A worked example on scientific computing with Python

Hans Petter Langtangen[1,2] (hpl@simula.no)

[1]Simula Research Laboratory
[2]University of Oslo

Nov 1, 2015

Contents.

This worked example

- fetches a data file from a web site,
- applies that file as input data for a differential equation modeling a vibrating system,
- solves the equation by a finite difference method,
- visualizes various properties of the solution and the input data.

The following programming topics are illustrated:

- basic Python constructs: variables, loops, if-tests, arrays, functions
- flexible storage of objects in lists,
- storage of objects in files (persistence),
- downloading files from the web,
- user input via the command line,
- signal processing and FFT,
- curve plotting of data,
- unit testing,

- symbolic mathematics,
- modules.

All files can be forked at `https://github.com/hplgit/bumpy`.

Optimal background for reading this note.

- some interest in exploring physics through numerical simulation
- some very basic knowledge of
  - differential equations
  - finite difference approximations
  - Python or Matlab
- significant interest in exploring Python for scientific computations to solve a real-world physical problem (with low mathematical complexity)

You can read in two ways: either as a detailed example on using Python for solving differential equations (some very basic Python knowledge[a] is preferred) or just to get an impression of how Python can be used in a Matlab-like fashion.

[a] `http://hplgit.github.io/bumpy/doc/web/index.html`

If you need motivation for using Python as programming language, see Appendix A. Lists of many useful tutorials and introductions to Python, with emphasis on scientific computing, are found in Appendix B.

1 A scientific application

1.1 Physical problem and mathematical model

The task is to make a simulation program that can predict how a (simple) mechanical system oscillates in response to environmental forces. Introducing $u(t)$ as some displacement of the system at time t, application of Newton's second law of motion to such a mechanical system often results in the following type of equation for u:

$$mu'' + f(u') + s(u) = F(t), \tag{1}$$

The prime, as in u', denotes differentiation with respect to time ($u'(t)$ or du/dt). Furthermore, m is the mass of the system, $f(u')$ is a friction force that gives rise to a damping of the motion, $s(u)$ represents a restoring force, such as a spring, and $F(t)$ models the external environmental forces on the system. Equation

(1) must be accompanied by two initial conditions: $u(0) = I$ and $u'(0) = V$. The values of these have no effect on the steady state behavior of $u(t)$ for large values of t, since this behavior is determined by the force $F(t)$ and the system parameters m, $f(u')$, and $s(u)$.

There are two types of the friction force f: linear damping $f(u') = bu'$ and quadratic damping $f(u') = bu'|u'|$. The input data consists of m, b, $s(u)$, $F(t)$, I, V, and specification of linear or quadratic damping. The unknown quantity to be computed is $u(t)$ for $t \in (0, T]$.

One example where the model above has relevance, is the vertical vibration of a vehicle in response to a bumpy road. Let $h(x)$ be the height of the road at some coordinate x along the road. When driving along this road with constant velocity v, the vehicle is moved up and down in time according to $h(vt)$, resulting in an external vertical force $F(t) = -mh''(vt)v^2$. We assume that the vehicle has springs and dampers that here are modeled as bu' (damper) and $s(s) = ku$ (spring), with given damping parameter b and spring constant k. The unknown $u(t)$ is the vertical displacement of the vehicle relative to the road. Figure 1 illustrates the situation. You may view an animation[1] of such a motion on the web (by the way, this animation was created by coupling the Pysketcher[2] tool for figure drawings with the differential equation solver presented later, see the script `bumpy_road_fig.py`[3] for all details).

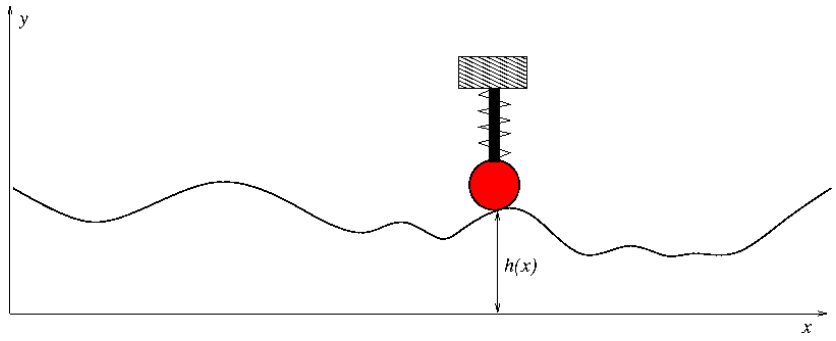

Figure 1: Vehicle on a bumpy road.

Another example regards the vertical shaking of a building due to earthquake-induced movement of the ground. If the vertical displacement of the ground is recorded as a function $d(t)$, this results in a vertical force $F(t) = -md''(t)$. The soil foundation acts as a spring and damper on the building, modeled through the damping parameter b and normally a linear spring term $s(u) = ku$.

In both cases we drop the effect of gravity, which is just a constant compression of the spring.

[1] `http://hplgit.github.io/bumpy/doc/src/mov-bumpy/m2_k1_5_b0_2/index.html`
[2] `https://github.com/hplgit/pysketcher`
[3] `https://github.com/hplgit/bumpy/blob/master/doc/src/fig-bumpy/bumpy_road_fig.py`

Our task is to compute and analyze the vertical $u(t)$ vibrations of a vehicle, given the shape $h(x)$ of the road and some velocity v.

1.2 Numerical model

The differential equation problem (1) can be solved by introducing finite difference approximations for u'' and u'. In case of quadratic damping one can use a geometric mean to approximate $u'|u'|$ and thereby linearize the equations. The result of using such numerical methods is an algorithm for computing $u(t)$ at discrete points in time. Let u^n be the approximation to u at time $t_n = n\Delta t$, $n = 1, 2, \ldots$, where Δt is a (small) time interval. For example, if $\Delta t = 0.1$, we find approximations u^1 to u at $t = 0.1$, u^2 at $t = 0.2$, u^3 to $t = 0.3$, and so forth. Any value u^{n+1} can be computed if u^n and u^{n-1} are known (i.e., previously computed). The formula for u^{n+1} is, in case of linear damping $f(u') = bu'$,

$$u^{n+1} = \left(2mu^n + (\frac{b}{2}\Delta t - m)u^{n-1} + \Delta t^2(F^n - s(u^n))\right)(m + \frac{b}{2}\Delta t)^{-1}, \quad (2)$$

where F^n means $F(t)$ evaluated for $t = t_n$. A special formula must be applied for $n = 0$:

$$u^1 = u^0 + \Delta t\, V + \frac{\Delta t^2}{2m}(-bV - s(u^0) + F^0)\,. \quad (3)$$

For quadratic damping we have a slightly different formula,

$$u^{n+1} = \left(m + b|u^n - u^{n-1}|\right)^{-1} \times \left(2mu^n - mu^{n-1} + bu^n|u^n - u^{n-1}| + \Delta t^2(F^n - s(u^n))\right), \quad (4)$$

and again a special formula for u^1:

$$u^1 = u^0 + \Delta t V + \frac{\Delta t^2}{2m}\left(-bV|V| - s(u^0) + F^0\right)\,. \quad (5)$$

The implementation of the computational algorithm can make use of an array `u` to represent u^n as `u[n]`. The force $F(t_n)$ is assumed to be available as an array element `F[n]`. The following Python function computes `u` given an array `t` with time points $t_0, t_1, \ldots$, the initial displacement `I`, mass `m`, damping parameter `b`, restoring force `s(u)`, environmental forces `F` as an array (corresponding to `t`).

