Python-Parameters Documentation

November 12, 2014

Keeping track of parameters during simulations can often be cumbersome, especially if those parameters are time- or co- dependent, or if unit conversions are necessary. The `python-parameters` module solves this problem by acting as a central repository of all parameters, their non-dimensionalisation, their interdependencies, their units and their bounds. While providing all of this functionality, `python-parameters` also attempts to maintain minimal overhead so that it is suitable for use in problems requiring iteration (such as numerical integration).

Properly enumerated, the `python-parameters` module can:

- Act as a central location for the storage and retrieval of model parameters.
- Keep track of parameter units, and perform unit conversions where that makes sense.
- Perform non-dimensionalisation of parameters in a consistent fashion to allow for simple model simulations.
- Allow parameters to inter-depend upon one another, and ensure that parameters do not depend upon one another in unresolvable ways.
- Allow parameters to depend on as-yet undeclared runtime values (such as integration time).
- Keep track of limits for parameter values, to ensure parameters do not escape pre-defined parameter ranges.
- Provide this functionality with minimal overhead to allow for speedy simulations.

As of version 1.1.9 (the version at time of publication); simple parameter storage and retrieval is only a factor of ≈15 slower than a simple parameter set and read; though speeds decrease depending upon how many of the more sophisticated features are used (such as parameter bounding).

All features are documented and unittested.

1 Installation

In most cases, installing this module is as easy as:

```
1 $ python2 setup.py install
```

If you run Arch Linux, you can instead run:

```
1 $ makepkg -i
```

2 Usage Overview

In this section, the syntax of `python-parameters` for each major feature is shown, along with simple examples of use within the python interpreter.

For most users, there are only two classes defined in `python-parameters` that are likely to be interest: `Parameters` and `SIQuantity`. Instances of `Parameters` class do the heavy lifting of managing parameters, whereas `SIQuantity` is the base type of physical quantities (i.e. it represents physical quantities with their units) and handles unit conversions. Both of these classes can be imported from the parameters module using:

```
>>> from parameters import SIQuantity, Parameters
```

2.1 SIQuantity

Instantiating an `SIQuantity` class is as simple as calling it with two arguments: a number and some units. For example:

```
>>> q = SIQuantity(3,'m*s')

>>> q
3 m*s
```

The "number" can be any integer, float or numpy array (or types conforming to standard python numeric operations). The "units" can be any SI unit combined with any SI prefix. A complete listing of units and prefixes is provided in tables 1 and 2 respectively. Custom units, or unit redefinitions, are also possible; see the source code for more details. Note that the long-form of unit names (rather than their abbreviation) can also be used:

```
>>> q = SIQuantity(3,'meter*second')

>>> q
3 m*s
```

Using prefixes with the base units is as simple as prepending them to the unit:

```
>>> SIQuantity(3,'km*ns')
3 km*ns
```

Arithmetic with `SIQuantity` instances is very simple, and support is provided for unit conversion, addition, subtraction, multiplication, division, raising to some power and equality tests. In cases where the operation does not make sense, an exception is raised with details as to why. Examples are now provided for each of these operations in turn.

Unit Names	Abbreviations	Dimensions
Fundamental SI units		
constant	non-dim	-
metre,meter	m	L
second	s	T
gram	g	M
ampere	A	I
kelvin	K	Θ
mole	mol	N
candela	cd	J
dollar	$	$
Imperial scales		
mile	mi	L
yard	yd	L
foot	ft	L
inch	in	L
point	pt	L
mmHg	mmHg	MLT^{-2}
Length scales		
angstrom	Å	L
astronomical unit	au	L
lightyear	ly	L
Volumes		
litre,liter	L	L^3
gallon	gal	L^3
quart	qt	L^3
Time scales		
year	year	T
day	day	T
hour	h	T
minute	min	T
hertz	Hz	T^{-1}
Force		
newton	N	MLT^{-2}
Pressure		
atm	atm	$ML^{-1}T^{-2}$
bar	bar	$ML^{-1}T^{-2}$
pascal	Pa	$ML^{-1}T^{-2}$
psi	psi	$ML^{-1}T^{-2}$
Energy & Power		
joule	J	ML^2T^{-2}
calorie	cal	ML^2T^{-2}
electronvolt	eV	ML^2T^{-2}
watt	W	ML^2T^{-3}
Electromagnetism		
coulomb	C	IT
farad	F	$T^4I^2L^{-2}M^{-1}$
henry	H	$ML^2T^{-2}I^{-2}$
volt	V	$ML^2I^{-1}T^{-3}$
ohm		$ML^2I^{-2}T^{-3}$
siemens	mho	$M^{-1}L^{-2}T^3I^2$
tesla	T	$MI^{-1}T^{-2}$
gauss	G	$MI^{-1}T^{-2}$
weber	Wb	$L^2MT^{-2}I^{-1}$

