Technische Universität München
Lehrstuhl für Informatik V
Dr. Tobias Neckel
M. Sc. Ionuț Farcaș

SS 2017

April 26, 2017

Algorithms for Uncertainty Quantification

Tutorial 1: Python overview

In this worksheet, we do a short overview of programming in `python` and we show how to install the uncertainty quantification library `chaospy`. Note: Everything that follows is a lot easier if you use a Unix-based operating system on your machine.
We assume that you are familiar with the basics concepts of computer programming:

- data types
- variables
- loops
- arrays
- functions
- etc.

If you are new to programming and you want to cover the basic concepts, see e.g. `tutorialspoint.com/computer_programming/computer_programming_basics.htm`.

Short overview of programming with python

Throughout these tutorials, we use `python` 2.7. How to use `python`[1]?

- directly in the command line: simply type `python`

[1] extensive `python` 2.7 documentation at `https://docs.python.org/2/index.html`

- via `Jupyter`[2] notebooks
- via an IDE, e.g. `Spyder`, `Eclipse` + `PyDev`, `PyCharm`

`python` is

- strongly typed: types are enforced
- dynamically, implicitly typed: you don't need to declare variables
- case sensitive: a and A are different variables
- object-oriented: everything is an object

Documentation

Via

help(<**object**>), <**object**>.__doc__

E.g.

```
>>> ['a'].__doc__
>>> "list() -> new empty list\nlist(iterable) -> new list
    initialized from iterable's items"
```

Data structures

Build-in data structures

- lists: similar to 1D arrays
- dictionaries: hash-tables (key-value pairs)
- tuples: immutable 1D arrays
- sets: since `python` 2.5

[2] `http://jupyter.org/`

```
>>> # list example
>>> mylist = ['a', 1, 2, [1, 2]]
>>> mylist[1] = 22
>>> mylist
[1, 22, 'a', [1, 2]]
>>> # tuple example
>>> mytuple = (1, 2)
>>> mytuple[1] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # dictionary example
>>> mydict = {0:'a', 1:'b'}
>>> mydict[0]
'a'
>>> # set example
>>> mylist = [1, 2, 2, 3]
>>> set(mylist)
set([1, 2, 3])
>>> # list slicing
>>> mylist = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> mylist[0:-1]
[1, 2, 3, 4, 5, 6, 7, 8]
>>> mylist[2:4]
[3, 4]
>>> mylist[-1:-4:-1]
[9, 8, 7]
>>> mylist[:5]
[1, 2, 3, 4, 5]
```

Flow control statements

Flow control statements in `python`: `if`, `for`, and `while`. To obtain a list of numbers, use `range<number>` (or better, `xrange<number>`)

```
>>> for i in xrange(5):
...     if i % 2:
...         print i
```

```
...     else:
...             print i - 1
...
-1
1
1
3
3
```

Functions

Functions are declared with the `def` keyword. Optional arguments are set in the function declaration. In `python`, functions can return a tuple (this way, you can effectively return multiple values)

```
>>> def increase_by_one(x):
...     return x + 1
...
>>> increase_by_one(2)
3
>>> increase_by_one(2.3)
3.3
>>> new_increase_by_one = lambda x: x + 1
>>> new_increase_by_one(1)
2
```

Use of external libraries

```
>>> # import entire library
>>> import [libname]
>>> # import specific functions/packages
>>> from [libame] import [funcname|packagename]
```

Object-oriented programming

Classes are declared with the `class` keyword. Private variables and methods can be declared, however, by convention, this is not enforced by the language.

```
>>> class MyClass:
...     variable = 10
...     def __init__(self):
...             print 'this is an instance of MyClass'
...     def myfunction(self, x):
...             return 2*x
...
>>> instance = MyClass()
this is an instance of MyClass
>>> instance.myfunction(10)
20
>>> instance.variable
10
>>> # inheritance example
>>> class SubClass(MyClass):
...     def __init__(self):
...             print 'this is an instance of the subclass'
...
>>> instance=SubClass()
this is an instance of the subclass
```

Polymorphism is also possible

```
>>> class Animal:
...     def __init__(self, name):
...             self.name = name
...     def talk(self):
...             raise NotImplementedError("Abstract method must
   be implemented")
...
>>> class Cat(Animal):
...     def talk(self):
...             return 'Meow!'
...
>>> class Dog(Animal):
...     def talk(self):
...             return 'Woof! Woof!'
...
>>> class Human(Animal):
```

```
...         def bark(self):
...                 print 'UQ is the best'
...
>>> animals = [Cat('Mr. Purrify'), Dog('Lassie'), Human('me')]
>>> for animal in animals:
...         print animal.name + ': ' + animal.talk()
...
Mr. Purrify: Meow!
Lassie: Woof! Woof!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 5, in talk
NotImplementedError: Abstract method must be implemented
```

Scientific computing with python

Since `python` is an interpreted and dynamically typed programming language, `python` programs can be slow compared to compiled statically typed programming languages, such as C and Fortran.
However, using suitable libraries, `python` can be used for scientific computing[3].