1.3 Simple implementation

Let us first implement the computational formulas for the linear damping case in a short and compact Python function:

```
from numpy import *

def solver_linear_damping(I, V, m, b, s, F, t):
```

```
N = t.size - 1              # No of time intervals
dt = t[1] - t[0]            # Time step
u = zeros(N+1)              # Result array
u[0] = I
u[1] = u[0] + dt*V + dt**2/(2*m)*(-b*V - s(u[0]) + F[0])

for n in range(1,N):
    u[n+1] = 1./(m + b*dt/2)*(2*m*u[n] + \
             (b*dt/2 - m)*u[n-1] + dt**2*(F[n] - s(u[n])))
return u
```

1.4 Dissection of the code

Functions in Python start with `def`, followed by the function name and the list of input objects separated by comma. The function body is indented, and the first non-indented line signifies the end of the function body block. Output objects are returned to the calling code by a `return` statement.

The arguments to this function and the variables created inside the function are not declared with type. We therefore need to know what the variables are supposed to be: `I`, `V`, `m`, and `b` are real numbers, while `F` and `t` are one-dimensional arrays of the same length, where `F` holds $F(t_n)$ and `t` holds t_n, $n = 0, 1, \ldots, N+1$.

The number of elements in an array `t` is given by `t.size` (or `len(t)`, but `t.size` works for multi-dimensional arrays too). Arrays are indexed by square brackets, and indices always start at 0. Numerical codes frequently needs to loop over array indices, i.e., a set of integers. Such a set is produced by `range(start, stop, increment)`, which returns a list of integers `start, start+increment, start+2*increment`, and so on, up to *but not including* `stop`. Writing just `range(stop)` means `range(0, stop, 1)`. The particular call `range(1, N)` used in the code above results in a list of integers: `1`, `2`, ..., `N-1`.

Array functionality is enabled by the `numpy` package, which offers functions such as `zeros` and `linspace`, as known from Matlab. Here we import all objects in the `numpy` package by the statement

```
from numpy import *
```

Comments start with the character `#` and the rest of the line is then ignored by Python.

Every variable in Python is an object. In particular, the `s` function above is a function object, transferred to the function as any other object, and called as any other function. Transferring a function as argument to another function is therefore simpler and cleaner in Python than in, e.g., C, C++, Java, C#, and Matlab.

How can we use the `solver_linear_damping` function? We need to *call* it with relevant values for the arguments. Suppose we want to solve a vibration problem with $I = 1$, $V = 0$, $F = 0$, $m = 2$, $b = 0.2$, $s(u) = 2u$, $\Delta t = 0.2$, for $t \in [0, 10\pi]$. This will be a damped sinusoidal solution (setting $b = 0$ will result in $u(t) = \cos t$). The test code becomes

```
from solver import solver_linear_damping
from numpy import *

def s(u):
    return 2*u

T = 10*pi      # simulate for t in [0,T]
dt = 0.2
N = int(round(T/dt))
t = linspace(0, T, N+1)
F = zeros(t.size)
I = 1; V = 0
m = 2; b = 0.2
u = solver_linear_damping(I, V, m, b, s, F, t)

from matplotlib.pyplot import *
plot(t, u)
savefig('tmp.pdf')   # save plot to PDF file
savefig('tmp.png')   # save plot to PNG file
show()
```

The `solver_linear_damping` function resides in the file `solver.py`, so if we make the call to this function in separate file (assumed above), we have to import the function as shown. We also need functions and variables from the `numpy` package, here `pi`, `linspace`, and `zeros`. Normally, there is one statement per line in Python programs, and there is no need to end the statement with a semicolon. However, if we want multiple statements on a line, they must be separated by semi colon as demonstrated in the initialization of `I` and `V`. Plotting the computed curve makes use of the `matplotlib` package, where we call its `plot`, `savefig`, and `show` functions. Figure 2 shows the result.

1.5 More advanced implementation

Let us extend the function above to also include the quadratic damping formulas. The $F(t)$ function in (1) is required to be available as an array in the `solver_linear_damping` function, but now we will allow for more flexibility: the `F` argument may either be a Python function `F(t)` or a Python array. In addition, we add some checks that variables have correct values or are of correct type.

```
import numpy as np

def solver(I, V, m, b, s, F, t, damping='linear'):
    """
    Solve m*u'' + f(u') + s(u) = F for time points in t.
    u(0)=I and u'(0)=V,
    by a central finite difference method with time step dt.
    If damping is 'linear', f(u')=b*u, while if damping is
    'quadratic', we have f(u')=b*u'*abs(u').
    s(u) is a Python function, while F may be a function
    or an array (then F[i] corresponds to F at t[i]).
    """
    N = t.size - 1              # No of time intervals
    dt = t[1] - t[0]            # Time step
    u = np.zeros(N+1)           # Result array
```

