Lecture 5: Debugging

Math 98

Reminders and Agenda

- Reminders on Dates
  - This is the last week of class for Math 98.
  - Continue working on HW4 and the project.
- Agenda
  - Using the MATLAB debugger
    - ★ Breakpoints
    - ★ Step and Run
    - ★ Review the MATLAB Documentation
  - Exercise on finding bugs
  - Debugging and Programming Best Practices
    - ★ Sections 2.7-2.8 of http://www.sfu.ca/∼wcs/ForGrads/ensc180spring2016f.pdf

bisectionbuggy.m

Consider this implementation of bisection

```
function p = bisectionbuggy(f, a, b, tol)
    while 1
        p = (a+b)/2;
        if p - a < tol
            break;
        end
        if f(b)*f(p) > 0
            a = p;
        else
            b = p;
        end
    end
end
```

bisectionbuggy.m

There's clearly something wrong with this....

```
>> p = bisectionbuggy(@(x) x, -1, 2, 1e-4)
p =
  1.999908447265625
```

Let's see if we can smoke out the bug with the debugger.

bisectionbuggy.m

Solution: Either change `f(a)*f(p) > 0` or `f(b)*f(p) < 0` or switch `a = p` and `b = p`

Incremental Development

When you start writing scripts that are more than a few lines, you might find yourself spending more and more time debugging. The more code you write before you start debugging, the harder it is to find the problem.

Incremental development is a way of programming that tries to minimize the pain of debugging.

Incremental Development: Three Steps

The fundamental steps of incremental debugging are:

1. Always start with a working program. If you have an example from a book or a program you wrote that is similar to what you are working on, start with that. Otherwise, start with something you know is correct, like $x = 5$. Run the program and confirm that you are running the program you think you are running. This step is important, because in most environments there are lots of little things that can trip you up when you start a new project. Get them out of the way so you can focus on programming.
2. Make one small, testable change at a time. A "testable" change is one that displays something on the screen (or has some other effect) that you can check. Ideally, you should know what the correct answer is, or be able to check it by performing another computation.
3. Run the program and see if the change worked. If so, go back to Step 2. If not, you will have to do some debugging, but if the change you made was small, it shouldn't take long to find the problem.

Unit Testing

In large software projects, **unit testing** is the process of testing software components in isolation before putting them together.

The programs we have seen so far are not big enough to need unit testing, but the same principle applies when you are working with a new function or a new language feature for the first time. You should test it in isolation before you put it into your program.

Unit Testing: Example

For example, suppose you know that x is the sine of some angle and you want to find the angle. You find the MATLAB function `asin`, and you are pretty sure it computes the inverse sine function. Pretty sure is not good enough; you want to be very sure.
Since we know $\sin(0) = 0$, we could try:

```
>> asin(0)
ans = 0
```

which is correct. We also know that sin of 90° is 1, so if we try `asin(1)` we expect the answer 90, right?

```
>> asin(1)
ans = 1.5708
```

What's going on here?

Unit Testing: Example (Cont)

Oops. We forgot that the trig functions in MATLAB work in radians, not degrees. So the correct answer is $\frac{\pi}{2}$, which we can confirm by dividing through by π:

```
>> asin(1)/pi
ans = 0.5000
```

With this kind of unit testing, you are not really checking for errors in MATLAB, you are checking your understanding. If you make an error because you are confused about how MATLAB works, it might take a long time to find, because when you look at the code, it looks right.

```
The worst bugs aren't in your code; they are in your head
```

Debugging in four acts

- **Reading**: Examine your code, read it back to yourself, and check that it means what you meant to say.
- **Running**: Experiment by making changes and running different versions. Often if you display the right thing at the right place in the program, the problem becomes obvious, but sometimes you have to spend some time to build scaffolding.
- **Ruminating**: Take some time to think! What kind of error is it: syntax, runtime, logical? What information can you get from the error messages, or from the output of the program? What kind of error could cause the problem you?re seeing? What did you change last, before the problem appeared?
- **Retreating**: At some point, the best thing to do is back off, undoing recent changes, until you get back to a program that works, and that you understand. Then you can starting rebuilding.