MMFUtils Documentation

Release 0.5.1

Michael McNeil Forbes

Mar 16, 2020

CONTENTS

Contents:

CHAPTER

ONE

MMFUTILS

1.1 mmfutils.interface

Stand-in for zope.interface if it is not available.

interface mmfutils.interface.**Interface**

class mmfutils.interface.**Attribute**(*__name__*, *__doc__=''*)
Bases: zope.interface.interface.Element

Attribute descriptions

interface = None

class mmfutils.interface.**implementer**(**interfaces*)
Bases: object

Declare the interfaces implemented by instances of a class.

This function is called as a class decorator.

The arguments are one or more interfaces or interface specifications (*~zope.interface.interfaces.IDeclaration* objects).

The interfaces given (including the interfaces in the specifications) are added to any interfaces previously declared.

Previous declarations include declarations for base classes unless implementsOnly was used.

This function is provided for convenience. It provides a more convenient way to call *classImplements*. For example:

```
@implementer(I1)
class C(object):
    pass
```

is equivalent to calling:

```
classImplements(C, I1)
```

after the class has been created.

mmfutils.interface.**verifyObject**(*iface*, *candidate*, *tentative=0*)

mmfutils.interface.**verifyClass**(*iface*, *candidate*, *tentative=0*)

mmfutils.interface.**describe_interface**(*interface*, *format='ipython'*)
Return an HTML object for Jupyter notebooks that describes the interface.

Parameters

- **interface** (*Interface*) – Interface to extract documentation from.
- **format** (*'rst', 'html', 'ipython'*) – Return format. 'rst' is raw ReStructuredText, 'html' is packaged as HTML, and 'ipython' is packaged as an IPython.display.HTML() object suitable for Jupyter notebooks.

Example

```
>>> class IExample(Interface):
...     x = Attribute("Floating point number")
...     def two():
...         "Return two"
>>> print(describe_interface(IExample, format='rst').strip())
``IExample``

 Attributes:

  ``x`` -- Floating point number

Methods:

  ``two()`` -- Return two
```

You can also get this wrapped as HTML:

```
>>> print(describe_interface(IExample, format='html').strip())
<!DOCTYPE html ...
<p><tt class="docutils literal">IExample</tt></p>
<blockquote>
<p>Attributes:</p>
<blockquote>
<tt class="docutils literal">x</tt> -- Floating point number</blockquote>
<p>Methods:</p>
<blockquote>
<tt class="docutils literal">two()</tt> -- Return two</blockquote>
</blockquote>
</div>
```

In a Jupyter notebook, this will properly display:

```
>>> describe_interface(IExample)
<IPython.core.display.HTML object>
```

Other formats are not yet supported:

```
>>> describe_interface(IExample, format='WYSIWYG')
Traceback (most recent call last):
   ...
NotImplementedError: format WYSIWYG not supported
```

1.2 mmfutils.containers

Provides convenience containers that support pickling and archiving.

Archiving is supported through the interface defined by the `persist` package (though use of that package is optional and it is not a dependency).

class mmfutils.containers.**ObjectBase**(***kw*)

Bases: object

General base class with a few convenience methods.

Summary:

- *__init__()* sets parameters and calls *init()*
- *init()* calculates all other parameters.

Motivation:

The motivation is objects intended to be used in computationally demanding settings. The idea is that the *init()* method will be called before starting a computation, ensuring that the object is up-to-date, and performing any expensive calculations. Then the object can be used in a computationally demanding setting.

I have been using this approach for some time and am generally happy with how it works. Some care is needed nesting calls to *init()* in derived classes, but I have found these cases easy to deal with. Other approaches such as using properties can carry a performance hit. Writing setters can work well, but demands a lot from the developer and become very complicated when properties depend on each other.

Details:

- The constructor *__init__()* should only be used to set variables in *self*. The reason is that the code here uses the variables set in the constructor to determine which attributes need to be pickled. Initialization of computed attributes should instead be done in the *init()* method .
- The constructor *__init__()* takes *kwargs* and will set these. This allows using *super().__init__()*. See e.g.:

  https://rhettinger.wordpress.com/2011/05/26/super-considered-super/
- The constructor will store all assigned variables (in *__dict__()*) to a list *picklable_attributes* which can then be used by the *Object* to provide pickling services. Here we simply use this to set an *initialized* flag to note the user that the object might be invalid and need *init()* called again.
- The *init()* method should make sure that the object ends in a consistent state so that further computations (without users setting attributes) can be computed efficiently. If the user sets attributes, *init()* should be called again.

Note: Do not use any of the following variables:

- **`picklable_attributes:** Reserved for the list of attributes that will be pickled. If this has been stored in *self.__dict__* then the constructor chain has finished processing.
- ***initialized*:** Used to flag if attributes have been changed but without *init()* being called.

By default setting any attribute in *picklable_attributes* will set the *initialized* flag to *False*. This will be set to *True* when *init()* is called. Objects can then include an *assert self.initialized* in the appropriate places.

Note: This redefines __setattr__ to provide the behaviour.

Examples

```
>>> class A(ObjectBase):
...     def __init__(self, x=0):
...         super().__init__(x=x)
...     def init(self):
...         self.x1 = self.x + 1    # A dependent variable
...         super().init()
...     def check(self):
...         if not self.initialized:
...             raise AssertionError("Please call init()!")
...         return self.x1 == self.x + 1
>>> a = A(x=0)
>>> a.check()
True
>>> a.x = 2.0
>>> a.check()
Traceback (most recent call last):
...
AssertionError: Please call init()!
>>> a.init()
>>> a.check()
True
```

__setattr__(*key*, *value*)
Sets the *initialized* flag to *False* if any picklable attribute is changed.

get_persistent_rep(*env*)
Return *(rep, args, imports)*.

Define a persistent representation *rep* of the instance self where the instance can be reconstructed from the string rep evaluated in the context of dict args with the specified imports = list of *(module, iname, uiname)* where one has either *import module as uiname*, *from module import iname* or *from module import iname as uiname*.

This satisfies the *IArchivable* interface for the *persist* package.

init()
Initialize Object.

initialized = False

picklable_attributes = ()

class mmfutils.containers.**Object**(**kw*)
Bases: mmfutils.containers.ObjectMixin, *mmfutils.containers.ObjectBase*

Extension of Object with pickling support.

Pickling will save only those variables defined in *picklable_attributes* which is usually defined when the base *__init__* is finished. The *init()* method will be called upon unpickling, thereby allowing unpicklable objects to be used (in particular function instances).

Note: Do not use any of the following variables:

- ***_empty_state*:** Reserved for objects without any state
- *_independent_attributes*:
- *_dependent_attributes*:

- ***_strict*:** If *True*, then only picklable attributes will be settable through *__setattr__()*.
- ***_check*:** If *True*, check that objects are actually picklable when they are set.
- ***_reserved_attributes*:** List of special attributes that should be excluded from processing.

To allow for some variables to be set without invalidating the object we also check the set of names *_independent_attributes*.

Examples

```
>>> class A(Object):
...     def __init__(self, x=0):
...         self.x = x
...         super().__init__()
...     def init(self):
...         self.x1 = self.x + 1    # A dependent variable
...         super().init()
...     def check(self):
...         if not self.initialized:
...             raise AssertionError("Please call init()!")
...         return self.x1 == self.x + 1
>>> a = A(x=0)
>>> a.check()
True
>>> a.x = 2.0
>>> a.check()
Traceback (most recent call last):
...
AssertionError: Please call init()!
>>> a.init()
>>> a.check()
True
```

class mmfutils.containers.**Container**(**argv*, ***kw*)

Bases: *mmfutils.containers.Object*, collections.abc.Sized, collections.abc.Iterable, collections.abc.Container

Simple container object.

Attributes can be specified in the constructor. These will form the representation of the object as well as picking. Additional attributes can be assigned, but will not be pickled.

Examples

```
>>> c = Container(b='Hi', a=1)
>>> c                          # Note: items sorted for consistent repr
Container(a=1, b='Hi')
>>> c.a
1
>>> c.a = 2
>>> c.a
2
>>> tuple(c)                   # Order is lexicographic
(2, 'Hi')
```

(continues on next page)

(continued from previous page)

```
>>> c.x = 6                 # Will not be pickled: only for temp usage
>>> c.x
6
>>> 'a' in c
True
>>> 'x' in c
False
>>> import pickle
>>> c1 = pickle.loads(pickle.dumps(c))
>>> c1
Container(a=2, b='Hi')
>>> c1.x
Traceback (most recent call last):
...
AttributeError: 'Container' object has no attribute 'x'
```

class mmfutils.containers.**ContainerList**(**argv*, ***kw*)

Bases: *mmfutils.containers.Container*, collections.abc.Sequence

Simple container object that behaves like a list.

Examples

```
>>> c = ContainerList(b='Hi', a=1)
>>> c                       # Note: items sorted for consistent repr
ContainerList(a=1, b='Hi')
>>> c[0]
1
>>> c[0] = 2
>>> c.a
2
>>> tuple(c)                # Order is lexicographic
(2, 'Hi')
```

class mmfutils.containers.**ContainerDict**(**argv*, ***kw*)

Bases: *mmfutils.containers.Container*, collections.abc.MutableMapping

Simple container object that behaves like a dict.

