High Performance Scientific Computing

Coursera Edition

Homework 2

Note: Coursera homework is not submitted or graded.

The goals of this homework are to:

- Get some experience with Python and *numpy*.
- Get experience writing a Python function and unit test.

Before tackling this homework, you should read some of the class notes and links they point to. In particular, the following sections are relevant:

- *Python*
- *Numerics in Python*
- *nose – Python unit testing framework*

1. **Polynomial interpolation.** Suppose we want to determine the quadratic polynomial $p(x) = c_0 + c_1 x + c_2 x^2$ that passes through three given data points (x_i, y_i) for $i = 1,\ 2,\ 3$.

   Requiring that $p(x_i) = y_i$ for these three values gives a linear system of 3 equations in 3 unknowns (the coefficients c_i).

   For example, if the data points are $(-1, 1),\ (0, -1),\ (2, 7)$ then the system of equations is

   $$\begin{aligned} c_0 - c_1 + c_2 &= 1 \\ c_0 + 0c_1 + 0c_2 &= -1 \\ c_0 + 2c_1 + 4c_2 &= 7 \end{aligned}$$

   This has the form $Ac = y$ where y is the vector of data, c is the vector of coefficients we seek, and A is the 3×3 matrix

   $$A = \begin{bmatrix} 1 & -1 & 1 \\ 1 & 0 & 0 \\ 1 & 2 & 4 \end{bmatrix}.$$

   To solve this system we can use the *solve* function from the *numpy.linalg* module, as illustrated in the Python script found in *$UWHPSC/codes/homework2/demo1.py*. The solution is $c = [-1, 0, 2]$ corresponding to the

This Page
Show Source
Quick search
Go
Enter search terms or a module, class or function name.

polynomial $p(x) = 2x^2 - 1$, as is easily verified.

This code also plots this polynomial along with the three data points that it interpolates.

Produce a script *hw2a.py* that solves the same problem but for the data points $(-1, 0)$, $(1, 4)$, $(2, 3)$. You can start with the *demo1.py* script and modify it appropriately.

The sript should print out the results in exactly the same form as from *demo1.py* and should also produce a similar plot. The script should save this plot as a png file *hw2a.png*.

Add *hw2a.py* to your repository, but not the output or png file.

2. Make this script more general by turning it into a module. You can start with *$UWHPSC/codes/homework2/demo2.py*. Copy this to *hw2b.py* in your own repository and then fill in the missing code. The resulting module will contain a function that can be used as follows, for example:

```
>>> import hw2b
>>> from numpy import array
>>> xi = array([-1.,  0.,  2.])
>>> yi = array([ 1., -1.,  7.])
>>> c = hw2b.quad_interp(xi,yi)
```

This should compute the expected coefficient vector c. The above test is also performed if you do:

```
>>> import hw2b
>>> hw2b.test_quad1()
```

Or if you install *nose*, you can run this test from Unix via:

```
$ nosetests -v hw2b.py
```

This sample module also has a “main program” that is executed only if you run the python code as a script, e.g. by:

```
$ python hw2b.py
```

or with the “run” command in IPython. Many Python modules will have a section like this at the end.

To set up the linear system, you will need to define the 3×3 matrix A in terms of the given *xi* points. You can use the approach used in *demo1.py* to specify the 9 elements of the *np.array*, but a better approach that generalizes more easily to the problems below is to set:

```
A = np.vstack([np.ones(3), xi, xi**2]).T
```

The *np.vstack* function takes as its argument a list of numpy arrays and stacks them vertically into a matrix. The first element of this list is *np.ones(3)*, which is *array([1,1,1])* (all ones, of length 3). So *np.vstack([np.ones(3), xi, xi**2])* is an array whose first row is all ones, the second row contains the values from *xi* and the third row contains the squares of these values. The *.T* at the end takes the transpose and turns this into the matrix

$$A = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{bmatrix}.$$

Convince yourself that this is the correct matrix for the interpolation problem, and experiment in IPython with the functions *np.ones* and *np.vstack* if you are unsure what they do.

Note: Unlike Matlab, there is no distinction between a row vector and a column vector for a 1-dimensional numpy array.

3. Add a new function *plot_quad* to your module *hw2b.py* that takes two numpy arrays *xi* and *yi* of length 3, calls *quad_interp* to compute *c*, and then plots both the interpolating polynomial and the data points, and saves the resulting figure as *quadratic.png*.

   Note that you will have to decide what range of x values to use for evaluating the interpolating polynomial. Since we want the polynomial to cover the range of the data points, use:

   ```
   x = linspace(xi.min() - 1,  xi.max() + 1, 1000)
   ```

4. Test your code by trying various choices of *xi* and *yi* to convince yourself that it is working. Add at least one more unit test function *test_quad2* to the module that corresponds to a different test. This homework will be graded by testing your function on other inputs, so please make sure it works well.
5. To think about: What happens if *xi = array([1., 1., 2.])* is specified as one of the input parameters? Why does the code raise an exception in this case? You do not need to turn anything in or modify the code to deal with such cases, but you should understand why this input is bad and what other inputs would be similarly bad.
6. Add two new functions *cubic_interp* and

plot_cubic that solve the interpolation problem and plot the results if the inputs *xi* and *yi* are of length 4 and we determine the cubic polynomial $p(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3$ that interpolates these 4 points. This requires solving a linear system of 4 equations for the 4 unknown coefficients.

It should produce a png file *cubic.png*.

Add at least one unit test *test_cubic1* to test this code.

7. Add two new functions *poly_interp* and *plot_poly* to the same module *hw2b.py* that generalize the above functions to accept arrays *xi* and *yi* of any length *n* (You should check that *len(xi) == len(yi)*). Assuming the *xi* values are distinct, this data will define a unique polynomial of degree *n-1* and the coefficients can be determined by solving an *n times n* linear system.

   Note: High-order polynomial interpolation has various numerical difficulties associated with it that we will not explore in this class. Also this approach of setting up and solving an $n \times n$ linear system is not the best way to compute the interpolating polynomial. But the point here is to work on Python coding.

   Note: To plot the polynomial you will have to evaluate it at many points. For a polynomial of higher degree this is best done using "Horner's rule": If the coefficients are in *c* with *len(c) == n* and *x* is the array of points to evaluate it at, then use:

   ```
   y = c[n-1]
   for j in range(n-1, 0, -1):
       y = y*x + c[j-1]
   ```

   Try *range?* in IPython to learn what that does and figure out why this loop works!

   Test your program with various inputs and write at least two unit tests *test_poly1* in which $n = 4$ and *test_poly2* in which $n = 5$.

   Note: The numpy functions *polyfit* and *polyval* do something similar to what's required here, and might be useful for comparison purposes. But note the coefficients in the polynomial are returned in a different order! *polyfit* also does least squares fitting if the degree specified is less than *n-1*.

8. Add and commit all required codes to your bitbucket repository.

   Note: At the end you should have the following files committed to your repository:

- $MYHPSC/homework2/hw2a.py
- $MYHPSC/homework2/hw2b.py

Contents | Bibliography | index