Numpy - multidimensional data arrays

package that provides high-performance vector, matrix and higher-dimensional data structures for `python`.

```
>>> import numpy as np
>>> # do clever things
```

SciPy - Library of scientific algorithms

It includes

- Special functions (scipy.special)
- Integration (scipy.integrate)
- Optimization (scipy.optimize)

[3] some materials on scientific computing `https://github.com/jrjohansson/scientific-python-lectures`

- Interpolation (scipy.interpolate)
- Fourier Transforms (scipy.fftpack)
- Signal Processing (scipy.signal)
- Linear Algebra (scipy.linalg)
- Sparse Eigenvalue Problems (scipy.sparse)
- Statistics (scipy.stats)
- Multi-dimensional image processing (scipy.ndimage)
- File IO (scipy.io)

matplotlib - 2D and 3D plotting

Using Fortran and C code

In `python` it is relatively easy to call out to libraries with compiled `C` or `Fortran` code

- `Fortran`: `F2PY`
- `C`: `ctypes`, `Cython`

Tools for high-performance computing applications

- thread parallelism: the `multiprocessing` package
- IPython parallel
- MPI: mpi4py
- opencl: pyopencl
- etc.

Assignment

A linear damped oscillator is a modeled by a second order ODE

$$\begin{cases} \frac{d^2y}{dt^2}(t) + c\frac{dy}{dt}(t) + ky(t) = f\cos(\omega t) \\ y(0) = y_0 \\ \frac{dy}{dt}(0) = y_1, \end{cases} \tag{1}$$

where c is the damping coefficient, k the spring constant, f the forcing amplitude, ω the frequency, y_0 represents the initial position, whereas y_1 is the initial velocity.

Assignment 1

Considering $t \in [0, 20]$, $c = 0.5$, $k = 2.0$, $f = 0.5$, $\omega = 1.05$, $y_0 = 0.5$, $y_1 = 0.0$, discretize the above equation using finite differences (central difference schemes on a Cartesian grid) with $\Delta t = 0.1, 0.05, 0.01$ and standard `python` lists.

Assignment 2

Considering the setup as in Assignment 1, discretize the oscillator using finite differences and `numpy` lists. What do you observe?

Assignment 3

With the same setup, solve (1) using the `odeint`[4] function from `scipy.integrate`.

chaospy installation/building from source

Throughout these tutorials, we will use the `chaospy` library. `chaospy`[5] is a library designed to perform uncertainty quantification using (but not restricted to) polynomial chaos expansions and advanced Monte Carlo methods implemented in Python 2 and 3. Assuming that you have a Unix-based operating system on your machine, to install `chaospy` you could use the `pip` package management system

- install `pip`: `sudo apt install python-pip`

[4] see `https://docs.scipy.org/doc/scipy-0.17.0/reference/generated/scipy.integrate.odeint.html`
[5] online documentation `http://chaospy.readthedocs.io/en/master/`

- install `numpy`, `scipy`, `matplotlib`: `pip install numpy scipy matplotlib`[6]
- install `scikit-learn`[7]: `pip install scikit-learn`
- install `chaospy`: `(sudo)`[8] `pip install chaospy`

or you could build `chaospy` directly from source

- install `git`: `sudo apt install git`
- clone the chaospy repository: `git clone https://github.com/jonathf/chaospy.git`
- build `chaospy`
  - `cd /path_to_chaospy/chaospy`
  - `sudo apt install python-numpy python-scipy python-matplotlib`[9]
  - `sudo apt install python-scikit-learn`
  - `(sudo)`[10] `python setup.py install`
  - add /path_to_chaospy/chaospy/build to your PYTHON_PATH
    - type `export $PYTHON_PATH=/path_to_chaospy/chaospy/build:$PYTHON_PATH` in your terminal or
    - add `$PYTHON_PATH=/path_to_chaospy/chaospy/build:$PYTHON_PATH` into your .bashrc file
- at the end test the build locally
  - `pip install -r requirements-dev.txt`
  - `python setup.py test`

[6] on systems where you don't have root access, use `pip install --user`
[7] to support more regression methods in `chaospy`; optional
[8] it might work without root access
[9] alternatively, you could use `pip install -r requirements-dev.txt`
[10] it might work without root access

chaospy installation on Windows

Disclaimer: this was not tried out, hence, some steps might be needed to be adjusted

- download and install `anaconda`: following the steps from `https://docs.continuum.io/anaconda/install-windows.html`
- install `numpy`, `scipy`, `matplotlib` via `conda install numpy scipy matplotlib`
- install `scikit-learn`, optional via `conda install scikit-learn`
- add `\path\to\anaconda\lib` to the `pythonpath` or
- pip very .py file to `anaconda`'s `python` (in `cmd`):
  ```
  assoc .py=Python.File
  ftype Python.File=C:path\to\Anaconda\python.exe "%1" %*
  ```