Attributes can be specified in the constructor. These will form the representation of the object as well as picking. Additional attributes can be assigned, but will not be pickled.

Examples

```
>>> from collections import OrderedDict
>>> c = ContainerDict(b='Hi', a=1)
>>> c                       # Note: items sorted for consistent repr
ContainerDict(a=1, b='Hi')
>>> c['a']
1
>>> c['a'] = 2
>>> c.a
2
>>> OrderedDict(c)
OrderedDict([('a', 2), ('b', 'Hi')])
```

1.3 mmfutils.contexts

Various useful contexts.

class mmfutils.contexts.**CoroutineWrapper**(*coroutine*)
Bases: object

Wrapper for coroutine contexts that allows them to function as a context but also as a function. Similar to open() which may be used both in a function or as a file object. Note: be sure to call close() if you do not use this as a context.

close()

send(**v*)

class mmfutils.contexts.**NoInterrupt**(*ignore=True*)
Bases: object

Suspend the various signals during the execution block and a simple mechanism to allow threads to be interrupted.

Parameters

- **ignore** (*bool*) – If True, then do not raise a KeyboardInterrupt if a soft interrupt is caught unless forced by multiple interrupt requests in a limited time.
- **are two main entry points** (*There*) –
- **and within a NoInterrupt() context.** (*method,*) –
- **Thread** (*Main*) –
- **----------** –
- **executed in a context from the main thread, a signal handler** (*When*) –
- **established which captures interrupt signals and represents** (*is*) –
- **instead as a boolean flag (conventionally called** (*them*) –
- **"interrupted")** –
- **interrupt suppression can be enabled by creating a** (*Global*) –
- **instance and calling suspend() This will stay in** (NoInterrupt *()*) –
- **until restore() is called, a forcing interrupt is received,** (*effect*) –
- **the instance is deleted. Additional calls to suspend() will** (*or*) –
- **the handlers, but they will not be nested.** (*reinstall*) –
- **can also be suspended in contexts. These can be** (*Interrupts*) –
- **These instances will become False at the end of the** (*nested.*) –
- **context.** –
- **Threads** (*Auxiliary*) –
- **----------** –

- **threads can create instances of NoInterrupt() or use** (*Auxiliary*) –
- **but cannot call suspend() or restore() In these cases** (*contexts,*) –
- **context does not suspend signals (see below), but the flag is** (*the*) –
- **useful as it can act as a signal force the auxiliary thread** (*still*) –
- **terminate if an interrupt is received in the main thread.** (*to*) –
- **couple of notes about using the context in auxiliary threads.** (*A*) –
- **Either suspend() must be called globally or a context must** (*1.*) – first be created in the main thread - otherwise the signal handlers will not be installed. An exception will be raised if an auxiliary thread tries to create a context without the handlers being installed. this case.
- **As stated in the python documents, signal handlers are always** (*2.*) – executed in the main thread. Likewise, only the main thread is allowed to set new signal handlers. Thus, the signal interrupting facilities provided here only work properly in the main thread. Also, forcing an interrupt cannot raise an exception in the auxiliary threads: one must wait for them to respond to the changed "interrupted" value.

  For more information about killing threads see:

  – http://stackoverflow.com/questions/323972/ is-there-any-way-to-kill-a-thread-in-python

force_n
Number of interrupts to force signal.

Type int

force_timeout
Time in which force_n interrupts must be received to trigger a forced interrupt.

Type float

Examples

The simplest use-cases look like these:

Simple context:

```
>>> with NoInterrupt():
...     pass             # do something
```

Context with a cleanly aborted loop:

```
>>> with NoInterrupt() as interrupted:
...     done = False
...     while not interrupted and not done:
...         # Do something
...         done = True
```

Map:

```
>>> NoInterrupt().map(abs, [1, -1, 2, -2])
[1, 1, 2, 2]
```

Keyboard interrupt signals are suspended during the execution of the block unless forced by the user (3 rapid interrupts within 1s). Interrupts are ignored by default unless *ignore=False* is specified, in which case they will be raised when the context is ended.

If you want to control when you exit the block, use the *interrupted* flag. This could be used, for example, while plotting frames in an animation (see doc/Animation.ipynb). Without the NoInterrupt() context, if the user sends a keyboard interrupt to the process while plotting, at best, a huge stack-trace is produced, and at worst, the kernel will crash (randomly depending on where the interrupt was received). With this context, the interrupt will change *interrupted* to True so you can exit the context when it is safe.

The last case is mapping a function to data. This will allow the user to interrupt the process between function calls.

In the following examples we demonstrate this by simulating interrupts

```
>>> import os, signal, time
>>> def simulate_interrupt(force=False):
...     os.kill(os.getpid(), signal.SIGINT)
...     if force:
...         # Simulated a forced interrupt with multiple signals
...         os.kill(os.getpid(), signal.SIGINT)
...         os.kill(os.getpid(), signal.SIGINT)
...     time.sleep(0.1)   # Wait so signal can be received predictably
```

This loop will get interrupted in the middle so that m and n will not be the same.

```
>>> def f(n, interrupted=False, force=False, interrupt=True):
...     while n[0] < 10 and not interrupted:
...         n[0] += 1
...         if n[0] == 5 and interrupt:
...             simulate_interrupt(force=force)
...         n[1] += 1
```

```
>>> n = [0, 0]
>>> f(n, interrupt=False)
>>> n
[10, 10]
```

```
>>> n = [0, 0]
>>> try:  # All doctests need to be wrapped in try blocks to not kill py.test!
...     f(n)
... except KeyboardInterrupt as err:
...     print("KeyboardInterrupt: {}".format(err))
KeyboardInterrupt:
>>> n
[5, 4]
```

Now we protect the loop from interrupts. >>> n = [0, 0] >>> try: … with NoInterrupt(ignore=False) as interrupted: … f(n) … except KeyboardInterrupt as err: … print("KeyboardInterrupt: {}".format(err)) KeyboardInterrupt: >>> n [10, 10]

One can ignore the exception if desired (this is the default as of 0.4.11): >>> n = [0, 0] >>> with NoInterrupt() as interrupted: … f(n) >>> n [10, 10]

Three rapid exceptions will still force an interrupt when it occurs. This might occur at random places in

your code, so don't do this unless you really need to stop the process. >>> n = [0, 0] >>> try: ... with NoInterrupt(ignore=False) as interrupted: ... f(n, force=True) ... except KeyboardInterrupt as err: ... print("KeyboardInterrupt: {}".format(err)) KeyboardInterrupt: >>> n [5, 4]

If *f()* is slow, we might want to interrupt it at safe times. This is what the *interrupted* flag is for:

```
>>> n = [0, 0]
>>> try:
...     with NoInterrupt(ignore=False) as interrupted:
...         f(n, interrupted)
... except KeyboardInterrupt as err:
...     print("KeyboardInterrupt: {}".format(err))
KeyboardInterrupt:
>>> n
[5, 5]
```

Again: the exception can be ignored >>> n = [0, 0] >>> with NoInterrupt() as interrupted: ... f(n, interrupted) >>> n [5, 5]

__bool__()
Return True if interrupted.

__enter__()
Enter context.

__nonzero__()
Return True if interrupted.

force_n = 3

force_timeout = 1

classmethod handle_original_signal(*signum*, *frame*)
Call the original handler.

classmethod handle_signal(*signum*, *frame*)
Custom signal handler.

This stores the signal for later processing unless it was forced or there are no current contexts, in which case the original handlers will be called.

classmethod is_registered()
Return True if handlers are registered.

map(*function*, *sequence*, **v*, ***kw*)
Map function onto sequence until interrupted or done.

Interrupts will not occur inside function() unless forced.

classmethod register()
Register the handlers so that signals can be suspended.

classmethod reset()
Reset the signal logs and return last signal *(signum, frame, time)*.

classmethod resume(*signals=None*)
Resumes the specified signals.

classmethod set_signals(*signals*)
Change the signal handlers.

Note: This does not change the signals listed in _suspended_signals list.

Parameters **signals** (*set()*) – Set of signal numbers.

classmethod suspend(*signals=None*)
Suspends the specified signals.

classmethod unregister(*full=False*)
Reset handlers to the original values. No more signal suspension.

Parameters **full** (*bool*) – If True, do a full reset, including counts.

mmfutils.contexts.**coroutine**(*coroutine*)
Decorator for a context that yeilds an function from a coroutine.

This allows you to write functions that maintain state between calls. The use as a context here ensures that the coroutine is closed.

Examples

Here is an example based on that suggested by Thomas Kluyver: http://takluyver.github.io/posts/readable-python-coroutines.html

```
>>> @coroutine
... def get_have_seen(case_sensitive=False):
...     seen = set()    # Set of words already seen.  This is the "state"
...     word = (yield)
...     while True:
...         if not case_sensitive:
...             word = word.lower()
...         result = word in seen
...         seen.add(word)
...         word = (yield result)
>>> with get_have_seen(case_sensitive=False) as have_seen:
...     print(have_seen("hello"))
...     print(have_seen("hello"))
...     print(have_seen("Hello"))
...     print(have_seen("hi"))
...     print(have_seen("hi"))
False
True
True
False
True
>>> have_seen("hi")
Traceback (most recent call last):
   ...
StopIteration
```

You can also use this as a function (like open()) but don't forget to close it.

```
>>> have_seen = get_have_seen(case_sensitive=True)
>>> have_seen("hello")
False
>>> have_seen("hello")
True
>>> have_seen("Hello")
False
>>> have_seen("hi")
False
>>> have_seen("hi")
True
```

(continues on next page)

(continued from previous page)

```
>>> have_seen.close()
>>> have_seen("hi")
Traceback (most recent call last):
   ...
StopIteration
```

mmfutils.contexts.**is_main_thread**()
Return True if this is the main thread.

mmfutils.contexts.**nointerrupt**(*f*)
Decorator that suspends signals and passes an interrupted flag to the protected function. Can only be called from the main thread: will raise a RuntimeError otherwise (use *@interrupted* instead).

