Math 4610 Lecture Notes

Functions as Arguments in Coding Languages *

Joe Koebbe

November 19, 2019

*These notes are part of an Open Resource Educational project sponsored by Utah State University

Functions as Arguments in Coding Languages: An Introduction

A common programming problem involves using functions or methods as arguments in a standalone subroutine, method, or other coding construct. Each and every language or platform (for example, Matlab) has one or two ways to address this oissue. In this section, we will go through a few (but not all) programming languages to document possible ways to use functions as arguments. A simple example will be used to illustrate how to succeed at this. In particular, a machine epsilon code will be used. In C, the code will look like the following.

```
double maxeps()
{
  double one = 1.0;
}
```

Functions as Arguments in Coding Languages: C

There are a number of ways in the C porgramming language to pass in a function as an argument. As an

Functions as Arguments in Coding Languages: Java

Due to the object oriented mature of the Java programming language, it is easy to create objects that contain a function. These notes will point the reader in a direction that will work. Suppose that we decide to apply Newton's method to approximate the roots of a function like

$$f(x) = sin(\pi + 10\ x^2)$$

from some initial point. We will need to encode the function above and it's derivative

$$f'(x) = 20\ x\ cos(\pi + 10\ x^2)$$

which is obtained using an application of the chain rule.
The following code (in Java) implements Newton's method for a generic function.

```
//
// set the class instantiation
// ---------------------------
//
public class SimpleNewtonMethod extends Object {
  //
  // start with the function evaluation itself
  // -----------------------------------------
  //
  public static double newt(FunctionObject fo,
                            double x0,
                            double tol,
                            int maxit) {
    double error  = 10.0 * tol;
    int iter = 0;
    double xold = x0;
    double xnew = x0;
    //
    // start the iteration
    // -------------------
    //
    while(error > tol  && iter < maxit) {
      iter++;
      xnew = xold - fo.fval(xold) / fo.dfval(xold);
      error = Math.abs(xnew - xold);
      xold = xnew;
    }
    return xnew;
  }
}
```

You should notice that there is another object that needs to be included in the compilation before this code will work. The function object coded up in this example might look like the following.

```
//
// set the class instantiation
// ---------------------------
```

```
//
public class FunctionObject extends Object {
  //
  // start with the function evaluation itself
  // -----------------------------------------
  //
  public static double fval(double x) {
    fval = Math.sin( Math.PI + 10.0 * x * x );
    return fval;
  }
  //
  // the following method will compute the derivative of the function at an
  // arbitrary point
  // ---------------
  //
  public static double dfval(double x) {
    dfval = 20.0 * x * Math.cos( Math.PI + 10.0 * x * x );
    return dfval;
  }
  //
  // the following method will compute the second derivative of the function at
  // an arbitrary point
  // ------------------
  //
  public static double df2dval(double x) {
    df2val = 20.0 * Math.cos( Math.PI + 10.0 * x * x )
            - 400.0 * x * x * Math.sin( Math.PI + 10.0 * x * x );
    return df2val;
  }
  //
  // local variables
  // ---------------
  //
  private static double fval;
  private static double dfval;
  private static double df2val;

}
```

Notice that the class is used to hold the information about the function being analyzed. In the code there is (1) a function evaluation, (2) an evaluation of the derivative, and (3) an evaluation of the second derivative for extra information about the function.
It is not difficult to exten this idea to other classes in the Java programming language. The only thing that needs to be changed in analyzing functions is the function definition and the definition of the derivative(s). Note that in order for the Newton method code to see the FunctionObject, you must place the two files containing the code in the same folder and then compile the SimpleNewtonMethod object using

---

```
javac SimpleNewtonMethod.java
```

The compiler will realize there is a dependency and look for the file containing the FunctionObject class in an appropriately named file.

