Introduction to Scientific Computing Lab 2

Conditionals and Looping

2017

1 Objectives

After completing these exercises you will be able to:

- Use conditionals to control user input
- Use loops to perform iterative tasks

2 Notes

Work individually

Create a directory for each week so you can come back to your codes in the future. Create files for each of the different exercises and name them in a logical manner, so `exercise1.c` for example.

When changing the source code, ensure you have **saved it before compiling** otherwise the compiler will only see the old file

When you have completed all exercises, ask a demonstrator to assess your work. They will test your code and ensure it is formatted well with good commenting, structure and variable names. This is a useful feedback mechanism, so listen to what the demonstrator has to say and their recommendations for improving your code.

In case of an error, read the compiler error. This will often tell you the line (or close to the line) where the error is occurring. Fix it, test and repeat for the errors you have. If you are getting nowhere then it can often be useful to copy the error into google, or use some keyword searches. If you are really stuck on one error then call over an assistant who will be able to point you in the right direction.

3 Finding Roots of Polynomials

Given a univariate function, $f(x)$, that has a continuous derivative $f'(x)$ at all values of x, the roots of the function are defined as being the values of x where

$$f(x) = 0 \tag{1}$$

The Newton-Raphson method is a simple method for finding roots of expressions. Given a current guess for the root, x_t, a better guess will be found at:

$$x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)} \tag{2}$$

Physically, the method equates the derivative of the function with the gradient of the straight line between the current guess and the next guess (you can derive the method in a couple of lines of algebra using this idea, and would help your understanding). The idea of the method is shown visually below, illustrating two steps of the method.

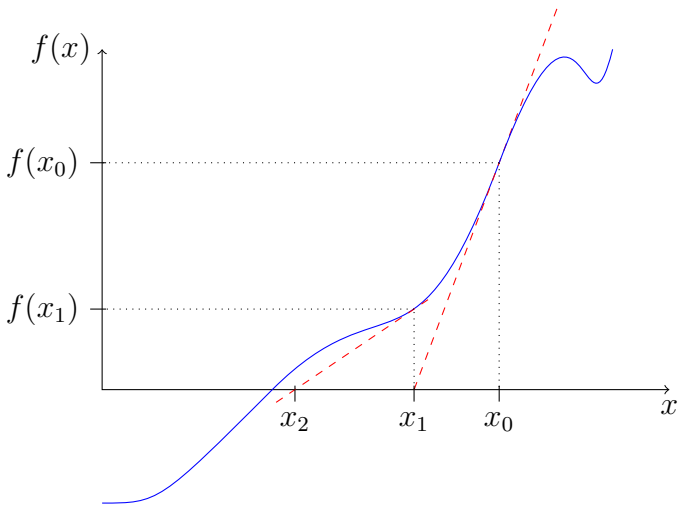

Figure 1: Illustration of Newton-Raphson method

This method can be used to estimate values of square-roots. For this exercise, we will estimate square roots of known numbers using the Newton-Raphson method. To estimate a square root, we must first cast a problem in the form of equation 1. Given a number, N, which we want to find the square root of, x, we know that

$$x^2 = N \tag{3}$$

therefore, we can write the function to be solved and its derivative as:

$$f(x) = x^2 - N \quad , \quad f'(x) = 2x \tag{4}$$

4 Exercises

Ensure you create a new directory for this week.

HINT: to test your code, use values of N=4, 9, 16, etc. where you know the answers.

1. Write a program that asks for the user to input the following data and store it:

   - N: the number we wish to find the square-root of
   - x_0: the initial guess for the algorithm (10 is a good value for testing)
   - t_{max}: the maximum number of iterations to be performed by the algorithm
   - ϵ: the error tolerance below which the algorithm will stop

   We are only dealing with real roots, so check whether N is positive, and if not then display a warning message to tell the user you are going to consider the absolute value of N, and use only the absolute value of N.

2. Extend exercise 1 to perform some Newton-Raphson iterations to estimate the square-root of N. You should decide whether to use a `while`, `do-while` or `for` loop, as appropriate (this exercise can be done with all three).

   - Ensure your program does not perform more iterations than t_{max}
   - If the error falls below the specified tolerance (i.e. if $f(x) \leq \epsilon$) then stop
   - Print out the history of the iterations: t, x, $f(x)$ and $f'(x)$

The output to your code could look something like:

```
t          x(t)          f(x)          f'(x)          x(t+1)
1     10.000000     96.000000     20.000000      5.200000
2      5.200000     23.040000     10.400000      2.984615
3      2.984615      4.907929      5.969231      2.162411
4      2.162411      0.676020      4.324822      2.006099
5      2.006099      0.024433      4.012198      2.000009
6      2.000009      0.000037      4.000019      2.000000
7      2.000000      0.000000      4.000000      2.000000
```

5 Next Week

Creating your own functions to make your code more modular.