APC 524 / AST 506 / MAE 506

Software engineering for scientific computing

Assignment 3: Automated testing on a root-finder

Assigned: 14 Nov 2018

Due: 21 Nov 2018, 11:55pm

A common task in scientific computing is finding a zero of a function: given some vector-valued function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$, identify one or more solutions of

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

For this assignment, you will be given a Python implementation of a simple multi-dimensional root finder (employing Newton's method) and some tests both of the root-finder itself and of some associated code. You will run those tests in order to diagnose and correct errors in the furnished code. You will then extend the implementation by introducing some additional functionality, as well as tests for those new features.

All code for this assignment must be written in Python 3. The code makes use of the `numpy` package in several places, but I've included comments/notes in the code about those uses. You'll probably need to use `numpy` in a couple of places in your own edits/additions to the code, but I think if you read the comments in the provided files, you should be able to figure out any `numpy` issues.

You must also track your development using your choice of distributed version-control system (git or hg).

Provided files

The provided tarball contains four files:[1]

- `newton.py` — implements the solver as a class called `Newton` with a `solve()` method for finding a root of a function given an initial guess (when using the class, you'd create a separate `Newton` instance for each function $\mathbf{f}$ whose roots you seek).

[1] The organizational structure of these files is far from ideal (I was rushing), and I don't want you to get the impression that this is good file layout. If we imagine that these classes and functions are part of some larger numerical library we're writing, then the `approximateJacobian` function in `functions.py` could be a viable part of that library, on a par with the `Newton` class in `newton.py`. This means that the file `functions.py` is mixing deliverable code (`approximateJacobian`) and helper code (the `Polynomial` class) that's there only to facilitate testing the other stuff. Ideally, we'd have files `newton.py` and `jacobian.py`, plus test modules for each of those, and then a separate file of "test-facilitation code" (that's where the Polynomial class would live) that you would import into each of your testing modules. Or something like that.

Likewise, the layout of the docstrings in these functions/classes isn't standard by any means. I just wanted to put in enough info for you to understand the assignment.

- `testNewton.py` – contains a class (subclass of `unittest.TestCase`) whose methods can test the class in `newton.py` (you are free to add other testing classes to this file if you like[2]). You can run these tests by making the file executable and calling it as a standalone script, by passing it as an argument to the `python3` command, or by using another driver like `py.test`.[3]

- `functions.py` — contains a function for numerically approximating the Jacobian matrix needed for the solver[4] in the `Newton` class (see the background section below). Also contains a class `Polynomial` that may be useful in generating more functions to test the Newton solver.

- `testFunctions.py` — contains some example unit tests for the ancillary code in `functions.py`.

The provided files `newton.py` and `functions.py` both have bugs, and some of the tests provided fail.

Assignment

This assignment has **four major components:**

(i) Version control — create a new repository for yourself and add the four provided files right away. Your final submission must include an exported "bundle" of your repository, so it is important to do this step first.

(ii) Preliminary (and light) debugging — run the tests provided, and fix the code in the provided files so that the provided tests all pass. You can use a symbolic debugger, but it may be overkill. The provided code is short, and the bugs aren't particularly subtle. You can probably find them just by inspecting the code.

(iii) Write more tests to expose more bugs/limitations — compose (and run) additional tests to check for other bugs or, more important, to make sure the code in `newton.py` and `functons.py` is as robust as possible. *Try to break the code in both source files!* Think about edge and corner cases (see the background discussion later in this document), and write tests that you think the code *won't* pass. Then, one test at a time, fix/rewrite/add to the code until the solver passes test the tests you just wrote.

[2] You might have, say, 20 tests to run on the `Newton` class, of which 8 naturally cluster into one group of related tests with common code among them, 5 form a second natural group, and the remaining 7 form yet a third natural group. Then the rule of thumb would be to have 3 separate classes (one with 8 methods, one 5 methods, one 7 methods), all of which we'd still keep in a single file `testNewton.py` because they're all testing code found in `newton.py`.

[3] See, e.g., `https://docs.pytest.org/en/latest/`.

[4] Why not include this function in the `newton.py` module? Because it's less *modular*. You might want the Jacobian code for things besides root-finding. And it should have its own batch of unit tests.