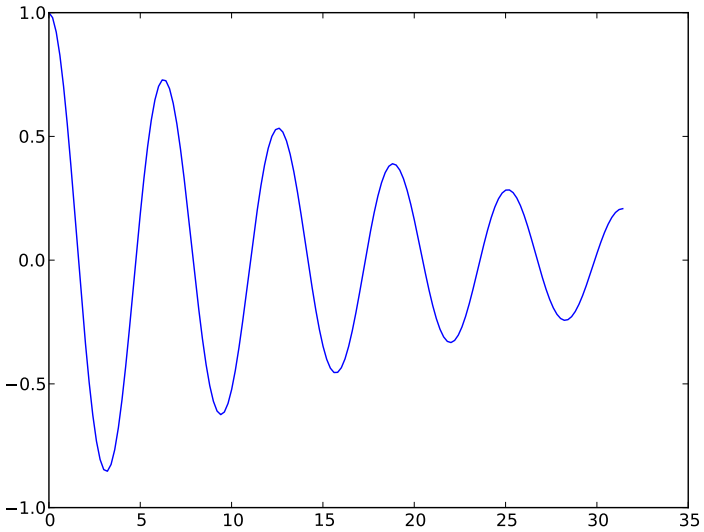

Figure 2: Plot of computed curve.

```
b = float(b); m = float(m)  # Avoid integer division

# Convert F to array
if callable(F):
    F = F(t)
elif isinstance(F, (list,tuple,np.ndarray)):
    F = np.asarray(F)
else:
    raise TypeError(
        'F must be function or array, not %s' % type(F))

u[0] = I
if damping == 'linear':
    u[1] = u[0] + dt*V + dt**2/(2*m)*(-b*V - s(u[0]) + F[0])
elif damping == 'quadratic':
    u[1] = u[0] + dt*V + \
           dt**2/(2*m)*(-b*V*abs(V) - s(u[0]) + F[0])
else:
    raise ValueError('Wrong value: damping="%s"' % damping)

for n in range(1,N):
    if damping == 'linear':
        u[n+1] = (2*m*u[n] + (b*dt/2 - m)*u[n-1] +
                  dt**2*(F[n] - s(u[n])))/(m + b*dt/2)
    elif damping == 'quadratic':
        u[n+1] = (2*m*u[n] - m*u[n-1] + b*u[n]*abs(u[n] - u[n-1])
                  - dt**2*(s(u[n]) - F[n]))/\
                  (m + b*abs(u[n] - u[n-1]))
return u, t
```

1.6 Dissection of the code

Two types of import. This time we replace `from numpy import *`, which imports over 500 variables and functions, by `import numpy as np`, which just imports one variable, the package `numpy`, here under the nickname `np`. All variables and functions in `numpy` must now be reached via the prefix `np.`, as in `np.zeros`, and `np.linspace`, and `np.pi` (for π). The advantage of the prefix is that we clearly see where functionality comes from. The disadvantage is that mathematical formulas like $\sin(\pi x)$ must be written `np.sin(np.pi*x)`. We can always perform an explicit import of some names, like `sin` and `pi`, to write the formula as `sin(pi*x)`:

```
import numpy as np
from numpy import sin, pi

def myfunction(x):
    return sin(pi*x)
```

Another disadvantage with the `np.` prefix is that names are no longer (almost) the same as in Matlab. Many will therefore prefer to do `from numpy import *` and skip the prefix.

Doc strings. The string, enclosed in triple double-quotes, right after the function definition, is a *doc string* used for documenting the function. Various tools can extract function definitions and doc strings to automatically produce nicely typeset manuals.

Avoiding integer division. At one line we explicitly convert `b` and `m` to `float` variables. This is not strictly necessary, but if we supply `b=2` and `m=4`, a computation like `b/m` will give zero as result because both `b` and `m` are then integers and `b/m` implies *integer division*, not the mathematical division of real numbers (this is not a special feature of Python - all languages with a strong heritage from C invoke integer division if both operands in a division are integers). We need to make sure that at least one of the operands in a division is a real number (`float`) to ensure the intended mathematical operation. Looking at the formulas, there is never a problem with integer division in our implementation, because `dt` is computed from `t`, which has `float` elements (`linspace` makes `float` elements by default), and all divisions in our code involve `dt` as one of the operands. However, future edits may alter the way formulas are written, so to be on the safe side we ensure that real input parameters are `float` objects.

Flexible variable type. As mentioned, we allow `F` to be either a function or an array. In the former case, we convert `F` to an array such that the rest of the code can assume that `F` is indeed an array. It is easy to check the type of variables in Python. The test `if callable(F)` is true if the object `F` can be called as a function.

Checking correct variable type. To test that `F` is an array, we can use `isinstance(F, np.ndarray)` (`ndarray` is the name of the array type in `numpy`). In the code above we allow that `F` can also be a list or a tuple. Running `F` through the `asarray` function makes an array out of `F` in the cases where `F` is a list or tuple. The final `else:` clause takes care of the situation where `F` is neither a function, nor an object that can easily be converted to an array, and an error message is issued. More precisely, we *raise* a `TypeError` *exception* indicating that we have encountered a wrong type. The error message will contain the type of `F` as obtained from `type(F)`. Instead of using the syntax `isinstance(F, list)` we may test `type(F) == list` or even `type(F) in (list,tuple,np.ndarray)`.

We could simplify the `if-else` test involving `F` to just the two lines

```
if callable(F):
    F = F(t)
```

if we are sure that `F` is either a function or a `numpy` array. Should the user send in something else for `F`, Python will encounter a run-time error when trying to index `F` as in `F[0]` (in the statement computing `u[1]`).

To be absolutely safe, we should test that the other arguments are of right type as well. For example,

```
if not isinstance(I, (float,int)):
    raise TypeError('V must be float or int, not %s' % type(V))
```

and similar for `V`, `m`, `b`, while `s` must be tested by `callable(s)`, `t` must be `np.ndarray`, and `damping` must be `str`.

Calling the function. Below is a simple example on how the `solver` function can be called to solve the differential equation problem $mu'' + bu' + ku = A\sin \pi t$, $u(0) = I$, $u'(0) = 0$:

```
import numpy as np
from numpy import sin, pi  # for nice math

def F(t):
    # Sinusoidal bumpy road
    return A*sin(pi*t)

def s(u):
    return k*u

A = 0.25
k = 2
t = np.linspace(0, 20, 2001)
u, t = solver(I=0, V=0, m=2, b=0.05, s=s, F=F, t=t)

# Show u(t) as a curve plot
import matplotlib.pyplot as plt
plt.plot(t, u)
plt.show()
```

Local and global variables. We use in this example a linear spring function $s(u)$,

```
def f(u):
    return k*u
```

Here, `u` is a *local variable*, which is accessible just inside in the function, while `k` is a *global variable*, which must be initialized outside the function prior to calling `f`.

Advanced programming of functions with parameters.

The best way to implement mathematical functions that has a set of parameters in addition some independent variables is to create a *class* where the parameters are attributes and a `__call__` method evaluates the function formula given the independent variables as arguments. This requires, of course, knowledge of classes and special methods like `__call__`.

As an example, the `f(u)` function above can be implemented as

```
class Spring:
    def __init__(self, k):
        self.k = k
    def __call__(self, u):
        return self.k*u

f = Spring(k)
```

Because of the `__call__` method, we can make calls `f(u)`. Note that the `k` parameter is bundled with the function in the `f` object.