Examples

```
>>> @nointerrupt
... def f(interrupted):
...     for n in range(3):
...         if interrupted:
...             break
...         print(n)
...         time.sleep(0.1)
>>> f()
0
1
2
```

1.4 mmfutils.debugging

Some debugging tools.

Most of these are implemented as decorators.

class mmfutils.debugging.**persistent_locals**(*func*)
Bases: object

Decorator that stores the function's local variables.

Examples

```
>>> @persistent_locals
... def f(x):
...     y = x**2
...     z = 2*y
...     return z
>>> f(1)
2
>>> sorted(f.locals.items())
[('x', 1), ('y', 1), ('z', 2)]
>>> f.clear_locals()
>>> f.locals
{}
```

clear_locals()

property locals

mmfutils.debugging.**debug**(*v, **kw)

Decorator to wrap a function and dump its local scope.

Parameters (or env) (*locals*) – Function's local variables will be updated in this dict. Use locals() if desired.

Examples

```
>>> env = {}
>>> @debug(env)
... def f(x):
...     y = x**2
...     z = 2*y
...     return z
>>> f(1)
2
>>> sorted(env.items())
[('x', 1), ('y', 1), ('z', 2)]
```

This will put the local variables directly in the global scope:

```
>>> @debug(locals())
... def f(x):
...     y = x**2
...     z = 2*y
...     return z
>>> f(1)
2
>>> x, y, z
(1, 1, 2)
>>> f(2)
8
>>> x, y, z
(2, 4, 8)
```

If an exception is raised, you still have access to the results:

```
>>> env = {}
>>> @debug(env)
... def f(x):
...     y = 2*x
...     z = 2/y
...     return z
>>> f(0)
Traceback (most recent call last):
  ...
  File "<doctest mmfutils.debugging.debug[14]>", line 1, in <module>
    f(0)
  File "<doctest mmfutils.debugging.debug[13]>", line 4, in f
    z = 2/y
ZeroDivisionError: division by zero
>>> sorted(env.items())
[('x', 0), ('y', 0)]
```

1.5 mmfutils.math

1.5.1 mmfutils.math.integrate

Integration Utilities.

mmfutils.math.integrate.**quad**(*f*, *a*, *b*, *epsabs=1e-12*, *epsrel=1e-08*, *limit=1000*, *points=None*, ***kwargs*)

An improved version of integrate.quad that does some argument checking and deals with points properly.

Return (ans, err).

Examples

```
>>> def f(x): return 1./x**2
>>> (ans, err) = quad(f, 1, np.inf, points=[])
>>> abs(ans - 1.0) < err
True
>>> (ans, err) = quad(f, 1, np.inf, points=[3.0, 2.0])
>>> abs(ans - 1.0) < err
True
```

mmfutils.math.integrate.**mquad**(*f*, *a*, *b*, *abs_tol=1e-12*, *verbosity=0*, *fa=None*, *fb=None*, *save_fx=False*, *res_dict=None*, *max_fcnt=10000*, *min_step_size=None*, *norm=<function <lambda>>*, *points=None*)

Return (res, err) where res is the numerically evaluated integral using adaptive Simpson quadrature.

mquad tries to approximate the integral of function f from a to b to within an error of abs_tol using recursive adaptive Simpson quadrature. mquad allows the function y = f(x) to be array-valued. In the matrix valued case, the infinity norm of the matrix is used as it's "absolute value".

Parameters

- **f** (*function*) – Possibly array valued function to integrate. If this emits a NaN, then an AssertionError is raised to allow the user to optimize this check away (as it exists in the core of the loops)
- **b** (*a,*) – Integration range (a, b)
- **fb** (*fa,*) – f(a) and f(b) respectively (if already computed)
- **abs_tol** (*float*) – Approximate absolute tolerance on integral
- **verbosity** (*int*) – Display info if greater than zero. Shows the values of [fcnt a b-a Q] during the iteration.
- **save_fx** (*bool*) – If True, then save the abscissa and function values in res_dict.
- **res_dict** (*dict*) – Details are stored here. Pass a dictionary to access these. The dictionary will be modified.

  fcnt : Number of function evaluations. xy : List of pairs (x, f(x)) if save_fx is defined.
- **max_fcnt** (*int*) – Maximum number of function evaluations.
- **min_step_size** (*float*) – Minimum step size to limit recursion.
- **norm** (*function*) – Norm to use to determine convergence. The absolute error is determined as *norm(f(x) - F)*.

- **points** (*[float]*) – List of special points to be included in abscissa.

Notes

Based on "adaptsim" by Walter Gander. Ref: W. Gander and W. Gautschi, "Adaptive Quadrature Revisited", 1998. http://www.inf.ethz.ch/personal/gander

Examples

Orthogonality of planewaves on [0, 2pi]

```
>>> def f(x):
...     v = np.exp(1j*np.array([[1.0, 2.0, 3.0]])*x)
...     return v.T.conj()*v/2.0/np.pi
>>> ans = mquad(f, 0, 2*np.pi)
>>> abs(ans - np.eye(ans.shape[0])).max() < _ABS_TOL
True
```

```
>>> res_dict = {}
>>> def f(x): return x**2
>>> ans = mquad(f, -2, 1, res_dict=res_dict, save_fx=True)
>>> abs(ans - 3.0) < _ABS_TOL
True
>>> x = np.array([xy[0] for xy in res_dict['xy']])
>>> y = np.array([xy[1] for xy in res_dict['xy']])
>>> abs(y - f(x)).max()
0.0
```

This works, but triggers a warning because of the singular # endpoints. >>> logger = logging.getLogger() >>> logger.disabled = True >>> def f(x): return 1.0/np.sqrt(x) + 1.0/np.sqrt(1.0-x) >>> abs(mquad(f, 0, 1, abs_tol=1e-8) - 4.0) < 1e-8 True >>> logger.disabled = False

```
>>> def f(x):
...     if x < 0:
...         return 0.0
...     else:
...         return 1.0
>>> abs(mquad(f, -2.0, 1.0) - 1.0) < 1e-10
True
```

```
>>> def f(x): return 1./x
>>> mquad(f, 1, np.inf)
Traceback (most recent call last):
    ...
ValueError: Infinite endpoints not supported.
```

mmfutils.math.integrate.**Richardson**(*f*, *ps=None*, *l=2*, *n0=1*)

Compute the Richardson extrapolation of f given the function

$$f(N) = f + \sum_{n=0}^{\infty} \frac{\alpha_n}{N^{p_n}}$$

The extrapolants are stored in the array *S`[n, s] where `S[n, 0] = f(n0*l**n)* and *S[n, s]* is the s'th extrapolant.

Note: It is crucial for performance that the powers \$p_n\$ be properly characterized. If you do not know the form of the errors, then consider using a non-linear acceleration technique such as `levin_sum()`.

Parameters **ps** (*iterable*) – Iterable returning the powers \$p_n\$. To generate the sequence \$p_0 + m d_p\$ for example, use `itertools.count``(p0, dp)()`.

Examples

Here we consider

$$f(N) = \sum_{n=1}^{N} \frac{1}{n^2} = \frac{\pi^2}{6} + (N^{-1})$$

```
>>> def f(N): return sum(np.arange(1, N+1, dtype=float)**(-2))
>>> r = Richardson(f, l=3, n0=2)
>>> for n in range(9):
...     x = next(r)
>>> err = abs(x - np.pi**2/6.0)
>>> assert err < 1e-14, 'err'
```

Now some other examples with different p values:

$$f(N) = \sum_{n=1}^{N} \frac{1}{n^4} = \frac{\pi^4}{90} + (N^{-3})$$

```
>>> def f(N): return sum(np.arange(1, N+1, dtype=float)**(-4))
>>> r = Richardson(f, ps=itertools.count(3,1))
>>> for n in range(8):
...     x = next(r)
>>> err = abs(x - np.pi**4/90.0)
>>> assert err < 1e-14, 'err'
```

$$f(N) = \sum_{n=1}^{N} \frac{1}{n^6} = \frac{\pi^6}{945} + (N^{-5})$$

```
>>> def f(N): return sum(np.arange(1, N+1, dtype=float)**(-6))
>>> r = Richardson(f, ps=itertools.count(5))
>>> for n in range(7):
...     x = next(r)
>>> err = abs(x - np.pi**6/945.0)
>>> assert err < 1e-14, 'err'
```

Richardson works with array valued functions:

```
>>> def f(N): return np.array([sum(np.arange(1, N+1, dtype=float)**(-2)),
...                            sum(np.arange(1, N+1, dtype=float)**(-4))])
>>> r = Richardson(f, l=3, n0=2)
>>> for n in range(7):
...     x = next(r)
>>> err = abs(x - np.array([np.pi**2/6.0, np.pi**4/90.0])).max()
>>> assert err < 1e-13, 'err'
```

It also works for complex valued functions:

```
>>> def f(N): return (sum(np.arange(1, N+1, dtype=float)**(-2)) +
...                   1j*sum(np.arange(1, N+1, dtype=float)**(-4)))
>>> r = Richardson(f, l=3, n0=2)
>>> for n in range(7):
...     x = next(r)
>>> err = abs(x - (np.pi**2/6.0 + 1j*np.pi**4/90.0))
>>> assert err < 1e-13, 'err'
```

mmfutils.math.integrate.**rsum**(*f*, *N0=0*, *ps=None*, *l=2*, *abs_tol=1e-12*, *rel_tol=1e-08*, *verbosity=0*)

Sum f using Richardson extrapolation.