Table 1: Supported units

Larger Prefixes	kilo	mega	giga	tera	peta	exa	zepto	yotta
Abbreviation	k	M	G	T	P	E	Z	Y
Scaling (10^x)	3	6	9	12	15	18	21	24
Smaller Prefixes	milli	micro	nano	pico	femto	atto	zepto	yocto
Abbreviation	m	{mu}	n	p	f	a	z	y
Scaling (10^x)	-3	-6	-9	-12	-15	-18	-21	-24

Table 2: Supported prefixes

```
>>> SIQuantity(3,'m/s^2')('in/ns^2') # Unit Conversion
1.1811023622e-16 ns/ns^2

>>> SIQuantity(3,'m')+SIQuantity(2,'mm') # Addition
3.002 m

>>> SIQuantity(3,'kJ')-SIQuantity(2,'cal') # Subtraction
2.9916264 kJ

>>> SIQuantity(3.,'mT') / SIQuantity(10,'s') # Division
0.3 mT/s

>>> 2*SIQuantity(3.,'mT') # Scaling
6.0 mT

>>> SIQuantity(2.6,'A')**2 # Squaring
6.76 A^2

>>> SIQuantity(1,'W*h') == SIQuantity(1e-9,'GW*h')
True

>>> SIQuantity(3,'m') + SIQuantity(2,'s') # Nonsense addition
UnitConversionError: Invalid conversion. Units 's' and 'm' do not match.
```

In each of the above examples, the returned object is another `SIQuantity` object, which can also undergo further arithmetic operation. For convenience, it is not necessary to manually instantiate all physical quantities into `SIQuantity` objects; provided that at least one of the operands involved in the calcuation is already an `SIQuantity` object. For example, the addition example above could be simplified to:

```
>>> SIQuantity(3,'m')+(2,'mm') # Addition
3.002 m

>>> (3,'m')+SIQuantity(2,'mm') # Addition
3002.0 mm
```

Be careful when using this contraction that order of operations does not cause the tuple to perform some undefined operation with something else. For example:

```
>>> 2*(3,'m')+SIQuantity(2,'mm') # Addition
This becomes (3,'m',3,'m') + SIQuantity(2,'mm'), which is ill-defined.
```

The `Quantity` class, which is the base class for the `SIQuantity` class is quite flexible, and if you want to implement a non-SI system, or add custom units, look at the source code for `SIQuantity` and `SID`, which should tell you everything you need to know.

2.2 Parameters

The `Parameters` class handles the heavy lifting of managing named parameters, their values, relationships and conversions; using `Quantity` objects to represent physical quantities. Initialising a `Parameters` object is as simple as:

```
>>> p = Parameters()
```

There are three optional arguments that can be passed to this initialiser:

- `default_scaled=True`: Whether parameters should return non-dimensionalised values instead of `Quantity` objects. By default, this is `True`. This can be negated at runtime.
- `constants=False`: Whether to populate the parameter namespaces with various physical constants, all of which are prefixed with "c_" (e.g. `c_hbar`). By default, this is `False`.
- `dispenser=None`: A custom `UnitDispenser` to use instead of the standard SI one. Most users will not need to touch this.

The tasks you might want to achieve with parameters within a `Parameters` instance loosely fall into three categories: definition, extraction, and transformation; which are dealt with in the next three subsubsections. There are more advanced ways of interacting with a `Parameters` instance which are beyond the scope of this overview. More information can be found by using the inbuild `help` facility built into Python on each of `Parameters'` methods.

