

# OPEN SOURCE SOFTWARE RESOURCES FOR NUMERICAL ANALYSIS TEACHING

MICHAL KAUKIČ

ABSTRACT. In this article we bring some remarks about use of Open Source Software in teaching of Numerical Analysis based on our experience with Matlab, Octave, and Python-based software systems. We discuss the merits and drawbacks of systems used. Finally, we settled on using Python. This general purpose object oriented language contributes not only to clearer understanding of Numerical Analysis concepts, but also results in deeper insight into Computer Science and Programming principles.

## 1. INTRODUCTION

In real life one seldom uses complicated formulas as the result of analysis and solving of practical problems. This is the matter also with engineering applications, where concrete *numerical output* is desired. We may use theoretical apparatus from calculus, algebra, discrete mathematics (e.g. integrals, limits, derivatives, matrices, vectors, graphs and networks, linear programming) to derive the suitable model for our situation. But the next step, obtaining useful output from given input data, is carried by *numerical algorithms* involving inexact, approximate computations using real hardware and software. This is the main area of study for Numerical analysis — the very specific branch of mathematics, dealing with materialisation of theoretical ideas in practically applicable algorithms. In this article we will speak about teaching Numerical analysis for Engineering students.

There are two approaches to teaching numerical methods. The first one is ‘just in time’ approach (teaching relevant numerical methods as part of theoretical course, e.g. numerical evaluation of definite integrals in course of calculus and methods for finding polynomial roots in algebra course). This is the approach, recommended by European Society for Engineering Education (SEFI) in the document *Mathematics for the European Engineer* [1]. The second approach is to teach numerical methods in specialised courses devoted to Numerical analysis.

We prefer to have specialised courses with Numerical analysis subjects, because of the very specific nature of numerical computations. If integrated into theoretical subjects, only few elementary numerical methods can be presented. That are often ‘brute force’ or inefficient methods, leaving students with very distorted image of true values of Numerical analysis. In theoretical courses, exactness is of utmost importance. On the other hand, Numerical analysis deals with imperfection and fuzziness of real world. In numerical computations, there are only rare cases, when certain quantities can be clearly determined. We cannot afford to make lengthy computations with arbitrary precision, so rounding errors will inevitably appear.

When we get as a result the decimal number 0.0000000021534 all we can say is that our true answer is a very small number, maybe zero. But – is it zero, small negative or small positive? Clearly, to answer similar questions, we should have some knowledge about estimated error of final result. This is often non-trivial task and on such problems the essence of Numerical analysis can be shown.

Another specific kind of problems caused by imperfection of computational tools (hardware and software) are the problems of stability of numerical computations. There is no problem for linear algebra specialist to show the existence and uniqueness of solution vector for arbitrary large system of  $n$  linear equations with  $n$  unknowns, if all the equations are linearly independent. In practice, we may encounter substantial difficulties for 15 equations with 15 unknowns, if the equations are ‘nearly dependent’. The rounding errors, practically invisible in input coefficients of linear equations, can be magnified by the unstable numerical computations to such extent, that the result is very far from true solution. Thus, the kind of problems studied in theoretical courses and in Numerical analysis are completely different.

Effective teaching of Numerical Analysis is closely related to the problem of using computers as the laboratory, experimental equipment, in the sense which is well known from other natural sciences. Numerical analysis must take into account the real world imperfections, which can be happily ignored by theoreticians. Thus, in the field of Numerical analysis there is a big room for experiments. We believe that numerous computer-based experiments are essential for successful teaching of Numerical Analysis.

## 2. AVAILABLE SOFTWARE FOR TEACHING NUMERICAL ANALYSIS

The choice of appropriate software tools is of the fundamental importance. This is often not perceived as such because of ‘automatism’ and inertia caused by tradition, software and hardware platform constraints. The choice of tools, once made, determines the effectivity and long-term maintainability of research and educational activities for many future years.

On the first look, we probably find three well-known commercial software systems for mathematical teaching and research: MATLAB, Mathematica, Maple<sup>1</sup>. Maple is very good for symbolical manipulations (e.g. to compute limits, integrals, derivatives by formulas, to simplify algebraic expressions). MATLAB is simple matrix-oriented language, very efficient for solving large problems by numerical methods (large systems of linear equations, also with sparse matrices, many toolboxes e.g. Symbolic Math, Optimisation, Statistics, Neural networks, Signal and Image processing. . .). Mathematica claims to be an ‘universal system for doing Mathematics on computer’. All three systems have good graphics capabilities (for 2-dimensional and 3-dimensional graphics). MATLAB was our first choice, when we pioneered teaching of Numerical analysis on computers about 12 years ago.