Examples

```
>>> def f(n):
...     return 1./(n+1)**2
>>> res, err = rsum(f)
>>> res
1.6449340668...
>>> abs(res - np.pi**2/6.0) < err
True
```

1.5.2 mmfutils.math.differentiate

Differentiation.

mmfutils.math.differentiate.**differentiate**(*f*, *x=0.0*, *d=1*, *h0=1.0*, *l=1.4*, *nmax=10*, *dir=0*, *p0=1*, *err=[0]*)

Evaluate the numerical dth derivative of f(x) using a Richardson extrapolation of the finite difference formula.

Parameters

- **f** (*function*) – The function to compute the derivative of.
- **x** (*{float, array}*) – The derivative is computed at this point (or at these points if the function is vectorized.
- **d** (*int, optional*) – Order of derivative to compute. *d=0* is the function *f(x)*, *d=1* is the first derivative etc.
- **h0** (*float, optional*) – Initial stepsize. Should be on about a factor of 10 smaller than the typical scale over which *f(x)* varies significantly.
- **l** (*float, optional*) – Richardson extrapolation factor. Stepsizes used are *h0/l**n*
- **nmax** (*int, optional*) – Maximum number of extrapolation steps to take.
- **dir** (*int, optional*) – If *dir* < *0*, then the function is only evaluated to the left, if positive, then only to the right, and if zero, then centered form is used.

Returns **df** – Order *d* derivative of *f* at *x*.

Return type {float, array}

Other Parameters

- **p0** (*int, optional*) – This is the first non-zero term in the taylor expansion of either the difference formula. If you know that the first term is zero (because of the coefficient), then you should set *p0=2* to skip that term.

  Note: This is not the power of the term, but the order. For centered difference formulae, *p0=1* is the *h**2* term, which would vanish if third derivative vanished at *x* while for the forward difference formulae this is the *h* term which is absent if the second derivative vanishes.

- **err[0]** (*float*) – This is mutated to provide an error estimate.

Notes

This implementation uses the Richardson extrapolation to extrapolate the answer. This is based on the following Taylor series error formulae:

$$\begin{aligned} f'(x) &= \frac{f(x+h) - f(x)}{h} - h\frac{f'')}{2} + \cdots \\ &= \frac{f(x+h) - f(x-h)}{2h} - h^2\frac{f''}{3!} + \cdots \\ f''(x) &= \frac{f(x+2h) - 2f(x+h) + f(x)}{h^2} - hf^{(3)} + \cdots \\ &= \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} - 2h^2\frac{f^{(4)}}{4!} + \cdots \end{aligned}$$

If we let $h = 1/N$ then these formula match the expected error model for the Richardson extrapolation

$$S(h) = S(0) + ah^p + (h^{p+1})$$

with $p=1$ for the one-sided formulae and $p=2$ for the centered difference formula respectively.

See `mmf.math.integrate.Richardson`

See also:

mmfutils.math.integrate.Richardson() Richardson extrapolation

Examples

```
>>> from math import sin, cos
>>> x = 100.0
>>> assert(abs(differentiate(sin, x, d=0)-sin(x))<1e-15)
>>> assert(abs(differentiate(sin, x, d=1)-cos(x))<1e-14)
>>> assert(abs(differentiate(sin, x, d=2)+sin(x))<1e-13)
>>> assert(abs(differentiate(sin, x, d=3)+cos(x))<1e-11)
>>> assert(abs(differentiate(sin, x, d=4)-sin(x))<1e-9)
>>> differentiate(abs, 0.0, d=1, dir=1)
1.0
>>> differentiate(abs, 0.0, d=1, dir=-1)
-1.0
>>> differentiate(abs, 0.0, d=1, dir=0)
0.0
```

Note that the Richardson extrapolation assumes that *h0* is small enough that the truncation errors are controlled by the taylor series and that the taylor series properly describes the behaviour of the error. For example, the following will not converge well, even though the derivative is well defined:

```
>>> def f(x):
...     return np.sign(x)*abs(x)**(1.5)
>>> df = differentiate(f, 0.0)
>>> abs(df) < 0.1
True
>>> abs(df) < 0.01
False
>>> abs(differentiate(f, 0.0, nmax=100)) < 3e-8
True
```

Sometimes, one may compensate by increasing nmax. (One could in principle change the Richardson parameter p, but this is optimized for analytic functions.)

The *differentiate()* also works over arrays if the function *f* is vectorized:

```
>>> x = np.linspace(0, 100, 10)
>>> assert(max(abs(differentiate(np.sin, x, d=1) - np.cos(x))) < 3e-15)
```

mmfutils.math.differentiate.**hessian**(*f*, *x*, ***kw*)

Return the gradient Hessian matrix of *f(x)* at *x* using *differentiate()*. This is not efficient.

Parameters

- **f** (*function*) – Scalar function of an array.
- **x** (*array-like*) – Derivatives evaluated at this point.
- **kw** (*dict*) – See *differentiate()* for options.

Examples

```
>>> def f(x): return np.arctan2(*x)
>>> def df(x): return np.array([x[1], -x[0]])/np.sum(x**2)
>>> def ddf(x):
...     return np.array([[-2.*x[0]*x[1], -np.diff(x**2)[0]],
...                      [-np.diff(x**2)[0], 2.*x[0]*x[1]]])/np.sum(x**2)**2
>>> x = [0.1, 0.2]
>>> D, H = hessian(f, x, h0=0.1)
>>> x= np.asarray(x)
>>> D, df(x)
(array([ 4., -2.]), array([ 4., -2.]))
>>> H, ddf(x)
(array([[-16., -12.],
        [-12.,  16.]]),
 array([[-16., -12.],
        [-12.,  16.]]))
```

1.5.3 mmfutils.math.bases

mmfutils.math.bases.interfaces

mmfutils.math.bases.bases

mmfutils.math.bases.utils

1.5.4 mmfutils.math.bessel

1.5.5 mmfutils.math.linalg

Linear Algebra Routines

mmfutils.math.linalg.**block_diag**(*arrays*)

Create a new diagonal matrix from the provided arrays.

Parameters **b, c, ..** (*a,*) – Input arrays.

Returns **D** – Array with a, b, c, ... on the diagonal.

Return type ndarray

Examples

1.5.6 mmfutils.math.special

1.5.7 mmfutils.math.wigner

Wigner Ville distribution.

This module contains some FFT-based routines for computing the Wigner-Ville distribution.

mmfutils.math.wigner.**wigner_ville**(*psi*, *dt=1*, *make_analytic=False*, *skip=1*, *pad=True*)

Return *(ws, P)* where *P* is the Wigner Ville quasi-distribution for psi.

Assumes that psi is periodic. Note: the frequencies at which *P* is valid are half the frequencies normally associated with the wavefunction. Thus we also return the associated frequencies to avoid possible confusion.

Parameters

- **psi** (*array-like*) – The input signal.
- **dt** (*float*) – Step size for the input abscissa.
- **make_analytic** (*bool*) – If True, then negative frequency components are set to zero.
- **skip** (*int*) – Downsample the time-domain by skipping this many points.
- **pad** (*bool*) – If True, then pad the input array to remove aliasing artifacts.

1.6 mmfutils.optimize

Optimization tools.

mmfutils.optimize.**bracket_monotonic**(*f*, *x0=0.0*, *x1=1.0*, *factor=2.0*)
Return *(x0, x1)* where *f(x0)*f(x1) < 0*.

Assumes that f is monotonic and that the root exists.

Proceeds by increasing the size of the interval by *factor* in the direction of the root until the root is found.

Examples

```
>>> import math
>>> bracket_monotonic(lambda x:10 - math.exp(x))
(0.0, 3.0)
>>> bracket_monotonic(lambda x:10 - math.exp(-x), factor=1.5)
(4.75, -10.875)
```

mmfutils.optimize.**ubrentq**(*f*, *a*, *b*, **v*, ***kw*)
Version of *scipy.optimize.brentq* with uncertainty processing using the uncertainties package.

mmfutils.optimize.**usolve**(*f*, *a*, **v*, ***kw*)
Return the root of *f(x) = 0* with uncertainties propagated.

Parameters

- **f** (*function*) – Function to find root of *f(x) = 0*. Note: this must work with only a single argument even if the solver supports *args* etc. Thus, use *lambda x: f(x, ...)* or *functools.partial* if needed.
- **solver** (*function*) – Solver function (default is scipy.optimize.brentq).
- **kw** (*v,*) – Remaining arguments will be passed as *solver(f, a, *v, **kw)*.

