PROBLEM SET 3

F. M. Ramazanoğlu

Due Wednesday, November 4, 17:30

Please Read! Guidelines and some Python help

- From now on you should put all the relevant python code for a given problem in a module (a `.py` file that contains some functions) such as `ps0309.py` which should have the function `mytests` that provides any plots or results you used in your solution for that problem (as in the sample Python files we provided). We will put a sample file on the Blackboard assignment page.
- You should also have the module `ps03all.py` which should provide all the material you use for all problems in the problem set, it should basically call the `mytests` functions from individual `ps03xx.py` files.
- Individual solution files as well as `ps03all.py` should work as scripts, i.e. if we run `python ps03all.py` on the terminal (not the Python interactive environment), we should also get all your results.[1]
- There is one change in style from MATLAB: our Python functions are named with all lower case letters and sub-scores, just like our variables. Also follow the style suggestions at the official Python tutorial.
- You will be repeating what you did in MATLAB on Python in this problem set. Do you wonder the counterparts of MATLAB expressions and functions on Python? SciPy people have thought about this and prepared a guide. Note that there are two commonly used linear algebra-specific sub-packages in Python: `numpy.linalg` and `scipy.linalg`. Both are similarly useful for basic needs, but `scipy.linalg` tends to be the more capable one as you can also see at the beginning of its documentation. For example, it has a special linear solver for banded matrices, `scipy.linalg.solve_banded`, whereas there is only the general purpose `numpy.linalg.solve` in NumPy. Use the former when appropriate, and look for even further specialized solvers. Python does not have the magic of MATLAB's `A\b` we discussed in class where the best method is chosen automatically for `A`, rather, you need to choose a special solver if you know that your matrix is special.
- There is also a sparse matrix package `scipy.sparse` with its own linear algebra submodule `scipy.sparse.linalg`. A banded matrix (e.g. a tridiagonal one) is technically sparse, but from what I read in the internet, the general suggestion is to use the specific functions in `scipy.linalg` in such cases, and use `scipy.sparse` for more general sparse matrices. If dealing with sparse matrices is an important part of your work, my ultimate advise would be trying both and seeing for yourself.

In this problem set you will repeat the problems from PS02 with a few additions, but this time using Python rather than MATLAB.

Problem 8 (6 points)

Repeat problem 5 using Python. Python also has special functions for sparse matrices, follow our advice in the guidelines in relation to them.

Problem 9 (6 points)

(a) Repeat problem 6 using Python. This time, examine the function in the interval $(-1/e\ 10^{20})$. You are free to use the sample files we provided.

(b) In lecture we mentioned how to make a function of a single variable into a function of a NumPy array, and emphasized that writing the function in a general way to admit arrays rather than using `numpy.vectorize` is the way to go. Show this explicitly by timing two different implementations of

[1] Here is a popular explanation of this script stuff: https://stackoverflow.com/questions/419163/what-does-if-name-main-do

`lambertw_custom`, one using vectorization and one inherently array-based.[2] NumPy, like MATLAB, is a lot faster when you do things as vector operations rather than using loops.

(c) Koç W function is defined as $KW(x^2 e^{x^2}) = x$. Plot this function in the interval $(-10^{20}\ 10^{20})$. If you do this the smart way, you can also test your implementation against builtin functions even though there is no builtin Koç W function on any Python package.

Problem 10

Repeat problem 7 using Python.

[2]See this stackoverflow question for some tests about this issue.