Besides of certain advantages, commercial software systems have also many drawbacks. Let us mention some of them (see also [3]):

---

<sup>1</sup>Maple, MATLAB, Mathematica are registered trademarks of their respective owners (Waterloo Maple Inc., The MathWorks, Inc., Wolfram Research, Inc.). Other trademarks mentioned in this article (e.g. MS Windows, UNIX, Linux, Free BSD, Open BSD) are registered trademarks of their respective owners too.

- small adaptability to special needs of individual user or smaller groups of users, closed model of development, lack of source code availability
- dependence on the software-vendor (expensive upgrades, new versions and extensions)
- weakness and unnecessary complicated interface in application areas not initially envisioned and planned by system designers
- vendor-defined functionality, problems with extending and improving the system beyond the predefined capabilities
- design deficiencies of commercial systems cannot be quickly improved for backward compatibility reasons; patches or service-packs are issued, which implies decreased reliability (on the user side) and maintainability (on the vendor side).

Why we choose MATLAB? This system was widely used in academic and enterprise institutions. There were great chances that (especially for the field of Electrical Engineering) students can use the MATLAB's simple programming language for real engineering problems outside of Mathematics and Numerical analysis courses. With (optional) Symbolic Math toolbox MATLAB has also the desired symbolic manipulation capabilities. We did not liked the complicated (and in our opinion, unnatural) style of Mathematica language.

Our search for alternative software lead us to several Open Source systems with MATLAB-like capabilities. The most MATLAB-compatible among them was Octave [4]. We think that the computational capabilities of Octave are equal (or, in some areas, better) than those of MATLAB. The only weakness is the lack of sophisticated graphical capabilities. There were several attempts to improve Octave graphics, none of them fully successful. Despite this, we can recommend this package as valuable MATLAB substitute.

In course of using MATLAB and later Octave we constantly perceived the apparent deficiencies of underlying simple language. Moreover, programming in MATLAB can be done in very bad style by inexperienced students, not encouraging to learn new ways of thinking about problems. The MATLAB language simply is not general enough to be useful outside of its primary domain – matrix and vector manipulations.

Although there is no universal system suitable for all areas of mathematical education, we should minimise the number of different tools used. The choice should be made so that students can reuse our tools in possibly widest area of their future professional career (having in mind preferably the students of Computer Science at our present workplace - Faculty of Informatics).

This brings us to the idea of using some of general purpose programming languages. The language of our choice should allow to express mathematical ideas with minimum programming effort and in 'mathematics-like' form. For this reason, PASCAL, C/C++, FORTRAN, even Java or Perl are not our favourites.

In recent five years we decided to give the try to programming language Python (see [5]), designed by Guido van Rossum, holder of Master's degree in Mathematics and Computer Science from the University of Amsterdam. We prefer Python for many reasons:

- exceptionally clear and simple syntax

- user-friendly interactive environment (Python shell) for easy rapid development and debugging of Python programs
- many extensions (modules) and applications for use in Mathematics and Engineering; especially for Numerical analysis, there are valuable modules Numeric (Numarray) [6] and SciPy [7]
- for efficiency, there is the possibility of extending Python via interfacing with C/C++ (FORTRAN) code or existing libraries.

### 3. BRIEF PYTHON OVERVIEW FROM MATHEMATICIANS VIEWPOINT

For interactive work, we recommend use of IPython Enhanced Interactive Python shell (see [8]), which is available for both UNIX-like operational systems and MS Windows. Here is an example IPython session (user input is given on lines beginning with In [ ]: marks, other text is the output produced by Python interpreter):

```
In [1]: x=1; s="We have: x="+str(x); print type(x), type(s)
<type 'int'> <type 'str'> We have: x=1
```

```
In [2]: A=[[1,2.0],[0,-1],[-3,s]]; print A, type(A)
[[1, 2.0], [0, -1], -3, 'We have: x=1'] <type 'list'>
```

```
In [3]: A.          (TAB pressed...)
...          (some rows removed)          ...
A.append          A.count          A.extend
A.index           A.insert          A.pop
A.remove          A.reverse          A.sort
```

```
In [4]: A.append?
Type:          builtin_function_or_method
Docstring:
    L.append(object) -- append object to end of list L
```

```
In [5]: (s+' ')*3
Out[5]: 'We have: x=1 We have: x=1 We have: x=1'
```

```
In [6]: [0]*10 + [1]*5
Out[6]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
```

Now some explanations. Python has no explicit type declarations of variables. Instead of writing `int a=1;` as in C, we can simply say `a=1`. Python is *object oriented* language, i.e. most of time we are dealing with *classes* and their instances (Python objects). Each object knows his *type* as can be seen on input lines In [1], In[2] where we created variables `x`, `s`, `A`. This type is determined by ‘birth certificate’ of object. So, `10` is of integer type, `-2.7` of float type, `[]` (empty list) of list type and `'adamiani'` of string type. Variables of type `str` (as is our variable `s`) are intended for holding text (essentially arbitrarily long).

