Lecture 4
Controlling Error and Conditional Statements

Measuring error and the Residual

If we are trying to find a numerical solution of an equation \( f(x) = 0 \), then there are a few different ways we can measure the error of our approximation. The most direct way to measure the error would be as

\[
\{ \text{Error at step } n \} = e_n = x_n - x^*
\]

where \( x_n \) is the \( n \)-th approximation and \( x^* \) is the true value. However, we usually do not know the value of \( x^* \), or we wouldn’t be trying to approximate it. This makes it impossible to know the error directly, and so we must be more clever.

One possible strategy, that often works, is to run a program until the approximation \( x_n \) stops changing. The problem with this is that sometimes it doesn’t work. Just because the program stops changing does not necessarily mean that \( x_n \) is close to the true solution.

For Newton’s method we have the following principle: At each step the number of significant digits roughly doubles. While this is an important statement about the error (since it means Newton’s method converges really quickly), it is somewhat hard to use in a program.

Rather than measure how close \( x_n \) is to \( x^* \), in this and many other situations it is much more practical to measure how close the equation is to being satisfied, in other words, how close \( y_n = f(x_n) \) is to 0. We will use the quantity \( r_n = f(x_n) - 0 \), called the residual, in many different situations. Most of the time we only care about the size of \( r_n \), so we look at \( |r_n| = |f(x_n)| \).

The if ... end statement

If we have a certain tolerance for \( |r_n| = |f(x_n)| \), then we can incorporate that into our Newton method program using an if ... end statement:

```matlab
function x = mynewton(f,f1,x0,n,tol)
% Solves f(x) = 0 by doing n steps of Newton's method starting at x0.
% Inputs: f -- the function, input as an inline
% f1 -- it's derivative, input as an inline
% x0 -- starting guess, a number
% tol -- desired tolerance, prints a warning if |f(x)|>tol
% 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
end
r = abs(f(x))
if r > tol
    warning('The desired accuracy was not attained')
end
```

11
In this program if checks if \( \text{abs}(y) > \text{tol} \) is true or not. If it is true then it does everything between there and end. If not true, then it skips ahead to end.

In the command window define the function

\[
> f = \text{inline}(\text{'x^3-5'},'x')
\]

and its derivative

\[
> f1 = \text{inline}(\text{'3*x^2'},'x')
\]

Then use the program with \( n = 5, \text{tol} = .01, \) and \( x_0 = 2 \). Next, change \( \text{tol} \) to \( 10^{-10} \) and repeat.

**The loop: while ... end**

While the previous program will tell us if it worked or not, we still have to input \( n \), the number of steps to take. Even for a well-behaved problem, if we make \( n \) too small then the tolerance will not be attained and we will have to go back and increase it, or, if we make \( n \) too big, then the program will take more steps than necessary.

One way to control the number of steps taken is to iterate until the residual \( |r_n| = |f(x)| = |y| \) is small enough. In MATLAB this is easily accomplished with a **while ... end** loop.

```matlab
function x = mynewton(tol)
    x = x0; % set x equal to the initial guess x0
    y = f(x);
    while abs(y) > tol % Do until the tolerance is reached.
        x = x - y/f1(x) % Newton's formula
        y = f(x)
    end
```

The statement **while ... end** is a loop, similar to **for ... end**, but instead of going through the loop a fixed number of times it keeps going as long as the statement \( \text{abs}(y) > \text{tol} \) is true.

One obvious drawback of the program is that \( \text{abs}(y) \) might never get smaller than \( \text{tol} \). If this happens, the program would continue to run over and over until we stop it. Try this by setting the tolerance to a really small number:

\[
> \text{tol} = 10^{-100}
\]

then run the program again for \( f(x) = x^3 - 5 \).

You can use Ctrl-c to stop the program when it's stuck.

One way to avoid an infinite loop is add a counter variable, say \( i \) and a maximum number of iterations to the programs.

Using the **while** statement, this can be accomplished as:

```matlab
function x = mynewton(tol)
    x = x0; % set x equal to the initial guess x0.
    i=0; % set counter to zero
    y = f(x);
    while abs(y) > tol & i < 1000 % Do until the tolerance is reached or max iter.
        x = x - y/f1(x) % Newton's formula
        y = f(x)
        i = i+1; % increment counter
    end
```
Exercises

4.1 In Calculus we learn that a geometric series has an exact sum:

\[ \sum_{i=0}^{\infty} r^i = \frac{1}{1-r} \]

provided that \(|r| < 1\). For instance, if \(r = .5\) then the sum is exactly 2. Below is a script program that lacks one line as written. Put in the missing command and then use the program to verify the result above.

How many steps does it take? How close is the answer to 2? Change \(r = .5\) to \(r = .9\). How many steps does it take now? How accurate is the answer? Try this also for \(r = .99, .999, .9999\). Report your results. At what point is the program’s output different from the known exact answer?

```
% Computes a geometric series until it seems to converge
format long
r = .5;
Snew = 0; % start sum at 0
Sold = -1; % set Sold to trick while the first time
i = 0; % count iterations
while Snew > Sold % is the sum still changing?
    Sold = Snew; % save previous value to compare to
    Snew = Snew + r^i;
    i=i+1;
end
Snew % prints the final value.
i % prints the # of iterations.
```

4.2 Modify your program from exercise 3.2 to compute the total distance traveled by the ball while its bounces are at least 1 centimeter high. Use a `while` loop to decide when to stop summing; do not use a `for` loop or trial and error. Turn in your modified program and a brief summary of the results.