2.2.1 Parameter Definition

Setting a parameter value can be done in several ways, with varying degress of flexibility.

If you want to set a parameter to a static value (i.e. one that does not actively depend upon later values of other parameters) you can use the following methods:

```
Set an attribute using: p.<param_name> = <param_value>
This method does not ``interpret'' the parameter value (explained later).
>>> p.x = 1
>>> p.y = (1,'cm')
Equivalently, you can set the parameter using a function call,
which has the benefit of being able to define multiple parameters at
once: p(<param_name_1>=<param_value_1>,....,<param_name_n>=<param_value_n>,)
This method _does_ ``interpret'' the parameter value.
>>> p(x=1,y=(1,'cm'))
Finally, in a way that is also not ``interpreted'', you can set a dictionary
of parameter values using:
>>> p << {'x':1, 'y':(1,'cm')}
```

Parameter names can be any legal python variable name (except those starting with an underscore). Parameter values can be any inbuilt number type (including `complex`), any function/lambda object (or a string representing a mathematical expression which is then converted to a function), any tuple of preceeding types with a unit, and any Quantity object.

The function/lambda support in the values for parameters is where the `Parameters` object garners much of its power. This allows parameters to depend on each other in potentially complex ways. For example:

```
>>> p(x=1,y=2,z=lambda x,y: x+y) # or p(....,z='x+y')
< Parameters with 3 definitions >
>>> p.z
3
```

In the above example, `x` and `y` are set to be 1 and 2 respectively, and `z` is set to be their sum. At this point, `z` is only statically defined as their *current* sum; were `x` or `y` to change, `z` would remain unchanged. This is because the parameter value for `z` was interpreted. To cause `z` to be a dynamic function of `x` and `y`, one must use a method that does not interpret the value:

```
>>> p(x=1,y=2)
< Parameters with 2 definitions >
>>> p.z = lambda x,y: x+y # Or equivalently: p.z = 'x+y' or p << {'z': lambda x,y: x+y}
>>> p.z
3
>>> p(x=3,y=1).z
4
```

`Parameters` objects will prevent functions from being added that cause parameters loops. That is, a parameter `x` cannot depend on a parameter `y` if parameter `y` directly or indirectly depends upon the value of `x`.

It is also possible to set parameters to depend on other parameters in an invertable way, so that when that parameter is directly changed, the underlying variables are updated self-consistently. This is achieved by setting a parameter equal to function/lambda object that has an extra option keyword argument of the same name. The function should return an inverse mapping for all dependent parameters as a list if z is specified. For example:

```
>>> p(x=1,y=2) << {'z':lambda x,y,z=None:x+y if z is None else [1,z-1]}
>>> p(z = 12) # Don't use p.z=12, because that leads to z no longer being a function
>>> p.x
1
>>> p.y
11
```

To see an overview of the parameters stored in a `Parameters` instance, and the relationships between them, simply run:

```
>>> p.show()
| Parameter |   Value    | Scaled |
|     x     | 1.0 units  |  1.0   |
|     y     | 11.0 units |  11.0  |
| z(x,y,z)  | 12.0 units |  12.0  |
```

To remove a parameter definition, simply use the `forget` method:

```
>>> p.forget('x','y','z') # You can add as many parameters as you like here.
< Parameters with 0 definitions >
```

Importantly, when you specify a parameter value, `Parameters` instances will record the units of the value specified as the units of that parameter. If you need to override these units, use:

```
>>> p.y = (1,'s')
>>> p & {'y':'ms'}
>>> p.y
1000 ms
```

Changing the scaling is only possible when converting to units with the same dimensions.