Variable `A` is of type `list`, i.e. the finite sequence of (heterogeneous) ‘things’ (Python objects) indexed by nonnegative integers. Instead of writing  $A_0, A_1, A_2, \dots$ , in Python we use `A[0], A[1], \dots`. This notation is also valid for strings, e.g. `s[3]` is the character ‘h’. Because `A[3]` is the string `s`, then `A[3][1]` is ‘e’. This is very similar as in C or PASCAL.

The input line `In [3]` is also interesting. Here we typed variable `A` of list type followed by dot and pressed `Tab`. In human language it means: ‘Give me please the names of all functions (methods), which can operate on object of type `list`’. And we got them. One of methods for `list` object is `append`, for which we can retrieve further information and help, as is shown on line `In [4]`. The functions (methods) of objects are called by ‘dot notation’, e.g. `A.append(s)` will append string `s` to the end of list object `A` (i.e. `A[4]` will become `s`).

Many operators are conveniently redefined for several types. On lines `In [1]`, `In [5]`, `In [6]` we can observe the actions of binary operators `*`, `+` on strings and lists.

Let us show how to define our own functions in Python. We can do this in Python shell, but any significant piece of code better should be written in text editor (like Notepad, but not text processor like Word, which generates binary, unreadable files) and saved for later use. Assume, we have created text file `fib.py` with following contents:

```
def fib(n):                                # Line 1
    assert (type(n) in [int, long]) \
           and (n >= 0)                    # Line 2

    pred = act = 1                          # Line 3
    for k in range(2, n+1):                # Line 4
        pred, act = act, pred + act        # Line 5
    return act                              # Line 6
```

This is Python implementation (definition) of function `fib(n)` returning the  $n$ -th element of well-known Fibonacci sequence  $\{F_n\}_{n=0}^{\infty}$  defined recursively by

$$F_0 = F_1 = 1; \quad F_{k+1} = F_k + F_{k-1} \text{ for } k = 1, 2, \dots$$

After saving to file `fib.py` we can write `run fib.py` in IPython and call our new function as e.g. `fib(0)`, `fib(10)`, `fib(2003)`.... Note that computation of `fib(2003)` took 0.0056 sec. on Pentium IV /1700Mhz Linux computer. The result has 418 decimal digits. This is also the proof of fact that **Python can do computations with arbitrarily long integers**.

In Python, **blocks of code are determined by indentation**, no `begin`, `end` keywords as in PASCAL or curly braces `{ }` as in C/C++. This is unlike any other language, but makes the structure of programs very readable and clear. We are convinced that ‘indentation’ convention acts also very beneficially on one’s programming style (helps to avoid complicated nested chains of loops and conditional statements or lengthy functions). Of course, there is an initial barrier acting against use of indentation, but our experience shows that students are quickly accustomed to new Python conventions. The only golden

rule, which cannot be overestimated is: *Do not mix spaces and Tab-s in your source code; for indentation, use spaces only.*

We assume that input parameter (independent variable) `n` is nonnegative integer, which is expressed by `assert` statement on line 2. If we try to call the function `fib` with inappropriate argument, e.g. `fib(1.3)` we end with assertion error. From line 2. we can also deduce that logical operators have ‘human readable’ names: `and`, `or`, `not` and that we can test membership ( $x \in L$ ) by statement: `x in L`, where `L` is variable of list type.

Lines 3. and 5. show multiple assignment; `pred = act = 1` is equivalent to `pred = 1; act = 1`, but `pred, act = act, pred + act` is not the same as `pred = act; act = pred + act` because in the latter statement the value of variable `pred` was altered before the second assignment.

Line 4. is the *loop statement*. The function `range(m,n)` creates list of consecutive integers `[m, m+1, ..., n-1]`. The `return` statement on line 6. tells us, what object should be returned from the function. And, as the reader probably already noted, all text from hash mark `#` to end of line is *comment*, ignored by interpreter but useful for human readers.

