl_rng_type_p = POINTER(_c_gsl_rng_type)


class _c_gsl_rng(ctypes.Structure):
    _fields_ = [('type', _c_gsl_rng_type_p),
                ('state', c_void_p)]
_c_gsl_rng_p = POINTER(_c_gsl_rng)


class _GSLFuncLoader(object):

    # see: http://code.activestate.com/recipes/576549-gsl-with-python3/
    gslcblas = ctypes.CDLL(find_library('gslcblas'), mode=ctypes.RTLD_GLOBAL)
    gsl = ctypes.CDLL(find_library('gsl'))

    def _load_1(self, name, argtypes=None, restype=None):
        func = getattr(self.gsl, name)
        if argtypes is not None:
            func.argtypes = argtypes
        if restype is not None:
            func.restype = restype
        setattr(self, name, func)
        return func

    def _load(self, name, argtypes=None, restype=None):
        if isinstance(name, basestring):
            return self._load_1(name, argtypes, restype)
        else:
            try:
                return [self._load_1(n, argtypes, restype) for n in name]
            except TypeError:
                raise ValueError('name=%r should be a string or iterative '
                                 'of string' % name)


func = _GSLFuncLoader()
func._load('gsl_strerror', [c_int], c_char_p)
func._load('gsl_rng_alloc', [_c_gsl_rng_type_p], _c_gsl_rng_p)
func._load('gsl_rng_set', [_c_gsl_rng_p, c_ulong])
func._load('gsl_rng_free', [_c_gsl_rng_p])
func._load('gsl_rng_types_setup',
           restype=c_void_p)  # POINTER(_c_gsl_rng_p)
```

```
func._load(['gsl_ran_gaussian',
            'gsl_ran_gaussian_ziggurat',
            'gsl_ran_gaussian_ratio_method'],
           [_c_gsl_rng_p, c_double],
           c_double)


gsl_strerror = func.gsl_strerror


def _get_gsl_rng_type_p_dict():
    """
    Get all ``gsl_rng_type`` as dict which has pointer to each object

    This is equivalent to C code bellow which is from GSL document:

    .. sourcecode:: c

          const gsl_rng_type **t, **t0;
          t0 = gsl_rng_types_setup ();
          for (t = t0; *t != 0; t++)
            {
              printf ("%s\n", (*t)->name);  /* instead, store t to dict */
            }

    """
    t = func.gsl_rng_types_setup()
    dt = ctypes.sizeof(c_void_p)
    dct = {}
    while True:
        a = c_void_p.from_address(t)
        if a.value is None:
            break
        name = c_char_p.from_address(a.value).value
        dct[name] = ctypes.cast(a, _c_gsl_rng_type_p)
        t += dt
    return dct


class gsl_rng(object):
    _gsl_rng_alloc = func.gsl_rng_alloc
    _gsl_rng_set = func.gsl_rng_set
    _gsl_rng_free = func.gsl_rng_free
    _gsl_rng_type_p_dict = _get_gsl_rng_type_p_dict()
    _ctype_ = _c_gsl_rng_p  # for railgun

    def __init__(self, seed=None, name='mt19937'):
        self._gsl_rng_name = name
        self._gsl_rng_type_p = self._gsl_rng_type_p_dict[name]
        self._cdata_ = self._gsl_rng_alloc(self._gsl_rng_type_p)
        # the name '_cdata_' is for railgun
        if seed is not None:
            self.set(seed)

    def __del__(self):
        self._gsl_rng_free(self._cdata_)

    def set(self, seed):
```

```
        self._gsl_rng_set(self._cdata_, seed)

    _gsl_ran_gaussian = {
        '': func.gsl_ran_gaussian,
        'ziggurat': func.gsl_ran_gaussian_ziggurat,
        'ratio_method': func.gsl_ran_gaussian_ratio_method,
        }

    def ran_gaussian(self, sigma=1.0, method=''):
        return self._gsl_ran_gaussian[method](self._cdata_, sigma)
```

# Kaplan-Yorke map

## Output

## Python code

```
from railgun import SimObject, relpath

class KaplanYorkeMap(SimObject):
    """
    Simulating Kaplan-Yorke Map using RailGun
    """

    _clibname_ = 'libkaplan_yorke_map.so'  # name of shared library
    _clibdir_ = relpath('.', __file__)  # library directory
    _cmembers_ = [  # declaring members of struct
        'num_i',  # num_* as size of array (no need to write `int`)
        'double xt[i]',  # array with num_i elements
        'double yt[i]',  # array with num_i elements
        'double mu',
        'double lmd',
        ]
    _cfuncs_ = ["gene_seq()"]


kym = KaplanYorkeMap(num_i=10**5, mu=2, lmd=0.4, xt_0=0.1, yt_0=0.1)
kym.setv()
kym.gene_seq()
(xt, yt) = kym.getv('xt, yt')

import pylab
pylab.figure(1, figsize=(4,3))
pylab.plot(xt, yt, 'k.', markersize=0.1)
pylab.show()
```

## C code

```
typedef struct kaplanyorkemap_{
  int num_i;
  double *xt;
  double *yt;
```

```
  double mu;
  double lmd;
} KaplanYorkeMap;

int KaplanYorkeMap_gene_seq(KaplanYorkeMap *self)
{
  int st;
  for (st = 1; st < self->num_i; ++st){
    self->xt[st] = 1 - self->mu * self->xt[st-1] * self->xt[st-1];
    self->yt[st] = self->lmd * self->yt[st-1] + self->xt[st-1];
  }
  return 0;
}
```

## Definition of the map

$$
\begin{aligned}
x_{n+1} &= T(x_n) \\
y_{n+1} &= \lambda y_n + h(x_n) \\
T(x) &= 1 - \mu x^2 \\
h(x) &= x
\end{aligned}
$$

## References

- Thermodynamics of chaotic systems: an introduction By Christian Beck, Friedrich Schlögl - Google Books
- Kaplan–Yorke map - Wikipedia, the free encyclopedia

CHAPTER 5

# Change Log

## v0.1.9

- Python 3.5 and 3.6 are supported now.
- *SimObject.reallocate()* is added.
- Changing *num_** raises error, as it could cause segmentation fault.
- *SimObject.cinfo* is added.
- Types other than *int* can be used for *num_**.
- C functions can be accessed via `super()`. This eliminates the need of the *cwrap_** function.
- *cwrap_** function defined in subclasses are not executed anymore. This change is backward-incompatible.

## v0.1.8

- Fix deep copy / pickle bug with multi-dimensional arrays.

## v0.1.7

- Shallow copy (`copy.copy()`), deep copy (`copy.deepcopy()`) and pickling (`pickle`) are supported.
- Documentation is improved.

CHAPTER 6

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

# Index

## Symbols



## A



## C



## G



## N



## R



## S



## Y