Throughout the use of the `Parameters` object, you may wish to ensure that certain parameters remain within certain bounds. You can set bounds on parameters using:

```
>>> p['x'] = (0,100)
This constrains the parameter x to be within 0-100 (inclusive)
>>> p['x'] = (0,None)
If None is used, it is taken to be + or - infinity depending on whether
it is the upper or lower bound. You can use this to create regions in which the
parameter cannot lie.
>>> p['x'] = [(None,10),(15,None)]
This causes any value of x to be permitted except values between 10 and 15 (inclusive).
```

For more advanced bounding, you use the `set_bounds` method, which takes three additional keyword arguments in addition to the bounds themselves: `error=True`, `clip=False`, and `inclusive=True`; each of which is boolean valued, with defaults as specified. If `error` is True, then when a parameter exits the bound, an error is raised (otherwise, a warning is raised). If `clip` is True, then `Parameters` will set the parameter to the nearest edge of allowed values, and warn the user if the `error` flag is true. If `inclusive` is True, bounds are considered to be inclusive. For example:

```
>>> p.set_bounds({'x':[(None,10),(15,None)]},inclusive=True,clip=True)
```

Bounds checking is computationally taxing, so if performance is a concern, we do not recommend using bounds.

If you want to temporarily define some parameters within some scope, and protect the existing definitions, `Parameters` objects support context management using the python `with` syntax. For example:

```
>>> p(x=1,y=2,z=3)
>>> with p:
...   p(z=4,a=10)
...   p('z','a')
...   # Do clever things with these parameters
>>> # z is now restored to its original value of 3; and a no longer exists within p.
```

Parameter definitions and values can be loaded and/or saved to a file using:

```
>>> p = Parameters.load('filename.py')
>>> p >> 'filename.py'
```

A sample file which can be loaded by Parameters is available in the source code repository.

2.2.2 Parameter Extraction

In the previous section, we have already seen one way of extracting parameter values; that is, using python object attributes.

```
>>> p.x = 1
>>> p.x
1
```

This method works except for when the variable name clashes with some method of the `Parameters` object; in which case the `Parameters` method will be returned.

The more general method for extracting parameters from a `Parameters` object is to specify parameters as arguments when calling the parameters object:

```
>>> p(x=1,y=2,z=3)
< Parameters with 3 definitions >
>>> p('x')
1
>>> p('x','y','z')
{'x':1, 'y':2, 'z':3}
```

As all arguments passed to the parameters object when calling are "interpreted", one can also pass functions, and retrieve its value:

```
>>> p(x=1,y=2,z=3)
< Parameters with 3 definitions >
>>> p('x+y+z')
6
>>> p('x+y','y-z')
{'x+y':3, 'y-z':-1}
```

If you want to temporarily override the values of some parameters during the evaluation of a variable, expression or function; simply pass them as keyword arguments during the call.

```
>>> p << {'x':1,'y':2,'z':'x+y'}
>>> p.z
3
>>> p('z',x=3,y=10)
13
>>> p('x','y','z',x=3)
{'x':3,'y':2,'z':5}
```

Lastly, if you want to extract the unitted value of a parameter, rather than its non-dimensional version (or vice versa if you used `default_scaled=False`), then simply prepend an underscore ('_') to the parameter name.

```
>>> p(x=(1,'m'),y=(2,'ns'),z=(3,'{mu}eV'))
< Parameters with 3 definitions >
>>> p('x','y','z')
{'x': 1, 'y': 2e-9, 'z': 4.806529461e-25}
>>> p('_x','_y','_z')
{'x': 1 m, 'y': 2 ns, 'z': 3 {mu}eV}
```

2.2.3 Parameter Transformations

The remainder of the functionality in the `Parameters` object involves various transformations of parameters.

Firstly, it is possible to change the scalings applied to yield the non-dimensional quantities. This allows, for example, for your non-dimension quantities to reflect the values of your parameters in natural units. This is achieved by explicitly changing the scaling of your parameters using:

```
>>> p.scaling(length=(2,'m'),current=(1,'mA')) # etc...
```

By default, the non-dimensional quantities are simply the numerical values of the parameters in SI base units; and so the above command cause non-dimensional values with dimensions $\propto L$ to be scaled by a factor of $\frac{1}{2}$. The current scaling for a dimension can be retrieved using:

```
>>> p.scaling('mass')
1 kg
>>> p.scaling('length','time') # Plus whichever other dimensions you care about
{'length': 2 m, 'time': 1 s}
```