Python is very high level language, offering convenient means for expressing mathematical facts, e.g. we can write `1 <= x <= 5.5`, just like in mathematical notation. In mathematics we often see constructions like this one:

$$L = \{x \in M : V(x)\}$$

meaning that  $L$  is the set of all values  $x \in M$  satisfying the condition  $V(x)$ . Python has in fact more powerful construction (called *list comprehension*):

$$L = [f(x) \text{ for } x \text{ in } M \text{ if } V(x)],$$

where `f` is arbitrary Python function or expression, containing `x`. How this works, should be clear from next few lines of an IPython session:

```
In [1]: M=range(11) # creates list M=[0,1,2,...,10]
```

```
In [2]: [k for k in M if k%3 == 0]
```

```
Out[2]: [0, 3, 6, 9]
```

```
In [3]: [k**2 for k in M if abs(k-5)>2]
```

```
Out[3]: [0, 1, 4, 64, 81, 100]
```

Note that `k**2` is Python way of writing  $k^2$ .

Most of the Python’s functionality is available in *modules*; e.g. for accessing mathematical functions `sin`, `cos`, `log`, ... we must first import them from module `math`:

```
import math
```

```
from math import sin, cos, log
```

```
s=sin(1.5); c=cos(-1.7) # sin, cos were imported
```

## 4. USING PYTHON FOR TEACHING OF NUMERICAL ANALYSIS

What makes Python suitable for doing Numerical Analysis are the modules `Numeric` (see [6]) and `SciPy` (see [7]). Module `Numeric` introduces new Python type `array`, which is efficient multidimensional array of numbers. In `Numeric` we can use many functions for vector and matrix manipulation, linear equations solving, computing of eigenvalues and eigenvectors. Module `SciPy` extends the capabilities of `Numeric` in many areas e.g. data interpolation and approximation, numerical integration, optimisation, solving of nonlinear equations and systems, solving of ordinary differential equations, Fourier transform, functions for Probability and statistics. Also the `matplotlib` (cf. [10]) module for two-dimensional plotting is very handy. With installation of mentioned modules and `IPython` shell we get MATLAB-like interactive environment very suitable for Numerical analysis teaching and experiments. Unlike in MATLAB, students can use powerful general purpose language Python for creation of their scripts. Note that `IPython` should be invoked by command:

```
ipython -pylab,
```

so we have access to `matplotlib` graphical capabilities and to MATLAB-like functions.

We prefer to work in operating system Linux; doing the same things in MS Windows brings many unnecessary complications – unless we are using Cygwin UNIX-like environment. The client machines in our laboratory are cheap PC-s with good graphics capabilities and good monitors; perhaps, they can be diskless stations. Client computers can boot Linux using network connection to server and the software which is installed centrally on server. Every student working on local computer in laboratory, essentially works on server. This model has many advantages which we will not further discuss here.

Below we bring some typical commands used in our introductory Numerical analysis course with explanatory comments:

```
from time import time           # needed for timing of computations
from scipy.linalg import solve   # import linear equations solver

# ----- Linear Equations -----
A=rand(1000,1000); b=rand(1000,1) # generate random 1000 x 1000
                                # matrix and right hand side vector
t=time(); x=solve(A,b); time()-t # solve system, get solution time

#----- Interpolation and approximation -----
x=linspace(0,2*pi,9); y=sin(x)   # generate equidistant data vector
                                # of 9 values in interval (0, 2*pi)
y=sin(x)                         # take a simple function values
                                # y=sin(x) for vector x
xx=linspace(0,2*pi,100)         # fine vector of 100 values
                                # in (0, 2*pi),
                                # needed for plotting
```

```

yp=polyfit(x,y,xx,8)      # polynomial interpolation
                           # of data points (x,y)
y4=polyfit(x,y,xx,4)     # Least Squares polynomial approximation
                           # of smaller degree n=4
plot(x,y,'o',xx,yp,xx,y4) # plot data points as small circles
                           # plot both interpolation
                           # and approximation curves (see below)