1.7 The excitation force

Considering the application where the present mathematical model describes the vibrations of a vehicle driving along a bumpy road, we need to establish the force array `F` from the shape of the road $h(x)$. Various shapes are available as a file with web address `http://hplbit.bitbucket.org/data/bumpy/bumpy.dat.gz`. The Python functionality for downloading this `gzip` compressed file as a local file `bumpy.dat.gz` and reading it into a `numpy` array goes as follows:

```
filename = 'bumpy.dat.gz'
url = 'http://hplbit.bitbucket.org/data/bumpy/bumpy.dat.gz'
import urllib
urllib.urlretrieve(url, filename)
h_data = np.loadtxt(filename)      # read numpy array from file
```

The `h_data` object is a rectangular `numpy` array where the first column contains the x coordinates along the road and the next columns contain various road shapes $h(x)$. We can extract the x data and redefine `h_data` to contain solely the $h(x)$ shapes:

```
x = h_data[0,:]                 # 1st column: x coordinates
h_data = h_data[1:,:]           # other columns: h shapes
```

In general, the syntax `a[s:t:i,2]` gives a *view* (not a copy) to the part of the array `a` where the first index goes from `s` to `t`, *but not including* the `t` value, in increments of `i`, and the second index is fixed at 2. Just writing `:` for an index means all legal values of this index.

Given $h(x)$, the corresponding acceleration $a(t)$ needed in the force $F(t) = -ma(t)$, follows from $a(t) = h''(vt)v^2$, where v is the velocity of the vehicle. The computation may utilize a finite difference approximation for the second-order derivative h'' and be encapsulated in a Python function:

```
def acceleration(h, x, v):
    """Compute 2nd-order derivative of h."""
    # Method: standard finite difference aproximation
    d2h = np.zeros(h.size)
    dx = x[1] - x[0]
    for i in range(1, h.size-1, 1):
        d2h[i] = (h[i-1] - 2*h[i] + h[i+1])/dx**2
    # Extraplolate end values from first interior value
    d2h[0] = d2h[1]
    d2h[-1] = d2h[-2]
    a = d2h*v**2
    return a
```

Note that here, `h` is a one-dimensional array containing the $h(x)$ values corresponding to a given coordinate array `x`. Also note that we for mathematical simplicity set $h''(x)$ at the end points equal to $h''(x)$ at the closest interior point.

The computations of `d2h` above was done array element by array element. This loop can be a slow process in Python for long arrays. To speed up computations dramatically, we can invoke a *vectorization* of the above algorithm. This means that we get rid of the loops and perform arithmetics on complete (or almost complete) arrays. The vectorized form of the `acceleration` function goes like

```
def acceleration_vectorized(h, x, v):
    """Compute 2nd-order derivative of h. Vectorized version."""
    d2h = np.zeros(h.size)
    dx = x[1] - x[0]
    d2h[1:-1] = (h[:-2] - 2*h[1:-1] + h[2:])/dx**2
    # Extraplolate end values from first interior value
    d2h[0] = d2h[1]
    d2h[-1] = d2h[-2]
    a = d2h*v**2
    return a
```

For each shape $h(x)$ we want to compute the corresponding vertical displacement $u(t)$ using the mathematical model (1). This can be accomplished by looping over the columns of `h_data` and calling `solver` for each column, i.e., each realization of the force F. The major arrays from the computations are collected in a list `data`. The two first elements in `data` are `x` and `t`. The next elements are 3-lists `[h, a, u]` for each road shape. Note that some elements in

`data` are arrays while others are list of arrays. This composition is convenient when analyzing and visualizing key quantities in the problem.

The computations of `u` for each road shape can be done as follows:

```
data = [x, t]       # key input and output data (arrays)
for i in range(h_data.shape[0]):
    h = h_data[i,:]             # extract a column
    a = acceleration(h, x, v)
    F = -m*a

    u = solver(t=t, I=0, m=m, b=b, f=f, F=F)

    data.append([h, F, u])
```

A parameter choice $m = 60$ kg, $v = 5$ m/s, $k = 60$ N/m, and $b = 80$ Ns/m corresponds to a velocity of 18 km/h and a mass of 60 kg, i.e., bicycle conditions.

1.8 A high-level solve function

The code above for simulating vertical vibrations in a vehicle is naturally implemented as a Python function. This function can take the most important physical parameters of the problem as input, along with information about the file with road shapes. We allow for defining road shapes either through a file on a web site or a local file.

```
def bumpy_road(url=None, m=60, b=80, k=60, v=5):
    """
    Simulate vertical vehicle vibrations.

    =========   ============================================
    variable    description
    =========   ============================================
    url         either URL of file with excitation force data,
                or name of a local file
    m           mass of system
    b           friction parameter
    k           spring parameter
    v           (constant) velocity of vehicle
    Return      data (list) holding input and output data
                [x, t, [h,F,u], [h,F,u], ...]
    =========   ============================================
    """
    # Download file (if url is not the name of a local file)
    if url.startswith('http://') or url.startswith('file://'):
        import urllib
        filename = os.path.basename(url)  # strip off path
        urllib.urlretrieve(url, filename)
    else:
        # Check if url is the name of a local file
        filename = url
        if not os.path.isfile(filename):
            print url, 'must be a URL or a filename'
            sys.exit(1)  # abort program
        # else: ok

    h_data = np.loadtxt(filename)  # read numpy array from file
```

```
x = h_data[0,:]                 # 1st column: x coordinates
h_data = h_data[1:,:]           # other columns: h shapes

t = x/v                         # time corresponding to x
dt = t[1] - t[0]

def f(u):
    return k*u

data = [x, t]       # key input and output data (arrays)
for i in range(h_data.shape[0]):
    h = h_data[i,:]             # extract a column
    a = acceleration(h, x, v)
    F = -m*a

    u = solver(t=t, I=0, m=m, b=b, f=f, F=F)

    data.append([h, F, u])
return data
```

Note that function arguments can be given default values (known as *keyword arguments* in Python). Python has a lot of operating system functionality, such as checking if a file, directory, or link exist, creating or removing files and directories, running stand-alone applications, etc.

1.9 Storing Python objects in files

After calling

```
road_url = 'http://hplbit.bitbucket.org/data/bumpy/bumpy.dat.gz'
data = solve(url=road_url, m=60, b=200, k=60, v=6)
```

the `data` array contains single arrays and triplets of arrays,

```
[x, t, [h,F,u], [h,F,u], ..., [h,F,u]]
```

This list, or any Python object, can be stored on file for later retrieval of the results, using the *pickling* functionality in Python:

```
import cPickle as pickle
# Or using a more advanced module
import dill as pickle
outfile = open('bumpy.res', 'w')
pickle.dump(data, outfile)
outfile.close()
```

The advantage of the `dill` module over `cPickle` is that it can store a wider range of Python objects (e.g., lambda functions).

The code above and the `bumpy_road` function are found in the file `bumpy.py`[4].