Most of the physical quantities you will be using have composite units, and you can view the cumulative scaling relative to SI units for a quantity using:

```
>>> p.scaling(length=10)
>>> p.unit_scaling('J')
10.0
>>> p.unit_scaling('J','W')
{'J': 10.0, 'W': 10.0}
```

Another useful tool of the `Parameters` object is unit conversion of (potentially) non-`Quantity` objects. The syntax for this is:
`p.convert( <object>, input=<input units>, output=<output units>, value=<True/False> )`,
where `input` is the units that `Parameters` should assume the object has, `output` is the desired output units, and `value` (with default True) specifies whether or not you are only interested in its numerical value, or whether a `Quantity` object should be returned. If not specified, or equal to `None`, the input and output units are assumed to refer to the non-dimensional quantities used by the `Parameters` object. For example:

```
The following command converts 1 ns to a non-dimensional quantity:
>>> p.convert(1, input='ns')
1e-9

The following command converts 4 J to a value with units of calories:
>>> p.convert(4, input='J', output='cal')
0.9553835865099838

The following command converts 5 $/day to a Quantity with units of years:
>>> p.convert(5, input='$/day', output='$/year', value=False)
1826.21095 $/year

Note that this utility still works for Quantity units as well, in which
case the 'input' argument is ignored, and read from the Quantity object.
>>> p.convert(SIQuantity(1,'km'),output='m')
1000.0
```

The last big feature of `Parameters` is support for generating ranges of parameters. This is especially useful when it is necessary to iterate over a parameter value while exploring some parameters space. The syntax for this command is similar to the normal parameter extraction method; but is kept separate for clarity and efficiency. Let us first consider a basic example:

```
>>> p.range('x',x=[1,2,3,4])
[1,2,3,4]
```

In this trivial example, we see the basic syntax of the `range` method. Parameters of interest are indicated as arguments; and parameter ranges/overrides are specified as kwargs. Arrays of numbers may be specified using lists or numpy arrays. Let's consider something a little more complicated.

```
>>> p.y = lambda x: x**2
>>> p.range('y',x=[0,1,2,3,4])
[0.0, 1.0, 4.0, 9.0, 16.0]
```

We see here that the ranges specified as keyword arguments are treated as temporary overrides, just as they were in the parameter extraction methods; but now they are iterated over. Combining static overrides and ranges is easy:

```
>>> p.z = lambda x,y: (x+y)**2
>>> p.range('z',x=[0,1,2,3,4],y=10)
[100.0, 121.0, 144.0, 169.0, 196.0]
```

You can also have multiple ranges at once, but they must have the same length:

```
>>> p.z = lambda x,y: (x+y)**2
>>> p.range('x','z',x=[0,1,2,3,4],y=[10,11,12,13,14])
{'x': [0, 1, 2, 3, 4], 'z': [100.0, 144.0, 196.0, 256.0, 324.0]}
```

Note that united quantites are also supported in ranges (as tuples or `Quantity` objects).

`Parameters` also supports automatic expansion of parameter ranges, in which case parameter ranges are passed as keyword arguments as tuples with one of the following forms:

- (<start>,<stop>,<count>) ; which will generate a linear array from <start> to <stop> with <count> values.
- (<start>,<stop>,<count>,<sampler>) ; which is as above, but where the <sampler> is expected to generate the array. <sampler> can be a string (either 'linear','log','invlog' for linear, logarithmic, or inverse logarithmic distributions respectively).
- (arg1, arg2, arg3, ..., <function>) ; where there must be three or more arguments (which need not be used), the expansion of which will be: <function>(*args).

For example:

```
>>> p.z = lambda x,y: (x+y)**2
>>> p.range('x',x=(0,3,5))
[0.0, 0.75, 1.5, 2.25, 3.0]
>>> p.range('x',x=(0,3,5,'log'))
[0.0,
 5.304838230116769e-07,
 9.486532980505137e-05,
 0.01687023675571047,
 2.9999999970000002]
>>> p.range('z',x=(0,3,5,'log'),y=(0,3,5,'invlog'))
[8.9999999820000021,
 0.00028462278725051619,
 3.599772319608467e-08,
 0.00028462278725051619,
 8.9999999820000021]
```