(iv) Add new features, plus tests for those features — there are two specific features you should add:

- *Bound the root*[5] – allow the user to specify a radius r around the initial guess $\mathbf{x}_0$ within which the computed root must lie.[6] In other words, the code should raise an exception if, on any iteration, $\|\mathbf{x}_k - \mathbf{x}_0\| > r$. Implement this by adding another keyword argument (i.e., another "kwarg") `max_radius` to the `__init__` method of the `Newton` class.[7].

  You should, of course, *unit test* this new feature (and all new features)! Try to come up with a function and an initial guess that would cause `Newton` to encounter this new scenario (it's ok to test this feature just in 1D).

- *Support analytic Jacobians* – the provided version of `newton.py` evaluates the Jacobian using an approximate numerical derivative (codified in a function in `functions.py`). But often, we can provide a closed-form expression for the Jacobian, which might be more accurate and/or efficient than a numerical approximation. Add this functionality by adding another kwarg `Df` to the `Newton` class's `__init__` method.[8] If `Df` is not specified by the user, `Newton` instances should use a numerical approximation to the Jacobian just as before.[9]

  Again, unit test this new feature. Exercise it to make sure it works. You should test this feature for at least one 1D function and one 2D function.

Details concerning the tests

There are actually quite a lot of potential problems that "professional-grade" root finders guard against. You're not expected to write a Newton's-method routine of that caliber. But your tests should probe at least some of the potential pathologies of Newton's method, just so you can appreciate the importance of handling these pathologies and the value of test-driven development (TDD) for exposing them to begin with.

[5] The need for this feature might even be exposed by one of the extra tests you end up writing for part (iii).

[6] For ease, and since we're dealing with functions $\mathbf{f}$ that could take vector inputs and yield vector outputs, you might want to use `numpy.linalg.norm` here.

[7] There's probably no default value that will be sensible for functions in general, so just provide some reasonable default and presume that users of the class will have the good sense to set this value properly for their needs.

[8] In case this isn't clear to newbies, the user should set `Df` to a *function*. Remember, in Python, functions are first-class objects that can be passed around just like any other type of object.

[9] You can achieve this requirement by choosing a judicious default value for `Df` and having your code behave appropriately if the user doesn't override that default.

Things you *don't* need to test:

- the `Polynomial` class. I'm mostly providing this for your convenience. Plain old polynomials can expose almost every potential failure point in a root-finder based on Newton's method. The `Polynomial` class is here just to give you a simple interface for building test functions.

Things you *should* test:

- The numerically approximated Jacobian should be fairly accurate in one dimension and higher dimensions. If it isn't, think about how you could modify the code.[10]
- A single step of Newton's method should do what it's supposed to (there's a problematic case here — decide how to handle it, perhaps by "fudging" the guess that just created the problem).
- The calculated roots should be correct for a few different functions of different dimensions (provided the initial guess is close enough). For this assignment, it's ok to test only a few 1D functions and one 2D function. Again, I'm not asking for a professional-grade root finder.
- If the function returns a value, is that the actual root, or did the solver just give up because `maxiter` (the user-specified maximum number of iterations) was exceeded? `Newton.solve()` should raise an exception[11] in the latter situation. To test this, it might be helpful to input a function/guess combo that is known never to converge or to result in an infinite loop.
- Any other tests you think are useful. See the background section below for ideas, but again, you don't have to go crazy testing every scenario under the sun. An extra test or two is fine.

It's ok to test all of the above (or anything else in this assignment) just with polynomials, at least for 1D functions (the supplied `Polynomial` class in `functions.py` may help with this, but you don't have to use it if you don't want to). To test a

[10] One option is to use a **symmetric difference quotient**. The downside of this is that it can mask a point of non-differentiability, but at a point where the function **f** is differentiable, the symmetric difference quotient typically provides a more accurate (for a given $h \equiv \Delta x$) estimate of the derivative

[11] In Python, errors are called "exceptions", and they're also objects. Each such object is an instance of some class such as `ValueError` or `ZeroDivisionError`, and all of those classes are themselves subclasses of a common base class. The general design of exceptions and how to raise them, along with the hierarchy of Python exception classes (i.e. which classes inherit from which others), are described here: `https://docs.python.org/3/tutorial/errors.html` `https://docs.python.org/3/library/exceptions.html`. Check out the list to see what's most appropriate to raise in this situation. There may not be a single best answer, so just make a sensible choice.

2D function, you can supply your own, but even that can be a 2D polynomial if you like.