[4] `https://github.com/hplgit/bumpy/blob/master/doc/src/src-bumpy/bumpy.py`

1.10 Computing the root mean square value

Since the roads have a quite noise shape, the force $F = -ma$ looks very noisy, while the response $u(t)$ to this excitation is significantly less noisy, see the bottom plot in Figure 5 for an example. It may be useful to compute the root mean square value[5] of u to get a number for the typical amplitude of the vibrations:

$$u_{\text{rms}} = \sqrt{T^{-1}\int_0^T u^2 dt} \approx \sqrt{\frac{1}{N+1}\sum_{i=0}^{N}(u^n)^2}\,.$$

The last expression can be computed as follows using a vectorized sum (`np.sum(u**2)`):

```
u_rms = []
for h, F, u in data[2:]:
    u_rms.append(np.sqrt((1./len(u))*np.sum(u**2))
```

Very often in numerical computing we have some list/array and want to compute a new list/array where each element is some expression involving an element of the first list/array, typically

```
v = []
for element in u:
    v.append(expression(element))
```

This code can more compactly be written as a *list comprehension*:

```
v = [expression(element) for element in u]
```

We may use the list comprehension construction for computing the `u_rms` values:

```
u_rms = [np.sqrt((1./len(u))*np.sum(u**2))
         for h, F, u in data[2:]]
```

2 User input

We can in this example easily set the input data directly in the program, e.g., in the call to the `solve` function, as demonstrated above. However, most users will find it more convenient to set parameters through a user interface rather than editing the source code directly.

2.1 Positional command-line arguments

The simplest, and often also the most effective type of user interface is to use the command line. Suppose m, k, and v, as well as the URL or filename for the road shapes, are fixed parameters and that the user is allowed to vary b only. Then it is convenient, both for the user and the programmer, to specify b as the first command-line argument to the program. If the name of the program file is `bumpy.py` and $b = 10$ is desired, we can write

[5] `http://en.wikipedia.org/wiki/Root_mean_square`

```
Terminal> python bumpy.py 10
```

The corresponding code in the program for setting input data and extract the user-given value of b reads

```
def prepare_input():
    url = 'http://hplbit.bitbucket.org/data/bumpy/bumpy.dat.gz'
    m = 60
    k = 60
    v = 5
    try:
        b = float(sys.argv[1])
    except IndexError:
        b = 80  # default
    return url, m, b, k, v
```

The command-line arguments are available as strings in the list `sys.argv`, from the element with index 1 and onward. The first command-line argument `sys.argv[1]` is a string so it must be converted to a `float` object (representing real number) prior to computations. If the command-line argument is missing, `sys.argv[1]` is illegal indexing and the `IndexError` exception is raised. We can test for this error and provide a default value. Without the `try-except` construction, the program will abort with an error message if no command-line argument is given.

2.2 Option-value pairs on the command line

Letting the user set many parameters on the command line is most conveniently done by allowing option-value pairs, e.g.,

```
Terminal> python bumpy.py --m 40 --b 280
```

All parameters have a default value which can be overridden on the command line by providing the string (option) `-name`, where `name` is the name of the parameter, followed by the desired value of the parameter. Implementation of option-value input is most easily carried out using Python's `argparse` module. The recipe goes as follows.

```
def command_line_options():
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('--m', '--mass', type=float,
                        default=60, help='mass of vehicle')
    parser.add_argument('--k', '--spring', type=float,
                        default=60, help='spring parameter')
    parser.add_argument('--b', '--damping', type=float,
                        default=80, help='damping parameter')
    parser.add_argument('--v', '--velocity', type=float,
                        default=5, help='velocity of vehicle')
    url = 'http://hplbit.bitbucket.org/data/bumpy/bumpy.dat.gz'
    parser.add_argument('--roadfile', type=str,
              default=url, help='filename/URL with road data')
    args = parser.parse_args()
    # Extract input parameters
```

```
    m = args.m; k = args.k; b = args.b; v = args.v
    url = args.roadfile
    return url, m, b, k, v
```

We may offer two options for each parameter, one reflecting the mathematical symbol (like `-v`) and one more descriptive text (like `-velocity`).

3 Visual exploration

This section explains how to load the data from the computation, stored as a pickled list in the file `bumpy.res`, into various arrays, and how to visualize these arrays. We want to produce the following plots:

- $u(t)$ and $u'(t)$ for $t \geq t_s$
- the spectrum of $u(t)$, for $t \geq t_s$ (using FFT) to see which frequencies that dominate in the signal
- for each road shape, a plot of $h(x)$, $a(t)$, and $u(t)$, for $t \geq t_s$

Test case. The simulation is run as

```
Terminal> python bumpy.py --v 10
```

meaning that all other parameters have their default values as specified in the `command_line_arguments` function.

Importing array and plot functionality. The following two imports give access to Matlab-style functions for plotting and computing with arrays:

```
from numpy import *
from matplotlib.pyplot import *
```

The Python community often prefers to prefix `numpy` by `np` and `matplotlib` by `plt` or `mpl`:

```
import numpy as np
import matplotlib.pyplot as plt
```

However, we shall stick to the former import to make the code as close as possible to the Matlab equivalent.

Reading results from file. Loading the computational data from file back to a list `data` must use the same module (`cPickle` or `dill`) as we used for writing the data:

```
import cPickle as pickle
# or
import dill as pickle
outfile = open('bumpy.res', 'r')
data = pickle.load(outfile)
outfile.close()

x, t = data[0:2]
```

Note that `data` first contains `x` and `t`, thereafter a sequence of 3-lists `[h, F, u]`. With `data[0:2]` we extract the sublist with elements `data[0]` and `data[1]` (`0:2` means up to, but not including 2). All the remaining 3-lists `[h, F, u]` can be extracted as `data[2:]`.

Since now we concentrate on the part $t \geq t_s$ of the data, we can grab the corresponding parts of the arrays in the following way, using boolean arrays as indices:

```
indices = t >= t_s    # True/False boolean array
t = t[indices]        # fetch the part of t for which t >= t_s
x = x[indices]        # fetch the part of x for which t >= t_s
```

Indexing by a boolean array extracts all the elements corresponding to the `True` elements in the index array.

Plotting u. A plot of u, corresponding to the second realization of the bumpy road, is easy to create:

```
figure()
realization = 1
u = data[2+realization][2][indices]
plot(t, u)
title('Displacement')
```

Note that `data[2:]` holds the triplets `[h,F,u]`, so the second realization is `data[3]`. The `u` part of the triplet is then `data[3][2]`, and the part of `u` where $t \geq t_s$ is then `data[3][2][indices]`.

Figure 3 (left) shows the result.