1.7 mmfutils.performance

1.7.1 mmfutils.performance.blas

1.7.2 mmfutils.performance.fft

FFTW wrappers for high-performance computing.

This module requires you to have installed the fftw libraries and `pyfftw`. Note that you must build the fftw with all precisions using something like:

```
PREFIX=/data/apps/fftw
VER=3.3.4
for opt in " " "--enable-sse2 --enable-single" \
           "--enable-long-double" "--enable-quad-precision"; do
  ./configure --prefix="${PREFIX}/${VER}"\
              --enable-threads\
              --enable-shared\
              $opt
  make -j8 install
done
```

Note: The FFTW library does not work with negative indices for axis. Indices should first be normalized by `inds % len(shape)`.

mmfutils.performance.fft.**fft**(*Phi*, *axis=-1*)

mmfutils.performance.fft.**ifft**(*Phit*, *axis=-1*)

mmfutils.performance.fft.**fftn**(*Phi*, *axes=None*)

mmfutils.performance.fft.**ifftn**(*Phit*, *axes=None*)

mmfutils.performance.fft.**fftfreq**(*n*, *d=1.0*)

Return the Discrete Fourier Transform sample frequencies.

The returned float array *f* contains the frequency bin centers in cycles per unit of the sample spacing (with zero at the start). For instance, if the sample spacing is in seconds, then the frequency unit is cycles/second.

Given a window length *n* and a sample spacing *d*:

```
f = [0, 1, ...,   n/2-1,     -n/2, ..., -1] / (d*n)   if n is even
f = [0, 1, ..., (n-1)/2, -(n-1)/2, ..., -1] / (d*n)   if n is odd
```

Parameters

- **n** (*int*) – Window length.
- **d** (*scalar, optional*) – Sample spacing (inverse of the sampling rate). Defaults to 1.

Returns **f** – Array of length *n* containing the sample frequencies.

Return type ndarray

Examples

```
>>> signal = np.array([-2, 8, 6, 4, 1, 0, 3, 5], dtype=float)
>>> fourier = np.fft.fft(signal)
>>> n = signal.size
>>> timestep = 0.1
>>> freq = np.fft.fftfreq(n, d=timestep)
>>> freq
array([ 0.  ,  1.25,  2.5 , ..., -3.75, -2.5 , -1.25])
```

mmfutils.performance.fft.**resample**(*f*, *N*)

Resample f to a new grid of size N.

This uses the FFT to resample the function *f* on a new grid with *N* points. Note: this assumes that the function *f* is periodic. Resampling non-periodic functions to finer lattices may introduce aliasing artifacts.

Parameters

- **f** (*array*) – The function to be resampled. May be n-dimensional
- **N** (*int or array*) – The number of lattice points in the new array. If this is an integer, then all dimensions of the output array will have this length.

Examples

```
>>> def f(x, y):
...     "Function with only low frequencies"
...     return (np.sin(2*np.pi*x)-np.cos(4*np.pi*y))
>>> L = 1.0
>>> Nx, Ny = 16, 13    # Small grid
>>> NX, NY = 31, 24    # Large grid
>>> dx, dy = L/Nx, L/Ny
>>> dX, dY = L/NX, L/NY
>>> x = (np.arange(Nx)*dx - L/2)[:, None]
>>> y = (np.arange(Ny)*dy - L/2)[None, :]
>>> X = (np.arange(NX)*dX - L/2)[:, None]
>>> Y = (np.arange(NY)*dY - L/2)[None, :]
>>> f_XY = resample(f(x,y), (NX, NY))
>>> np.allclose(f_XY, f(X,Y))                          # To larger grid
True
>>> np.allclose(resample(f_XY, (Nx, Ny)), f(x,y))   # Back down
True
```

1.7.3 mmfutils.performance.numexpr

Tools for working with Numexp.

At present all this module provides is a safe way of importing `numexpr`. This prevents a hard crash (i.e. segfault) when the MKL is enabled but cannot be found. Just go:

```
>>> from mmfutils.performance.numexpr import numexpr
```

1.7.4 mmfutils.performance.threads

Thread Control

This module provides control of the number of threads used by the MKL and numexpr. It uses the global set SET_THREAD_HOOKS which should contain functions that take a single argument and set the number of threads for that particular part of the system.

Use *set_num_threads(nthreads)* to call all of these hooks.

`mmfutils.performance.threads.`**`set_num_threads`**(*nthreads*)

Set the maximum number of threads to use.

Calls all the hooks in *mmfutils.performance.threads.SET_THREAD_HOOKS*

Tools for high-performance computing.

This module may rely on many other packages that are not easy to install such as pyfftw and the corresponding fftw implementation.

1.8 mmfutils.plot

1.8.1 mmfutils.plot.animation

1.8.2 mmfutils.plot.cmaps

1.8.3 mmfutils.plot.colors

1.8.4 mmfutils.plot.contour

1.8.5 mmfutils.plot.errors

1.8.6 mmfutils.plot.publish

1.8.7 mmfutils.plot.rasterize

1.9 mmfutils.parallel

1.10 mmfutils.solve

1.10.1 Submodules

mmfutils.solve.broyden

1.10.2 Module contents

1.11 mmfutils.testing

Testing utilities.

mmfutils.testing.**allclose**(*a*, *b*, *use_covariance=False*, ***kw*)

Return *True* if a and be are close.

Like np.allclose, but first tries a strict equality test, and also works for quantities with uncertainties.

Parameters **use_covariance** (*bool, float*) – If True and parameters have uncertainties, then use their covariance information. Two parameters are considered equal in this case if their difference is zero to within the factor use_covariance times the std_dev of the difference. (If use_covariance is True, this is 1 standard deviation, but floats can be used.)

CHAPTER

TWO

MMF UTILS

Small set of utilities: containers and interfaces.

This package provides some utilities that I tend to rely on during development. Presently it includes some convenience containers, plotting tools, and a patch for including zope.interface documentation in a notebook.

(Note: If this file does not render properly, try viewing it through nbviewer.org)

Documentation: http://mmfutils.readthedocs.org

Source: https://bitbucket.org/mforbes/mmfutils

Issues: https://bitbucket.org/mforbes/mmfutils/issues

Build Status:

Main

Fork

Table of Contents

2.1 Installing

This package can be installed from from the bitbucket project:

```
pip install hg+https://bitbucket.org/mforbes/mmfutils
```

CHAPTER

THREE

USAGE

3.1 Containers

3.1.1 ObjectBase and Object

The `ObjectBase` and `Object` classes provide some useful features described below. Consider a problem where a class is defined through a few parameters, but requires extensive initialization before it can be properly used. An example is a numerical simulation where one passes the number of grid points N and a length L, but the initialization must generate large grids for efficient use later on. These grids should be generated before computations begin, but should not be re-generated every time needed. They also should not be pickled when saved to disk.

Deferred initialization via the ``init()`` method: The idea here changes the semantics of `__init__()` slightly by deferring any expensive initialization to `init()`. Under this scheme, `__init__()` should only set and check what we call picklable attributes: these are parameters that define the object (they will be pickled in `Object` below) and will be stored in a list `self.picklable_attributes` which is computed at the end of `ObjectBase.__init__()` as the list of all keys in `__dict__`. Then, `ObjectBase.__init__()` will call `init()` where all remaining attributes should be calculated.

This allows users to change various attributes, then reinitialize the object once with an explicit call to `init()` before performing expensive computations. This is an alternative to providing complete properties (getters and setters) for objects that need to trigger computation. The use of setters is safer, but requires more work on the side of the developer and can lead to complex code when different properties depend on each other. The approach here puts all computations in a single place. Of course, the user must remember to call `init()` before working with the object.

To facilitate this, we provide a mild check in the form of an `initialized` flag that is set to `True` at the end of the base `init()` chain, and set to `False` if any variables are in `pickleable_attributes` are set.

Serialization and Deferred Initialization: The base class `ObjectBase` does not provide any pickling services but does provide a nice representation. Additional functionality is provided by `Object` which uses the features of `ObjectBase` to define `__getstate__()` and `__setstate__()` methods for pickling which pickle only the `picklable_attributes`. Note: unpickling an object will **not** call `__init__()` but will call `init()` giving objects a chance to restore the computed attributes from pickles.