And remember, every time you add a test that the code doesn't pass, you should modify the code until it *does* pass that test. So don't make your tests too ambitious (but don't make them repetitive or trivial, either).

Submission

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

- `newton.py`
- `testNewton.py`
- `functions.py`
- `testFunctions.py`
- `README`: Required. This should be a plain text file with a brief explanation of what the Newton class does and how it's used. It doesn't have to be long, but it should describe the features (including the ones you added). A few examples/sample lines of use-cases can probably clarify things without your having to write a lot of text.

You may add tests to `testNewton.py` and `testFunctions.py`, but you shouldn't delete any of the ones that are already there. Each time you add a test, all previous tests should still pass. You are welcome, however, to delete comments from any of the files that were just pedagogic commentary from me.

Submit these files as a bundled Git or Mercurial repository called `hw3bundle.git` or `hw3bundle.hg`. I believe the git command to accomplish this is

```
git bundle create hw3bundle.git master
```

but I'm not 100% sure. I also believe you can execute that command from any folder within your versioned project, but again, I'm not sure (it might need to be run from the top folder). To bundle with Mercurial, go one folder above the top of your repository, and execute

```
hg -R /path/to/yourrepo bundle --all hw3bundle.hg
```

That should work without issue.

Once you've finished, please submit the assignment using the CS Dropbox system at `https://dropbox.cs.princeton.edu/APC524_F2018/HW3Roots`

Background on Newton's method

Newton's method is one of many popular algorithms for numerically approximating a root of a function. Given a guess $\mathbf{x}_n$ for the root that turns out to be wrong, the method takes a linear approximation of $\mathbf{f}$ at $\mathbf{x}_n$, finds the root of that approximate linear function, and uses that root $\mathbf{x}_{n+1}$ as the next guess, iterating until $\mathbf{f}(\mathbf{x}_k)$ is close enough to $\mathbf{0}$.[12]

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:

- Specify an initial guess x_0 for the root and a tolerance $\epsilon > 0$ for how close $f(x)$ should be to zero in order to classify the input as a root
- Now loop over the following steps, starting with $k = 0$:
  - Is $|f(x_k)| < \epsilon$? If so, then x_k is the desired root.
  - If not, then use
    $$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{2}$$
    as the next guess.

Typically, Newton's method converges quadratically: the number of accurate digits in the current guess roughly doubles every iteration. However, for a repeated root, the method may converge more slowly. And if the initial guess is not close enough to an actual root, the method may fail to converge altogether, or it may unexpectedly "jump" to a distant region of the domain and converge to a root other than the one being sought. All of these important caveats are factored into "production-grade" solvers that use Newton's method.

Some potential pathologies

Here's a (non-exhaustive) list of issues that Newton's method can run into, even for 1D scalar functions:

- Sometimes, the iterator can send your next guess out to infinity. For instance, say you have a parabola with two roots, and by bad luck, one of your guesses ends up being the vertex.

[12] The Wikipedia article for Newton's method is actually pretty good, and there are a ton of discussions and resources online about the method, how to visualize it, and what corner/edge cases to be leery of.

- For some functions, certain guesses will converge, while others won't. For instance, draw out the function xe^{-x} (which has a root at $x = 0$) and think about what happens if your initial guess is $x_0 = 0.9$ vs $x_0 = 1.1$.
- Newton's method can get caught in infinite loops if you aren't careful. One way is to feed it a function that doesn't actually have a (real) root (try sketching out a function like that). Another more insidious case is when your k^{th} guess turns out to coincide with your $(k-2)^{\text{th}}$ guess: $x_k = x_{k-2}$. Here, you can get caught in an infinite loop that bounces forever between two guesses. A production-level Newton-based root-finder (which, again, you don't need to write for this assignment) would probably try to detect this situation and recover from it by "tweaking" the value of the problematic guess.

The above things are helpful to think about when devising tests. I don't need you to test all or any of these specific things — they're just suggestions in case you're having trouble thinking of things to test.

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. When $\mathbf{f}$ is vector-valued, we also need to replace run-of-the-mill absolute value with the Euclidean norm

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

and stop when $\|\mathbf{f}(\mathbf{x}_k)\| < \epsilon$.

With these changes, the solver loop is the same as for the one-dimensional case. In fact, the 1D case is just the multi-dimensional case, with a 1×1 Jacobian.