Computing and plotting u'. Given a discrete function u^n, $n = 0, \ldots, N$, the corresponding discrete derivative can be computed by

$$v^n = \frac{u^{n+1} - u^{n-1}}{2\Delta t}, \quad n = 1, \ldots, N-1\,. \tag{6}$$

The values at the end points, v^0 and v^N, must be computed with less accurate one-sided differences:

$$v^0 = \frac{u^1 - u^0}{\Delta t}, \quad v^N = \frac{u^N - u^{N-1}}{\Delta t}$$

Computing $v = u'$ by (6) for the part $t \geq t_s$ is done by

```
v = zeros_like(u)           # same length and data type as u
dt = t[1] - t[0]            # time step
for i in range(1,u.size-1):
    v[i] = (u[i+1] - u[i-1])/(2*dt)
v[0] = (u[1] - u[0])/dt
v[N] = (u[N] - u[N-1])/dt
```

Plotting of `v` is done by

```
figure()
plot(t, v)
legend(['velocity'])
xlabel('t')
title('Velocity')
```

Figure 3 (right) shows that the velocity is much more "noisy" than the displacement.

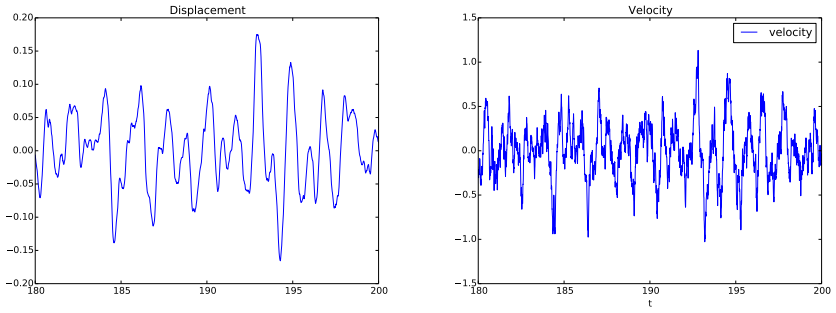

Figure 3: Displacement (left) and velocity (right) for $t \geq t_s = 180$.

Vectorized computation of u'. For large arrays, finite difference formulas like (6) can be heavy to compute. A vectorized expression where the loop is avoided is much more efficient:

```
v = zeros_like(u)
v[1:-1] = (u[2:] - u[:-2])/(2*dt)
v[0] = (u[1] - u[0])/dt
v[-1] = (u[-1] - u[-2])/dt
```

The challenging expression here is `u[2:] - u[:-2]`. The array slice `u[2:]` starts at index `2` and goes to the very end of the array, i.e., the slice contains `u[2], u[3], ..., u[N]`. The slice `u[:-2]` starts from the beginning of `u` and goes up to, *but not including* the second last element, i.e., `u[0], u[1], ..., u[N-2]`. The expression `u[2:] - u[:-2]` is then an array of length `N-2` with elements `u[2]-u[0]`, `u[3]-u[1]`, and so on, and with `u[N]-u[N-2]` as final element. After dividing this array by `2*dt` we have evaluated formula (6) for all indices from `1` to `N-2`, which is the same as the slice `1:-1` (used on the left-hand

side). Rather than using `N` (equal to `u.size-1`) in the last line above, we use the `-1` index for the last element and `-2` for the second last element.

How efficient is this vectorization? The interactive IPython shell has a convenient feature to test the efficiency of Python constructions:

```
In [1]: from numpy import zeros

In [2]: N = 1000000

In [3]: u = zeros(N)

In [4]: %timeit v = u[2:] - u[:-2]
1 loops, best of 3: 5.76 ms per loop

In [5]: v = zeros(N)

In [6]: %timeit for i in range(1,N-1): v[i] = u[i+1] - u[i-1]
1 loops, best of 3: 836 ms per loop

In [7]: 836/5.76
Out[20]: 145.13888888888889
```

This session show that the vectorized expression is 145 times faster than the explicit loop!

3.1 Computing the spectrum of signals

The spectrum of a $u(t)$ function, here represented by discrete values in the arrays `u` and `t`, can be computed by the Python function

```
def frequency_analysis(u, t):
    A = fft(u)
    A = 2*A
    dt = t[1] - t[0]
    N = t.size
    freq = arange(N/2, dtype=float)/N/dt
    A = abs(A[0:freq.size])/N
    # Remove small high frequency part
    tol = 0.05*A.max()
    for i in xrange(len(A)-1, 0, -1):
        if A[i] > tol:
            break
    return freq[:i+1], A[:i+1]
```

Note here that we truncate that last part of the spectrum where the amplitudes are small (this usually gives a plot that is easier to inspect).

In the present case, we utilize the `frequency_analysis` through

```
figure()
u = data[3][2][indices]  # second realization of u
f, A = frequency_analysis(u, t)
plot(f, A)
title('Spectrum of u')
```

Figure 4 shows the amplitudes and that the dominating frequencies lie around 0.5 and 1 Hz.

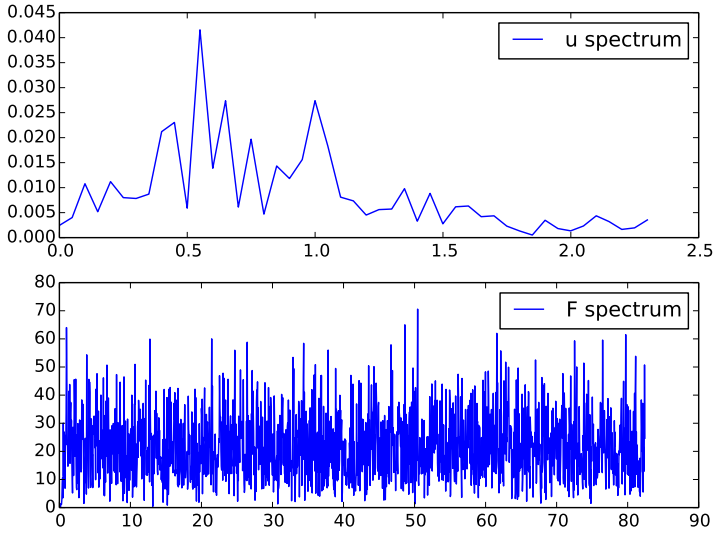

Figure 4: Spectra of displacement and excitation.