- **Note:** *Before using, consider if these features are really needed – with all such added functionality comes additional potential failure modes from side-interactions. The ``ObjectBase`` class is quite simple, and therefore quite safe, while ``Object`` adds additional functionality with potential side-effects. For example, a side-effect of support for pickles is that ``copy.copy()`` will also invoke ``init()`` when copying might instead be much faster. Thus, we recommend only using ``ObjectBase`` for efficient code.*

Object Example

```
ROOTDIR = !hg root
ROOTDIR = ROOTDIR[0]
import sys;sys.path.insert(0, ROOTDIR)

import numpy as np

from mmfutils.containers import ObjectBase, ObjectMixin

class State(ObjectBase):
    _quiet = False
    def __init__(self, N, L=1.0, **kw):
        """Set all of the picklable parameters, in this case, N and L."""
        self.N = N
        self.L = L

        # Now register these and call init()
        super().__init__(**kw)
        if not self._quiet:
            print("__init__() called")

    def init(self):
        """All additional initializations"""
        if not self._quiet:
            print("init() called")
        dx = self.L / self.N
        self.x = np.arange(self.N, dtype=float) * dx - self.L/2.0
        self.k = 2*np.pi * np.fft.fftfreq(self.N, dx)

        # Set highest momentum to zero if N is even to
        # avoid rapid oscillations
        if self.N % 2 == 0:
            self.k[self.N//2] = 0.0

        # Calls base class which sets self.initialized
        super().init()

    def compute_derivative(self, f):
        """Return the derivative of f."""
        return np.fft.ifft(self.k*1j*np.fft.fft(f)).real

s = State(256)
print(s)  # No default value for L
```

```
init() called
__init__() called
State(L=1.0, N=256)
```

```
s.L = 2.0
print(s)
```

```
State(L=2.0, N=256)
```

One feature is that a nice `repr()` of the object is produced. Now let's do a calculation:

```
f = np.exp(3*np.cos(2*np.pi*s.x/s.L)) / 15
df = -2.*np.pi/5.*np.exp(3*np.cos(2*np.pi*s.x/s.L))*np.sin(2*np.pi*s.x/s.L)/s.L
np.allclose(s.compute_derivative(f), df)
```

```
False
```

Oops! We forgot to reinitialize the object... (The formula is correct, but the lattice is no longer commensurate so the FFT derivative has huge errors).

```
print(s.initialized)
s.init()
assert s.initialized
f = np.exp(3*np.cos(2*np.pi*s.x/s.L)) / 15
df = -2.*np.pi/5.*np.exp(3*np.cos(2*np.pi*s.x/s.L))*np.sin(2*np.pi*s.x/s.L)/s.L
np.allclose(s.compute_derivative(f), df)
```

```
False
init() called
```

```
True
```

Here we demonstrate pickling. Note that using `Object` makes the pickles very small, and when unpickled, `init()` is called to re-establish `s.x` and `s.k`. Generally one would inherit from `Object`, but since we already have a class, we can provide pickling functionality with `ObjectMixin`:

```
class State1(ObjectMixin, State):
    pass

s = State(N=256, _quiet=True)
s1 = State1(N=256, _quiet=True)
```

```
import pickle, copy
s_repr = pickle.dumps(s)
s1_repr = pickle.dumps(s1)
print(f"ObjectBase pickle:  {len(s_repr)} bytes")
print(f"ObjectMixin pickle: {len(s1_repr)} bytes")
```

```
ObjectBase pickle:  4438 bytes
ObjectMixin pickle: 102 bytes
```

Note, however, that the speed of copying is significantly impacted:

```
%timeit copy.copy(s)
%timeit copy.copy(s1)
```

```
3.55 µs ± 669 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
43.9 µs ± 4.95 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Another use case applies when `init()` is expensive. If x and k were computed in `__init__()`, then using properties to change both N and L would trigger two updates. Here we do the updates, then call `init()`. Good practice is to call `init()` automatically before any serious calculation to ensure that the object is brought up to date before the computation.

```
s.N = 64
s.L = 2.0
s.init()
```

Finally, we demonstrate that `Object` instances can be archived using the `persist` package:

```
import persist.archive
a = persist.archive.Archive(check_on_insert=True)
a.insert(s=s)

d = {}
exec(str(a), d)

d['s']
```

```
State(L=2.0, N=64, _quiet=True)
```

3.1.2 Container

The `Container` object is a slight extension of `Object` that provides a simple container for storing data with attribute and iterative access. These implement some of the Collections Abstract Base Classes from the python standard library. The following containers are provided:

- `Container`: Bare-bones container extending the `Sized`, `Iterable`, and `Container` abstract ase classes (ABCs) from the standard `containers` library.
- `ContainerList`: Extension that acts like a tuple/list satisfying the `Sequence` ABC from the `containers` library (but not the `MutableSequence` ABC. Although we allow setting and deleting items, we do not provide a way for insertion, which breaks this interface.)
- `ContainerDict`: Extension that acts like a dict satisfying the `MutableMapping` ABC from the `containers` library.

These were designed with the following use cases in mind:

- Returning data from a function associating names with each data. The resulting `ContainerList` will act like a tuple, but will support attribute access. Note that the order will be lexicographic. One could use a dictionary, but attribute access with tab completion is much nicer in an interactive session. The `containers.nametuple` generator could also be used, but this is somewhat more complicated (though might be faster). Also, named tuples are immutable - here we provide a mutable object that is picklable etc. The choice between `ContainerList` and `ContainerDict` will depend on subsequent usage. Containers can be converted from one type to another.

Container Examples

```
from mmfutils.containers import Container

c = Container(a=1, c=2, b='Hi there')
print(c)
print(tuple(c))
```

```
Container(a=1, b='Hi there', c=2)
(1, 'Hi there', 2)
```

```
# Attributes are mutable
c.b = 'Ho there'
print(c)
```

```
Container(a=1, b='Ho there', c=2)
```

```
# Other attributes can be used for temporary storage but will not be pickled.
import numpy as np

c.large_temporary_array = np.ones((256,256))
print(c)
print(c.large_temporary_array)
```

```
Container(a=1, b='Ho there', c=2)
[[1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]
 ...
 [1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]]
```

```
import pickle
c1 = pickle.loads(pickle.dumps(c))
print(c1)
c1.large_temporary_array
```

```
Container(a=1, b='Ho there', c=2)
```

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

AttributeError                            Traceback (most recent call last)

<ipython-input-13-bd53d5116502> in <module>
      2 c1 = pickle.loads(pickle.dumps(c))
      3 print(c1)
----> 4 c1.large_temporary_array


AttributeError: 'Container' object has no attribute 'large_temporary_array'
```

3.2 Contexts

The `mmfutils.contexts` module provides two useful contexts:

`NoInterrupt`: This can be used to susspend `KeyboardInterrupt` exceptions until they can be dealt with at a point that is convenient. A typical use is when performing a series of calculations in a loop. By placing the loop in a `NoInterrupt` context, one can avoid an interrupt from ruining a calculation:

```
from mmfutils.contexts import NoInterrupt

complete = False
```

(continues on next page)

(continued from previous page)

```
n = 0
with NoInterrupt() as interrupted:
    while not complete and not interrupted:
        n += 1
        if n > 10:
            complete = True
```

Note: One can nest `NoInterrupt` contexts so that outer loops are also interrupted. Another use-case is mapping. See doc/Animation.ipynb for more examples.

```
res = NoInterrupt().map(abs, range(-100, 100))
np.sign(res)
```

```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1])
```

3.3 Interfaces

The interfaces module collects some useful zope.interface tools for checking interface requirements. Interfaces provide a convenient way of communicating to a programmer what needs to be done to used your code. This can then be checked in tests.

```
from mmfutils.interface import Interface, Attribute, verifyClass, verifyObject,␣
↪implementer

class IAdder(Interface):
    """Interface for objects that support addition."""

    value = Attribute('value', "Current value of object")

    # No self here since this is the "user" interface
    def add(other):
        """Return self + other."""
```

Here is a broken implementation. We muck up the arguments to `add`:

```
@implementer(IAdder)
class AdderBroken(object):
    def add(self, one, another):
        # There should only be one argument!
        return one + another

try:
    verifyClass(IAdder, AdderBroken)
except Exception as e:
    print("{0.__class__.__name__}: {0}".format(e))
```

```
BrokenMethodImplementation: The implementation of add violates its contract
        because implementation requires too many arguments.
```

Now we get `add` right, but forget to define `value`. This is only caught when we have an object since the attribute is supposed to be defined in `__init__()`:

```
@implementer(IAdder)
class AdderBroken(object):
    def add(self, other):
        return one + other

# The class validates...
verifyClass(IAdder, AdderBroken)

# ... but objects are missing the value Attribute
try:
    verifyObject(IAdder, AdderBroken())
except Exception as e:
    print("{0.__class__.__name__}: {0}".format(e))
```

```
BrokenImplementation: An object has failed to implement interface <InterfaceClass __
↪main__.IAdder>

        The value attribute was not provided.
```

Finally, a working instance:

```
@implementer(IAdder)
class Adder(object):
    def __init__(self, value=0):
        self.value = value
    def add(self, other):
        return one + other

verifyClass(IAdder, Adder) and verifyObject(IAdder, Adder())
```

```
True
```

3.3.1 Interface Documentation

We also monkeypatch `zope.interface.documentation.asStructuredText()` to provide a mechanism for documentating interfaces in a notebook.

```
from mmfutils.interface import describe_interface
describe_interface(IAdder)
```

3.4 Parallel

The `mmfutils.parallel` module provides some tools for launching and connecting to IPython clusters. The `parallel.Cluster` class represents and controls a cluster. The cluster is specified by the profile name, and can be started or stopped from this class:

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.INFO)
import numpy as np
from mmfutils import parallel
cluster = parallel.Cluster(profile='default', n=3, sleep_time=1.0)
cluster.start()
cluster.wait()  # Instance of IPython.parallel.Client
view = cluster.load_balanced_view
x = np.linspace(-6, 6, 100)
y = view.map(lambda x:x**2, x)
print(np.allclose(y, x**2))
cluster.stop()
```

```
Waiting for connection file: ~/.ipython/profile_default/security/ipcontroller-client.
↪json
```

```
INFO:root:Starting cluster: ipcluster start --daemonize --quiet --profile=default --
↪n=3
```

```
Waiting for connection file: ~/.ipython/profile_default/security/ipcontroller-client.
↪json
```

```
INFO:root:waiting for 3 engines
INFO:root:0 of 3 running
INFO:root:3 of 3 running
INFO:root:Stopping cluster: ipcluster stop --profile=default
```

```
True
Waiting for connection file: ~/.ipython/profile_default/security/ipcontroller-client.
↪json
```

If you only need a cluster for a single task, it can be managed with a context. Be sure to wait for the result to be computed before exiting the context and shutting down the cluster!