Vector/Matrix Operations: The Euclidean Length of a Vector

In many problems it will be necessary to measure the length or magnitude of a vector (at least so say the villian in Despicable Me). So, we need some way to determine the length. Actually, there are infinitely many ways to do this. The approach most students see first involves using the Euclidean distance between two points used to define a vector. From any standard linear algebra course, the Euclidean length can be defined as follows.

$$||v|| = (v_1^2 + v_2^2 + \cdots + v_n^2)^{1/2}$$

where the notation, $|| * ||$, provides a mathematical notation for the magnitude of the vector. The notation emphasizes the difference between the absolute value of the difference of two real numbers and the length of a vector. This can be implemented easily into a method or routine that will return this length of a vector.

```
public double l2norm(double[] v) {
  double sum = 0.0;
  //
  // extract the length of the vector
  // --------------------------------
  //
  int n = v.length;
  //
  // compute the sum of squares
  // --------------------------
  //
  for(int i=0; i<n; i++) sum = sum + v[i] * v[i];
  //
  // return the value
  // ----------------
  //
  return Math.sqrt(sum);
  //
}
```

Note that the Java package including the square root function defines the last step in the code. It is best to use an intrinsic for the square root since implementing our own version of a square root method would take some time.

Vector/Matrix Operations: Definition of the Norm of a Vector

There are many ways to compute the length of a vector. In this section we will consider several ways to do this. However, we need to have a general definition of what is meant by the length of a vector. To start, the definition of a norm is given.

Definition 1 *Suppose that V is a vector space and* $\mathbf{u}$ *and* $\mathbf{v}$ *are any two vectors in V. Also, assume a is an arbitrary number/scalar. The norm of a vector is a function*

$$||*|| : V \to \Re$$

such that

1. $||v|| = 0$ *if and only if* $\mathbf{v} = \mathbf{0}$,
2. $||a\mathbf{v}|| = |a|\ ||\mathbf{v}||$, *and*
3. $||\mathbf{u} + \mathbf{v}|| \leq ||\mathbf{u}|| + ||\mathbf{v}||$.

The Euclidean magnitude/length of a vector can be shown to satisfy this definition of a norm. This means we can use the terms length, magnitude, and norm interchangeably. The term norm is used in the name, **l2norm**, chosen for the method in the code above.
There are an infinite number of ways to compute norms on vector spaces. A general definition for a norm on n-dimensional real space is the following.

$$||v||_p = (v_1^p + v_2^p + \cdots + v_n^p)^{1/p} = \left(\sum_{i=1}^{n} |v_i|^p \right)^{1/p}$$

for any positive integer p. The Euclidean norm is obtained when $p = 2$. The reality in computational mathematics is that there are three norms of interest including the 2-norm. The other two norms are the 1-norm and the other is the infinity-norm. The definitions for these two norms are given below.

$$||v||_1 = |v_1| + |v_2| + \cdots + |v_n| = \sum_{i=1}^{n} |v_i|$$

and

$$||v||_\infty = \max_{1 \leq i \leq n} |v_i|$$

are the definitions we will use in our work. The gold standard for measuring length is typically the 2-norm. However, in some problems the 1-norm and infinity-norm are easier to compute in many problems.
The 1-norm and infinity-norm can easily be coded into methods as was done above to the 2-norm. Using the 2-norm code, the other two methods just need a couple of minor changes to implement the other two norms into their own methods.

Vector/Matrix Operations: Errors in Vector Approximations

Just as the errors in approximating roots of functions of a single variable, having a way to compute the magnitude of the error in approximating one vector $\mathbf{v}$ by another vector $\mathbf{u}$. We can use the definitions of norms earlier in this section to define a consistent formula for the vector approximation error. If we use a generic definition of the norm of a vector, we can define

$$\text{absolute error} = ||\mathbf{v} - \mathbf{u}||$$

and

$$\text{relative error} = \frac{||\mathbf{v} - \mathbf{u}||}{||\mathbf{u}||}$$

These formulas should look familiar when compared to error measurement in root finding problems.
A code that will implement the absolute error with the 2-norm might look like

```
public double absl2err(double[] u, double[] v) {
  double sum = 0.0;
  //
  // extract the length of the vector
  // --------------------------------
  //
  int n = v.length;
  double diff = 0.0;
  for(int i=0; i<n; i++) {
    diff = u[i] - v[i];
    sum = sum + diff * diff;
  }
  //
  // return the norm of the difference
  // ---------------------------------
  //
  return Math.sqrt(sum);
  //
}
```

Of course, we could reuse code that we have already written as follows. Provided that the methods have been created and test and inserted in some sort of archive that is available for our use. A first simpler version is the following.

```
public double absl2err(double[] u, double[] v) {
  double [] diff = null;
  int n = v.length;
  diff = vecsub(u, v);
  return l2norm(diff);
}
```

An even more concise version of the method might like the following.

```
public double absl2err(double[] u, double[] v) {
  return l2norm(vecsub(u, v));
}
```