3.2 Multiple plots in the same figure

Finally, we can run through all the 3-lists `[h, a, u]` and visualize these curves in the same plot figure:

```
for realization in range(len(data[2:])):
    h, F, u = data[2+realization]
    h = h[indices]
    F = F[indices]
    u = u[indices]

    figure()
    subplot(3, 1, 1)
    plot(x, h, 'g-')
    legend(['h %d' % realization])
    hmax = (abs(h.max()) + abs(h.min()))/2
    axis([x[0], x[-1], -hmax*5, hmax*5])
    xlabel('distance'); ylabel('height')

    subplot(3, 1, 2)
    plot(t, F)
    legend(['F %d' % realization])
    xlabel('t'); ylabel('acceleration')

    subplot(3, 1, 3)
    plot(t, u, 'r-')
    legend(['u %d' % realization])
    xlabel('t'); ylabel('displacement')
    savefig('hFu%d.png' % realization)
```

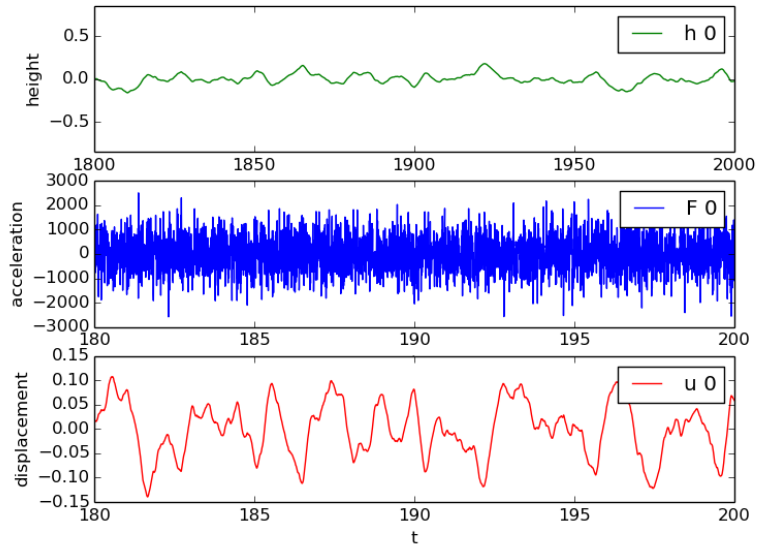

Figure 5: First realization of a bumpy road, with corresponding excitation of the wheel and resulting vertical vibrations.

If all the plot commands above are placed in a file, as in `explore.py`[6], a final `show()` call is needed to show the plots on the screen. On the other hand, the commands are usually more conveniently performed in an interactive Python shell, preferably IPython or IPython notebook.

4 Advanced topics

4.1 Symbolic computing via SymPy

Python has a package SymPy that offers symbolic computing. Here is a simple introductory example where we differentiate a quadratic polynomial, integrate it again, and find the roots:

```
>>> import sympy as sp
>>> x, a = sp.symbols('x a')         # Define mathematical symbols
>>> Q = a*x**2 - 1                   # Quadratic function
>>> dQdx = sp.diff(Q, x)             # Differentiate wrt x
>>> dQdx
2*a*x
>>> Q2 = sp.integrate(dQdx, x)       # Integrate (no constant)
>>> Q2
a*x**2
>>> Q2 = sp.integrate(Q, (x, 0, a)) # Definite integral
```

[6] https://github.com/hplgit/bumpy/blob/master/doc/src/src-bumpy/explore.py

```
>>> Q2
a**4/3 - a
>>> roots = sp.solve(Q, x)          # Solve Q = 0 wrt x
>>> roots
[-sqrt(1/a), sqrt(1/a)]
```

One can easily convert a SymPy expression like `Q` into a Python function `Q(x, a)` to be used for further numerical computing:

```
>>> Q = sp.lambdify([x, a], Q)      # Turn Q into Py func.
>>> Q(x=2, a=3)                     # 3*2**2 - 1 = 11
11
```

Sympy can do a lot of other things. Here is an example on computing the Taylor series of $e^{-x}\sin(rx)$, where r is the smallest root of $Q = ax^2 - 1 = 0$ as computed above:

```
>>> f = sp.exp(-a*x)*sp.sin(roots[0]*x)
>>> f.series(x, 0, 4)
-x*sqrt(1/a) + x**3*(-a**2*sqrt(1/a)/2 + (1/a)**(3/2)/6) +
a*x**2*sqrt(1/a) + O(x**4)
```

4.2 Testing

Software testing in Python is best done with a *unit test framework* such as nose[7] or pytest[8]. These frameworks can automatically run all functions starting with `test_` recursively in files in a directory tree. Each `test_*` function is called a *test function* and must take no arguments and apply `assert` to a boolean expression that is `True` if the test passes and `False` if it fails.

Example on a test function.

```
def halve(x):
    """Return half of x."""
    return x/2.0

def test_halve():
    x = 4
    expected = 2
    computed = halve(x)
    # Compare real numbers using tolerance
    tol = 1E-14
    diff = abs(computed - expected)
    assert diff < tol
```

Test function for the numerical solver. A frequently used technique to test differential equation solvers is to just specify a solution and fit a source term in the differential equation such that the specified solution solves the equation. The initial conditions must be set according to the specified solution. This technique is known as the *method of manufactured solutions*.

[7] `https://nose.readthedocs.org/`
[8] `http://pytest.org/latest/`

Here we shall specify a solution that is quadratic or linear in t because such lower-order polynomials will often be an exact solution of both the differential equation *and* the finite difference equations. This means that the polynomial should be reproduced to machine precision by our `solver` function.

We can use SymPy to find an appropriate $F(t)$ term in the differential equation such that a specified solution $u(t) = I + Vt + qt^2$ fits the equation and initial conditions. With quadratic damping, only a linear u will solve the discrete equation so in this case we choose $u = I + Vt$.

We embed the SymPy calculations and the numerical calculations in a *test function*:

```
def lhs_eq(t, m, b, s, u, damping='linear'):
    """Return lhs of differential equation as sympy expression."""
    v = sp.diff(u, t)
    d = b*v if damping == 'linear' else b*v*sp.Abs(v)
    return m*sp.diff(u, t, t) + d + s(u)

def test_solver():
    """Verify linear/quadratic solution."""
    # Set input data for the test
    I = 1.2; V = 3; m = 2; b = 0.9; k = 4
    s = lambda u: k*u
    T = 2
    dt = 0.2
    N = int(round(T/dt))
    time_points = np.linspace(0, T, N+1)

    # Test linear damping
    t = sp.Symbol('t')
    q = 2  # arbitrary constant
    u_exact = I + V*t + q*t**2   # sympy expression
    F_term = lhs_eq(t, m, b, s, u_exact, 'linear')
    print 'Fitted source term, linear case:', F_term
    F = sp.lambdify([t], F_term)
    u, t_ = solver(I, V, m, b, s, F, time_points, 'linear')
    u_e = sp.lambdify([t], u_exact, modules='numpy')
    error = abs(u_e(t_) - u).max()
    tol = 1E-13
    assert error < tol

    # Test quadratic damping: u_exact must be linear
    u_exact = I + V*t
    F_term = lhs_eq(t, m, b, s, u_exact, 'quadratic')
    print 'Fitted source term, quadratic case:', F_term
    F = sp.lambdify([t], F_term)
    u, t_ = solver(I, V, m, b, s, F, time_points, 'quadratic')
    u_e = sp.lambdify([t], u_exact, modules='numpy')
    error = abs(u_e(t_) - u).max()
    assert error < tol
```

Using a test framework. We recommend to use `pytest` as test framework. Subdirectories `test*` are examined for files `test_*.py` that have test functions `test_*()`, and all such tests are executed by the following command:

```
Terminal> py.test -s
```

The `-s` option makes all output from the tests appear in the terminal window.

