APC 524 / AST 506 / MAE 506

Software engineering for scientific computing

Assignment 3: Root finding and automated testing

Assigned: 18 Oct 2016
Due: 28 Oct 2016, 11:55pm

A common task in scientific computing is to find a root of a particular function. That is, for a vector-valued function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$, we want to solve

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}, \qquad \text{for } \mathbf{x} \in \mathbb{R}^n. \tag{1}$$

In this assignment, we will give you an implementation of a simple root finder (Newton's method) and an example test. You will then extend this implementation by writing additional functionality and tests.

For this assignment, you are required to use the Python programming language (`http://www.python.org/`), and the `numpy` library (`http://scipy.org/`). Additionally, you will keep track of your changes using the Git version control system through GitHub (`http://github.com`).

Newton's Method

There are many methods for numerically approximating a root of a function, but here we focus on Newton's method. This method uses linear approximations of the function to iterate towards an approximation of the root. Typically the approximation converges quadratically: the number of accurate digits roughly doubles every iteration. However, a good initial guess is needed, and if the initial guess is not close enough to an actual root, the method may fail to converge.

One dimension

First consider the one-dimensional version, and assume that the first derivative of $f(x)$ is continuous and nonzero. Newton's method is given by the following algorithm:

- Make an initial guess x_0
- Loop:
  - Find x_{k+1} with
$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{2}$$
- Stop when $|f(x_k)| < \epsilon$ for some prescribed $\epsilon > 0$.

Higher dimensions

The method generalizes to higher dimensional systems where we seek a solution to $\mathbf{f}(\mathbf{x}) = \mathbf{0}$. Now, the derivative of $\mathbf{f}$ is the Jacobian matrix $\mathbf{Df}(\mathbf{x})$, given by

$$\mathbf{Df}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}. \tag{3}$$

The iteration step then becomes

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{Df}(\mathbf{x}_k)^{-1}\mathbf{f}(\mathbf{x}_k). \tag{4}$$

It is clear that if the Jacobian is singular (non-invertible), then Equation (4) cannot be solved and Newton's method fails. Rewriting the algorithm for arbitrary dimensions:

- Make an initial guess $\mathbf{x}_0$
- Loop:
  - Find $\mathbf{x}_{k+1}$ from Equation (4).
- Stop when $\|\mathbf{f}(\mathbf{x}_k)\| < \epsilon$ for some prescribed $\epsilon > 0$. Here, $\|\cdot\|$ denotes the standard (Euclidean) norm:

$$\|\mathbf{z}\| = \left(\sum_{j=1}^{n} z_j^2 \right)^{1/2}$$

Assignment

First, you will write tests and correct mistakes in the code you are provided with. Second, you'll add new functionality to the root finder and write tests to be sure it works properly.

We have provided you with four files:

- `newton.py` implements the Newton solver
- `testNewton.py` contains an example test for the routines in `newton.py`
- `functions.py` contains a function to approximate the Jacobian matrix needed for the Newton solver, as well as a class `Polynomial` that may be useful in generating other functions for testing the Newton solver
- `testFunctions.py` contains example tests for the routines in `functions.py`

Your specific tasks are as follows:

- First, create a new GitHub repository for yourself, and add the four files we provided. (Your final submission will need to include the GitHub repository, so it is important to do this first.)
- The provided files `newton.py` and `functions.py` both have bugs, and some of the tests provided fail! You'll first need to debug these, so that the provided tests pass.
- Next, add additional tests to check for other potential bugs, fixing the provided files where necessary. Some things to test are:
  - The numerically computed Jacobian is accurate in one dimension and higher dimensions.
  - A single step of the Newton method performs as it should.
  - The roots approximated are correct for a variety of functions of different dimensions (as long as the initial guess is close enough).
  - `Newton` raises an exception if the method fails to converge after the maximum number of iterations.
  - Any other tests you think are useful.
- The version of `newton.py` you were given uses an approximate numerical derivative to find the Jacobian. For many functions, we can compute the Jacobian analytically and improve the accuracy and efficiency. You should add this functionality by altering the `__init__` member function of class `Newton` to include a new optional argument `Df`, which specifies the function to be used for calculating the Jacobian. If `Df` is not specified, the default behavior should be the same as the original version.
- Since you've added a new feature, you'll also need to add corresponding tests. Some ideas:
  - Test that the analytic Jacobian is actually the one used by your root finder.
  - Add some functions and their analytic Jacobians to `functions.py`, including at least one function of a single variable, and at least one function of multiple variables.
  - Add an easy way of testing that a new analytic Jacobian you write is accurate, for instance by comparing with the approximate Jacobian routine. (This test code should, of course, be included in `testFunctions.py`.)
  - Add any other tests you think are useful.
- Add a condition that the approximated root must lie within a radius r of the initial guess $\mathbf{x}_0$, or the iteration loop raises an exception. That is, the code should stop if $\|\mathbf{x}_k - \mathbf{x}_0\| > r$. (You can make use of `numpy.linalg.norm`.)

- Add tests for this new feature, including particular examples and initial guesses that should cause `Newton` to encounter this new condition.

Submission

Your submission must include only the following files (your updated versions, of course):

- `newton.py`
- `testNewton.py`
- `functions.py`
- `testFunctions.py`
- `README`: Explanation of your code, required.

For this assignment you'll submit these files as a bundled Git repository, `hw3.bundle`. To create a suitable bundle, execute the following from within your repository:

```
git bundle create hw3.bundle master
```

More information on Git can be found at `http://git-scm.com/documentation` and in the slides from lecture.

Please only use features available in Python versions ≤ 2.7. You can (and should) use the module `numpy`. If you need to install or update Python or `numpy` on your system we strongly recommend using the `anaconda` distribution (`http://continuum.io/downloads`).

When you are finished, submit the assignment using the CS Dropbox system at `https://dropbox.cs.princeton.edu/APC524_F2016/HW3`