```
with parallel.Cluster(profile='default', n=3, sleep_time=1.0) as client:
    view = client.load_balanced_view
    x = np.linspace(-6, 6, 100)
    y = view.map(lambda x:x**2, x, block=True)  # Make sure to wait for the result!
print(np.allclose(y, x**2))
```

```
Waiting for connection file: ~/.ipython/profile_default/security/ipcontroller-client.
↪json
```

```
INFO:root:Starting cluster: ipcluster start --daemonize --quiet --profile=default --
↪n=3
```

```
Waiting for connection file: ~/.ipython/profile_default/security/ipcontroller-client.
↪json
```

```
INFO:root:waiting for 3 engines
INFO:root:0 of 3 running
INFO:root:3 of 3 running
INFO:root:Stopping cluster: ipcluster stop --profile=default
```

```
Waiting for connection file: ~/.ipython/profile_default/security/ipcontroller-client.
↪json
True
```

If you just need to connect to a running cluster, you can use `parallel.get_client()`.

3.5 Performance

The `mmfutils.performance` module provides some tools for high performance computing. Note: this module requires some additional packages including numexp, pyfftw, and the `mkl` package installed by anaconda. Some of these require building system libraries (i.e. the FFTW). However, the various components will not be imported by default.

Here is a brief description of the components:

- `mmfutils.performance.blas`: Provides an interface to a few of the scipy BLAS wrappers. Very incomplete (only things I currently need).
- `mmfutils.performance.fft`: Provides an interface to the FFTW using `pyfftw` if it is available. Also enables the planning cache and setting threads so you can better control your performance.
- `mmfutils.performance.numexpr`: Robustly imports numexpr and disabling the VML. (If you don't do this carefully, it will crash your program so fast you won't even get a traceback.)
- `mmfutils.performance.threads`: Provides some hooks for setting the maximum number of threads in a bunch of places including the MKL, numexpr, and fftw.

3.6 Plotting

Several tools are provided in `mmfutils.plot`:

3.6.1 Fast Filled Contour Plots

`mmfutils.plot.imcontourf` is similar to matplotlib's `plt.contourf` function, but uses `plt.imshow` which is much faster. This is useful for animations and interactive work. It also supports my idea of saner array-shape processing (i.e. if `x` and `y` have different shapes, then it will match these to the shape of `z`). Matplotlib now provies `plt.pcolourmesh` which is similar, but has the same interface issues.

```
%matplotlib inline
from matplotlib import pyplot as plt
import time
import numpy as np
from mmfutils import plot as mmfplt
x = np.linspace(-1, 1, 100)[:, None]**3
```

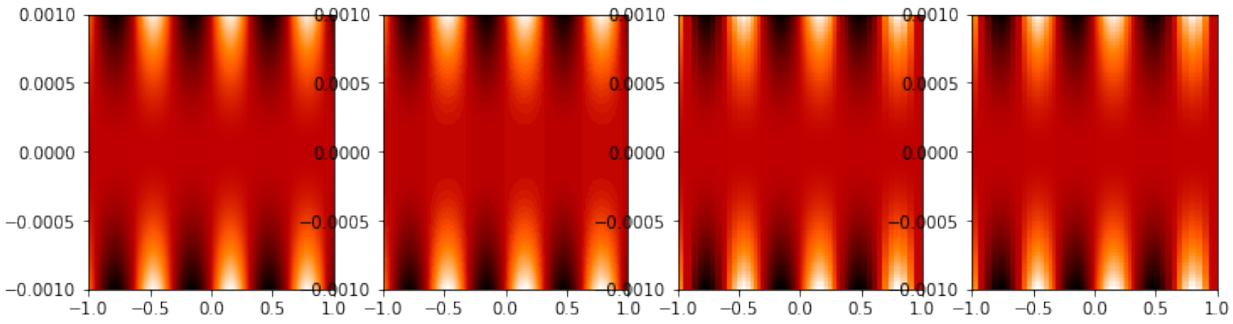
(continues on next page)

(continued from previous page)

```
y = np.linspace(-0.1, 0.1, 200)[None, :]**3
z = np.sin(10*x)*y**2
plt.figure(figsize=(12,3))
plt.subplot(141)
%time mmfplt.imcontourf(x, y, z, cmap='gist_heat')
plt.subplot(142)
%time plt.contourf(x.ravel(), y.ravel(), z.T, 50, cmap='gist_heat')
plt.subplot(143)
%time plt.pcolor(x.ravel(), y.ravel(), z.T, cmap='gist_heat')
plt.subplot(144)
%time plt.pcolormesh(x.ravel(), y.ravel(), z.T, cmap='gist_heat')
```

```
CPU times: user 9.53 ms, sys: 2.13 ms, total: 11.7 ms
Wall time: 11.7 ms
CPU times: user 36.1 ms, sys: 771 µs, total: 36.8 ms
Wall time: 37.1 ms
CPU times: user 251 ms, sys: 34 ms, total: 285 ms
Wall time: 265 ms
CPU times: user 3.6 ms, sys: 119 µs, total: 3.72 ms
Wall time: 3.73 ms
```

```
<matplotlib.collections.QuadMesh at 0x104a8e850>
```



3.7 Angular Variables

A couple of tools are provided to visualize angular fields, such as the phase of a complex wavefunction.

```
%matplotlib inline
from matplotlib import pyplot as plt
import time
import numpy as np
from mmfutils import plot as mmfplt
x = np.linspace(-1, 1, 100)[:, None]
y = np.linspace(-1, 1, 200)[None, :]
z = x + 1j*y

plt.figure(figsize=(9,2))
ax = plt.subplot(131)
mmfplt.phase_contour(x, y, z, colors='k', linewidths=0.5)
ax.set_aspect(1)
```

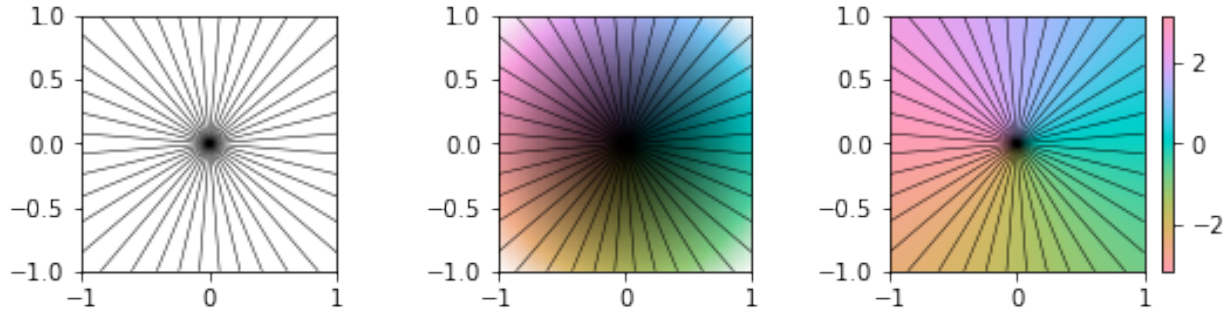
(continues on next page)

(continued from previous page)

```
# This is a little slow but allows you to vary the luminosity.
ax = plt.subplot(132)
mmfplt.imcontourf(x, y, mmfplt.colors.color_complex(z))
mmfplt.phase_contour(x, y, z, linewidths=0.5)
ax.set_aspect(1)

# This is faster if you just want to show the phase and allows
# for a colorbar via a registered colormap
ax = plt.subplot(133)
mmfplt.imcontourf(x, y, np.angle(z), cmap='huslp')
ax.set_aspect(1)
plt.colorbar()
mmfplt.phase_contour(x, y, z, linewidths=0.5)
```

```
(<matplotlib.contour.QuadContourSet at 0x1a19181690>,
 <matplotlib.contour.QuadContourSet at 0x1a19181ad0>)
```



3.8 Debugging

A couple of debugging tools are provided. The most useful is the `debug` decorator which will store the local variables of a function in a dictionary or in your global scope.