To run the tests in a specific file `tests/test_bumpy.py`, do

```
Terminal> py.test -s tests/test_bumpy.py
```

4.3 Modules

Python software is frequently organized as modules, or in collection of modules, called packages. A module enables sharing functions, variables, and classes between programs and other modules.

It is very easy to make a module in Python. Just put all the functions and global variables (and classes, if you have) you want to include in the module in a file. If you have a main program calling functions or using global variables, these statements will be executed with you import the module in other programs, which is not what you want. Therefore, move the main program to a so-called *test block* at the end of the module:

```
if __name__ == '__main__':
    <statements in the main program>
```

The module file, say its name is `mymod.py`, now typically looks like

```
import module1
from module2 import func1, func2

def myfunc1(...):
    ...

def myfunc2(...):
    ...

if __name__ == '__main__':
    <statements in the main program>
```

The name of the module `mymod` if the filename is `mymod.py`. During `import mymod` or `from mymod import *`, the special Python variable `__name__` equals the name of the module and the test block with statements in the main program will not be executed. However, if you run `mymod.py` as a program, `__name__` equals `__main__` and the main program will be executed. In this way, you can have a single file that both acts as a library and that is an executable program.

A `from mymod import *` will import the global functions `myfunc1` and `myfunc2`, plus the global names `module1`, `somefunc1`, and `somefunc2`.

A Quick motivation for programming with Python

- Why Python?[9]
- Why Python for Scientific Computing?[10]

[9] `http://www.pyzo.org/whypython.html`
[10] `http://fperez.org/py4science/warts.html`

B Scientific Python resources

Full tutorials on scientific programming with Python.

- Python Scientific Lecture Notes[11] (from EuroSciPy tutorials, based on Python Scientific[12])
- Scientific Python Lectures as IPython notebooks[13]
- Stefan van der Walt's lectures[14]

NumPy resources.

- NumPy Tutorial[15]
- NumPy User Guide[16]
- Advanced NumPy Tutorial[17]
- Advanced NumPy Course[18]
- NumPy Example List[19]
- NumPy Medkit[20]
- NumPy and SciPy Cookbook[21]
- NumPy for Matlab Users[22]
- NumPy for Matlab/R/IDL Users[23]

Useful resources.

- AstroPython[24]
- Talk on IPython by Fernando Perez[25]

[11] http://scipy-lectures.github.com/
[12] http://web.phys.ntnu.no/ ingves/Teaching/TFY4240/Assignments/PythonScientific.pdf
[13] https://github.com/jrjohansson/scientific-python-lectures
[14] https://github.com/stefanv/teaching
[15] http://www.scipy.org/Tentative_NumPy_Tutorial
[16] http://docs.scipy.org/doc/numpy/user/
[17] https://github.com/pv/advanced-numpy-tutorial
[18] http://scipy2010.blogspot.com/2010/06/tutorials-day-1-advanced-numpy.html
[19] http://www.scipy.org/Numpy_Example_List
[20] http://mentat.za.net/numpy/numpy_advanced_slides/
[21] http://www.scipy.org/Cookbook
[22] http://www.scipy.org/NumPy_for_Matlab_Users
[23] http://mathesaurus.sf.net
[24] http://www.astropython.org/
[25] http://www.youtube.com/watch?feature=player_embedded&v=F4rFuIb1Ie4

- Useful software in the Scientific Python Ecosystem[26]
- IPython[27]
- Python(x,y)[28]
- Basic Motion Graphics with Python[29]

Some relevant Python books.

- Think Python[30]
- Learn Python The Hard Way[31]
- Dive Into Python[32]
- Think Like a Computer Scientist[33]
- Introduction to Python Programming[34] (for scientists)
- A Primer on Scientific Programming with Python[35] (slides[36] are also available)
- Python Scripting for Computational Science[37]

Course material on Python programming in general.

- The Official Python Tutorial[38]
- Python Tutorial on tutorialspoint.com[39]
- Interactive Python tutorial site[40]
- A Beginner's Python Tutorial on wikibooks.org[41]
- Python Programming on wikibooks.org[42]

[26] http://fperez.org/py4science/starter_kit.html
[27] http://ipython.scipy.org/doc/manual/html
[28] http://code.google.com/p/pythonxy/wiki/Welcome
[29] http://www.yourmachines.org/tutorials/mgpy.html
[30] http://www.greenteapress.com/thinkpython/
[31] http://learnpythonthehardway.org/book/
[32] http://diveintopython.org/toc/index.html
[33] http://www.freenetpages.co.uk/hp/alan.gauld/
[34] http://femhub.com/textbook-python/python_en.pdf/
[35] http://goo.gl/SWEQlz
[36] http://hplgit.github.io/scipro-primer/slides/index.html
[37] http://goo.gl/q8tM7D
[38] http://docs.python.org/2/tutorial/
[39] http://www.tutorialspoint.com/python/
[40] http://www.learnpython.org/
[41] http://en.wikibooks.org/wiki/A_Beginner's_Python_Tutorial
[42] http://en.wikibooks.org/wiki/Python_Programming/

- Non-Programmer's Tutorial for Python on wikibooks.org[43]
- Python For Beginners[44]
- Gentle Python introduction for high school[45] (with associated IPython notebooks[46])
- A Gentle Introduction to Programming Using Python[47] (MIT OpenCourseWare)
- Introduction to Computer Science and Programming[48] (MIT OpenCourseWare with videos)
- Learning Python Programming Language Through Video Lectures[49]
- Python Programming Tutorials Video Lecture Course[50] (Learners TV)
- Python Videos, Tutorials and Screencasts[51]

[43] http://en.wikibooks.org/wiki/Non-Programmer's_Tutorial_for_Python_2.6
[44] http://www.pythonforbeginners.com/
[45] http://intropython.org
[46] https://github.com/ehmatthes/intro_programming
[47] http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-189-a-gentle-introduction-to-programming-us
[48] http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-00-introduction-to-computer-science-and-pro
[49] http://www.catonmat.net/blog/learning-python-programming-language-through-video-lectures/
[50] http://www.learnerstv.com/Free-Computers-Video-lectures-ltv163-Page1.htm
[51] http://showmedo.com/videotutorials/python

Index

doc strings, 8

list comprehensions, 14

pickling, 13