```

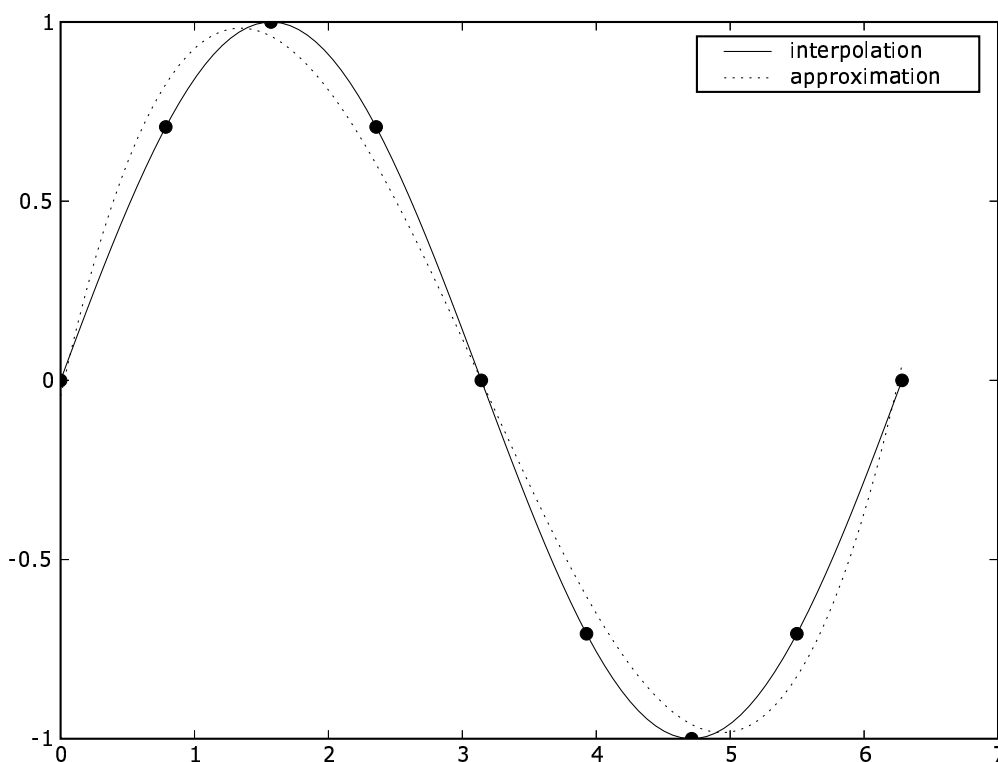


FIGURE 1. Interpolation and approximation curves

This figure shows some features of module `matplotlib`. As in MATLAB, we have plenty of styles for both lines and points markers and we can place legend with explanation of figure components. The possibilities for customising figures are really great. We can dynamically change attributes of axis system, lines (width, colour, style,...) and another graphical objects, which are parts of figure. We present the graphical capabilities of this Python-based development environment to the students from the very beginning, to encourage them to experiment with visualisation of Numerical analysis facts and objects.

```

#----- Numerical integration -----
def fcn(x,y):
    return exp(-(2*x**2 + y**2))

yd=lambda x: x*(x-1)
yh=lambda x: 2*x

I2=dblquad(fcn,0,1,yd,yh)

#----- Systems of nonlinear equations -----
def nel1(xy):
    x,y=xy
    return array([x+2*y, x*x+y*y-4])

r=fsolve(nel1,array[(2,3)])
nel1(r)

```

We hope that the reader got some feeling of what can be done in this Python-simulated MATLAB-like environment. The conclusions we can draw from our experience with Python as teaching aid for Numerical analysis are, that Python language with suitable modules and user-friendly environment can be successfully used instead of commercial software. This has the additional benefit of learning a simple, useful general-purpose language, which students can use later in their career (important for students of Computer Science but also for general Engineering students).

Next, we consider to prepare the teaching material with computer-based learning also for Calculus and Algebra topics. We plan to start with experimental teaching this year, in the winter semester of Calculus and Algebra courses for first year students of our faculty.

#### REFERENCES

- [1] SEFI Math. working group, *Mathematics for the European Engineer*, SEFI HQ, 2002
- [2] M. Kaukič, *Open Source Software in Mathematical Education*, Proc. of I. International Conference APLIMAT 2002, Bratislava, pp. 233-238
- [3] Erika Gyöngyösi, *The More Effective Use of Computers in Teaching Mathematics*, International Journal for Mathematics Teaching and Learning, 2002
- [4] J. W. Eaton, *Octave Manual*, available on <http://www.octave.org/docs.html>
- [5] G. van Rossum, *Python Documentation*, available on <http://www.python.org/doc>
- [6] D. Asher, P. F. Dubois, K. Hinsen, J. Hugunin, T. Oliphant, *Numerical Python*, available on <http://numpy.sourceforge.net>
- [7] SciPy Developers, *SciPy*, available on <http://www.scipy.org>
- [8] F. Perez, *IPython - An enhanced Interactive Python*, available from <http://ipython.scipy.org>
- [9] M. Kaukič, *Numerical analysis with Python*, Proc. of II. International Conference APLIMAT 2003, Bratislava, pp. 417-422
- [10] J. D. Hunter, *Matplotlib*, available on <http://matplotlib.sourceforge.net>