```
from mmfutils.debugging import debug

@debug(locals())
def f(x):
    y = x**1.5
    z = 2/x
    return z

print(f(2.0), x, y, z)
```

```
1.0 2.0 2.8284271247461903 1.0
```

3.9 Mathematics

We include a few mathematical tools here too. In particular, numerical integration and differentiation. Check the API documentation for details.

CHAPTER

FOUR

DEVELOPER INSTRUCTIONS

If you are a developer of this package, there are a few things to be aware of.

1. If you modify the notebooks in `docs/notebooks` then you may need to regenerate some of the `.rst` files and commit them so they appear on bitbucket. This is done automatically by the `pre-commit` hook in `.hgrc` if you include this in your `.hg/hgrc` file with a line like:

```
%include ../.hgrc
```

Security Warning: if you do this, be sure to inspect the `.hgrc` file carefully to make sure that no one inserts malicious code.

This runs the following code:

```
!cd $ROOTDIR; jupyter nbconvert --to=rst --output=README.rst doc/README.ipynb
```

```
[NbConvertApp] Converting notebook doc/README.ipynb to rst
[NbConvertApp] Writing 36253 bytes to doc/README.rst
```

We also run a comprehensive set of tests, and the pre-commit hook will fail if any of these do not pass, or if we don't have complete code coverage. We run these tests in a conda environment that can be made using the makefile:

```
make envs
make test   # conda run -n _mmfutils pytest
```

To run these manually you could do:

```
cond activate _mmfutils
pytest
```

Here is an example:

```
!cd $ROOTDIR; conda activate _mmfutils; pytest -n4
```

Complete code coverage information is provided in `build/_coverage/index.html`.

```
from IPython.display import HTML
with open(os.path.join(ROOTDIR, 'build/_coverage/index.html')) as f:
    coverage = f.read()
HTML(coverage)
```

4.1 Releases

We try to keep the repository clean with the following properties:

1. The default branch is stable: i.e. if someone runs `hg clone`, this will pull the latest stable release.
2. Each release has its own named branch so that e.g. `hg up 0.5.0` will get the right thing. Note: this should update to the development branch, *not* the default branch so that any work committed will not pollute the development branch (which would violate the previous point).

To do this, we advocate the following proceedure.

1. **Update to Correct Branch**: Make sure this is the correct development branch, not the default branch by explicitly updating:

   ```
   hg up <version>
   ```

   (Compare with `hg up default` which should take you to the default branch instead.)

2. **Work**: Do your work, committing as required with messages as shown in the repository with the following keys:

   - `DOC`: Documentation changes.
   - `API`: Changes to the exising API. This could break old code.
   - `EHN`: Enhancement or new functionality. Without an `API` tag, these should not break existing codes.
   - `BLD`: Build system changes (`setup.py`, `requirements.txt` etc.)
   - `TST`: Update tests, code coverage, etc.
   - `BUG`: Address an issue as filed on the issue tracker.
   - `BRN`: Start a new branch (see below).
   - `REL`: Release (see below).
   - `WIP`: Work in progress. Do not depend on these! They will be stripped. This is useful when testing things like the rendering of documentation on bitbucket etc. where you need to push an incomplete set of files. Please collapse and strip these eventually when you get things working.
   - `CHK`: Checkpoints. These should not be pushed to bitbucket!

3. **Tests**: Make sure the tests pass.

   ```
   conda env update --file environment.yml
   conda activate _mmfutils; pytest
   ```

   (`hg com` will automatically run tests after pip-installing everything in `setup.py` if you have linked the `.hgrc` file as discussed above, but the use of independent environments is preferred now.)

4. **Update Docs**: Update the documentation if needed. To generate new documentation run:

   ```
   cd doc
   sphinx-apidoc -eTE ../mmfutils -o source
   rm source/mmfutils.*tests*
   ```

   - Include any changes at the bottom of this file (`doc/README.ipynb`).
   - You may need to copy new figures to `README_files/` if the figure numbers have changed, and then `hg add` these while `hg rm` the old ones.

   Edit any new files created (titles often need to be added) and check that this looks good with

```
make html
open build/html/index.html
```

Look especially for errors of the type "WARNING: document isn't included in any toctree". This indicates that you probably need to add the module to an upper level `.. toctree::`. Also look for "WARNING: toctree contains reference to document u'...' that doesn't have a title: no link will be generated". This indicates you need to add a title to a new file. For example, when I added the `mmf.math.optimize` module, I needed to update the following:

```
.. doc/source/mmfutils.rst
mmfutils
========

.. toctree::
    ...
    mmfutils.optimize
    ...
```

```
.. doc/source/mmfutils.optimize.rst
mmfutils.optimize
=================

.. automodule:: mmfutils.optimize
    :members:
    :undoc-members:
    :show-inheritance:
```

5. **Clean up History**: Run `hg histedit`, `hg rebase`, or `hg strip` as needed to clean up the repo before you push. Branches should generally be linear unless there is an exceptional reason to split development.

6. **Release**: First edit `mmfutils/__init__.py` to update the version number by removing the `dev` part of the version number. Commit only this change and then push only the branch you are working on:

   ```
   hg com -m "REL: <version>"
   hg push -b .
   ```

7. **Pull Request**: Create a pull request on the development fork from your branch to `default` on the release project bitbucket. Review it, fix anything, then accept the PR and close the branch.

8. **Publish on PyPI**: Publish the released version on PyPI using twine

   ```
   # Build the package.
   python setup.py sdist bdist_wheel

   # Test that everything looks right:
   twine upload --repository-url https://test.pypi.org/legacy/ dist/*

   # Upload to PyPI
   twine upload dist/*
   ```

9. **Build Conda Package**: This will run all the tests in a fresh environment as specified by `meta.yaml`. Make sure that the dependencies in `meta.yaml`, `environment.yml`, and `setup.py` are consistent. Note that the list of versions to be built is specified in `conda_build_config.yaml`.

   ```
   conda build .
   conda build . --output    # Use this below
   ```

(continues on next page)

(continued from previous page)

```
anaconda login
anaconda upload --all /data/apps/conda/conda-bld/noarch/mmfutils-0.5.0-py_0.tar.
↪bz2
```

10. **Start new branch**: On the same development branch (not `default`), increase the version number in `mmfutils/__init__.py` and add `dev`: i.e.:

    version = '0.5.1dev'

Then create this branch and commit this:

```
hg branch "0.5.1"
hg com -m "BRN: Started branch 0.5.1"
```

11. Update MyPI index.
12. Optional: Update any `setup.py` files that depend on your new features/fixes etc.

CHAPTER

FIVE

CHANGE LOG

5.1 REL: 0.5.1

API changes: * Split `mmfutils.containers.Object` into `ObjectBase` which is simple and `ObjectMixin` which provides the picking support. Demonstrate in docs how the pickling can be useful, but slows copying.

5.2 REL: 0.5.0

API changes: * Python 3 support only. * `mmfutils.math.bases.interface` renamed to `mmfutils.math.bases.interfaces`. * Added default class-variable attribute support to e`mmfutils.containers.Object`. * Minor enhancements to `mmfutils.math.bases.PeriodicBasis` to enhance GPU support. * Added `mmfutils.math.bases.interfaces.IBasisLz` and support in `mmfutils.math.bases.bases.PeriodicBasis` for rotating frames. * Cleanup of build environment and tests. * Single environment `_mmfutils` now used for testing and documentation.

5.3 REL: 0.4.13

API changes:

- Use `@implementer()` class decorator rather than `classImplements` or `implements` in all interfaces.
- Improve `NoInterrupt` context. Added `NoInterrupt.unregister()`: this allows `NoInterrupt` to work in a notebook cell even when the signal handlers are reset. (But only works in that one cell.)
- Added Abel transform `integrate2` to Cylindrical bases.

Issues: * Resolved issue #22: Masked arrays work with `imcontourf` etc. * Resolved issue #23: `NoInterrupt` works well except in notebooks due to ipykernel issue #328. * Resolved issue #24: Python 3 is now fully supported and tested.

5.4 REL: 0.4.10

API changes:

- Added `contourf`, `error_line`, and `ListCollections` to `mmfutils.plot`.
- Added Python 3 support (still a couple of issues such as `mmfutils.math.integrate.ssum_inline`.)
- Added `mmf.math.bases.IBasisKx` and update `lagrangian` in bases to accept `k2` and `kx2` for modified dispersion control (along x).
- Added `math.special.ellipkinv`.
- Added some new `mmfutils.math.linalg` tools.

Issues:

- Resolved issue #20: `DyadicSum` and `scipy.optimize.nonlin.Jacobian`
- Resolved issue #22: imcontourf now respects masked arrays.
- Resolved issue #24: Support Python 3.

5.5 REL: 0.4.9

< *incomplete* >

5.6 REL: 0.4.7

API changes:

- Added `mmfutils.interface.describe_interface()` for inserting interfaces into documentation.
- Added some DVR basis code to `mmfutils.math.bases`.
- Added a diverging colormap and some support in `mmfutils.plot`.
- Added a Wigner Ville distribution computation in `mmfutils.math.wigner`
- Added `mmfutils.optimize.usolve` and `ubrentq` for finding roots with `` `uncertanties <https://pythonhosted.org/uncertainties/>`__ `` support.

Issues:

- Resolve issue #8: Use `` `ipyparallel <https://github.com/ipython/ipyparallel>`__ `` now.
- Resolve issue #9: Use pytest rather than `nose` (which is no longer supported).
- Resolve issue #10: PYFFTW wrappers now support negative `axis` and `axes` arguments.
- Address issue #11: Preliminary version of some DVR basis classes.
- Resolve issue #12: Added solvers with `` `uncertanties <https://pythonhosted.org/uncertainties/>`__ `` support.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

D

F

G

H

I

L

M

N

O

P

Q

R

S

U

V

W