Lecture 3

Newton's Method and Loops

Solving equations numerically

For the next few lectures we will focus on the problem of solving an equation:

$$f(x) = 0. \tag{3.1}$$

As you learned in calculus, the final step in many optimization problems is to solve an equation of this form where f is the derivative of a function, F, that you want to maximize or minimize. In real engineering problems the functions, f, you wish to find roots for can come from a large variety of sources, including formulas, solutions of differential equations, experiments, or simulations.

Newton iterations

We will denote an actual solution of equation (3.1) by x^*. There are three methods which you may have discussed in Calculus: the bisection method, the secant method and Newton's method. All three depend on beginning close (in some sense) to an actual solution x^*.

Recall Newton's method. You should know that the basis for Newton's method is approximation of a function by its linearization at a point, i.e.

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0). \tag{3.2}$$

Since we wish to find x so that $f(x) = 0$, set the left hand side ($f(x)$) of this approximation equal to 0 and solve for x to obtain:

$$x \approx x_0 - \frac{f(x_0)}{f'(x_0)}. \tag{3.3}$$

We begin the method with the initial guess x_0, which we hope is fairly close to x^*. Then we define a sequence of points $\{x_0, x_1, x_2, x_3, \ldots\}$ from the formula:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, \tag{3.4}$$

which comes from (3.3). If $f(x)$ is reasonably well-behaved near x^* and x_0 is close enough to x^*, then it is a fact that the sequence will converge to x^* and will do it very quickly.

The loop: `for ... end`

In order to do Newton's method, we need to repeat the calculation in (3.4) a number of times. This is accomplished in a program using a *loop*, which means a section of a program which is repeated. The

simplest way to accomplish this is to count the number of times through. In MATLAB, a `for ... end` statement makes a loop as in the following simple function program:

```
% mysum
% Gives the sum of the first n integers
%
n = 100
S = 0;              % start at zero
% The loop:
for i = 1:n         % do n times
    S = S + i;      % add the current integer
end                 % end of the loop
S
```

Save this program as a **script** named `mysum` and run it by typing `mysum` in the command window. The result will be the sum of the first 100 integers. Next change n to 1,000,000 and run the program again.

All `for ... end` loops have the same format, it begins with `for`, followed by an index (`i`) and a range of numbers (`1:n`). Then come the commands that are to be repeated. Last comes the `end` command.

Loops are one of the main ways that computers are made to do calculations that humans cannot. Any calculation that involves a repeated process is easily done by a loop.

Now let's do a program that does n steps (iterations) of Newton's method. We will need to input the function, its derivative, the initial guess, and the number of steps. The output will be the final value of x, i.e. x_n. If we are only interested in the final approximation, not the intermediate steps, which is usually the case in the real world, then we can use a single variable `x` in the program `mynewton.m` and change it at each step:

```
function x = mynewton(f,f1,x0,n)
    % Solves f(x) = 0 by doing n steps of Newton's method starting at x0.
    % Inputs: f -- the function
    %         f1 -- it's derivative
    %         x0 -- starting guess, a number
    %         n -- the number of steps to do
    % Output: x -- the approximate solution
    x = x0;                     % set x equal to the initial guess x0
    for i = 1:n                 % Do n times
        x = x - f(x)/f1(x)      % Newton's formula, prints x too
    end
end
```

In the command window set to print more digits via

```
≫ format long
```

and to not print blank lines via

```
≫ format compact
```

Then define a function: $f(x) = x^3 - 5$ i.e.

```
≫ f = @(x) x^3 - 5
```

and define $f1$ to be its derivative, i.e.

```
≫ f1 = @(x) 3*x^2
```

Then run `mynewton` on this function. By trial and error, what is the lowest value of `n` for which the program converges (stops changing). By simple algebra, the true root of this function is $\sqrt[3]{5}$. How close is the program's answer to the true value?

Convergence

Newton's method converges rapidly when $f'(x^*)$ is nonzero and finite, and x_0 is close enough to x^* that the linear approximation (3.2) is valid. Let us take a look at what can go wrong.

For $f(x) = x^{1/3}$ we have $x^* = 0$ but $f'(x^*) = \infty$. If you try

```
≫ f = @(x) x^(1/3)
≫ f1 = @(x) (1/3)*x^(-2/3)
≫ x = mynewton(f,f1,0.1,10)
```

then x explodes.

For $f(x) = x^2$ we have $x^* = 0$ but $f'(x^*) = 0$. If you try

```
≫ f = @(x) x^2
≫ f1 = @(x) 2*x
≫ x = mynewton(f,f1,1,10)
```

then x does converge to 0, but not that rapidly.

If x_0 is not close enough to x^* that the linear approximation (3.2) is valid, then the iteration (3.4) gives some x_1 that may or may not be any better than x_0. If we keep iterating, then either

- x_n will eventually get close to x^* and the method will then converge (rapidly), or
- the iterations will not approach x^*.

Exercises

3.1 Enter: `format long`. Use `mynewton.m` on the function $f(x) = x^5 - 7$, with $x_0 = 2$. By trial and error, what is the lowest value of `n` for which the program converges (stops changing). Compute the error, which is how close the program's answer is to the true value. Compute the residual, which is the program's answer plugged into f. (See the next section for discussion.) Are the error and residual zero?

3.2 For $f(x) = x^3 - 4$, perform 3 iterations of Newton's method with starting point $x_0 = 2$. (By hand, but use a calculator.) Calculate the solution ($x^* = 4^{1/3}$) on a calculator and find the errors and percentage errors of x_0, x_1, x_2 and x_3. Use enough digits so that you do not falsely conclude the error is zero. Put the results in a table.

3.3 Suppose a ball is dropped from a height of 2 meters onto a hard surface and the coefficient of restitution of the collision is .9 (see Wikipedia for an explanation). Write a well-commented **script** program to calculate the total distance the ball has traveled when it hits the surface for the `n`-th time. Enter: `format long`. By trial and error approximate how large `n` must be so that total distance stops changing. Turn in the program and a brief summary of the results. (This program does not use Newton's method. It should be modeled on `mysum.m`.)