Queens College, CUNY, Department of Computer Science

Numerical Methods

CSCI 361 / 761

Spring 2018

Instructor: Dr. Sateesh Mane

due Friday July 13, 2018 11.59 pm

2 Homework set 2

- **Experience has demonstrated that in many cases the source of difficulty is not the mathematics.**
- **The source of difficulty is the English (understanding the text).**
- **If you do not understand the words in the lectures or homework, THEN ASK.**
- **If you do not understand the concepts in the lectures or homework, THEN ASK.**
- **Send me an email, explain what you do not understand.**
- **Do not just keep quiet and then produce nonsense in exams.**

2.1 Functions

- We shall write code to implement the bisection and Newton–Raphson root finding algorithms.
- For bisection, we require only the function value.
- For Newton–Raphson, we also require the derivative.
- **Write the following forward declarations, which we shall use below.**

```
double func(double x);
void func(double x, double &f, double &fprime);
```

2.2 Bisection

- The function signature for our bisection function is

```
int root_bisection(double target, double tol_f, double tol_x, int max_iter,
                   double x_low, double x_high,
                   double & x, int & num_iter)
```

- The return type is "`int`" not `void`, because the algorithm might not converge. If the calculation succeeds, we return 0. It it fails, we return 1.
- The inputs are (i) `double target`, (ii) `double tol_f`, (iii) `double tol_x`, (iv) `int max_iter`, (v) `double x_low`, (vi) `double x_high`.
- The outputs are (vii) `double & x` (the root), (viii) `int & num_iter` (number of iterations).
- Obviously `target` is the target value. We wish to solve $f(x) = \text{target}$.
- We also need tolerance parameters.
  Remember that an iterative algorithm needs a cutoff parameter to stop iterating.
  We use `tol_f` for convergence along the y-axis and `tol_x` for convergence along the x-axis.
- As a safety check, we also input an upper limit `max_iter` on the number of iterations, in case the computations take too long.

The following pseudocode describes the steps. You can use it as the basis to write a working function.

1. Initialize `x = 0` and `num_iter = 0`.
2. Calculate a value `double y_low = func(x_low)`.
   Also calculate `double diff_y_low = y_low - target`.
3. If `|diff_y_low|` $\leq$ `tol_f`, *then we are done.*
   The value of `y_low` is already within the tolerance.
   Set `x = x_low` and "return 0" (= success) and exit.
4. Next calculate a value `double y_high = func(x_high)`.
   Also calculate `double diff_y_high = y_high - target`.
5. If `|diff_y_high|` $\leq$ `tol_f`, *then we are done.*
   The value of `y_high` is already within the tolerance.
   Set `x = x_high` and "return 0" (= success) and exit.
6. Next we must check if we have bracketed a root.
   In order to bracket a root, `y_low` and `y_high` must lie on opposite sides of `target`.
   This means `diff_y_low` and `diff_y_high` must have **opposite signs.**
7. Test if `(diff_y_low * diff_y_high > 0.0)`.
   If yes, ***then we have failed***. The inputs `x_low` and `x_high` do not bracket a root.
   Set `x = 0` and **"return 1" (= fail)** and exit.

8. If we have made it this far, then we know that we have bracketed a root.
   We know that there is a solution for x somewhere between `x_low` and `x_high`.

9. Hence we begin the bisection (iteration) loop.

```
for (num_iter = 1; num_iter < max_iter; ++num_iter) {
   // (to be filled in below)
}
```

10. In the loop, set `x = (x_low + x_high)/2.0` and calculate `double y = func(x)`.

11. Also calculate `double diff_y = y - target`.

12. If `|diff_y|` $\leq$ `tol_f`, *then we are done.*
    The value of `y` is within the tolerance.
    We have found a "good enough" value for x.
    Hence "return 0" (= success) and exit.

13. **Next test if `(diff_y * diff_y_low > 0.0)`.**
    If yes, it means `y` and `y_low` are on the same side of `target`.
    This means `x` and `x_low` are on the same side of the root.
    Update `x_low = x`.

14. Else obviously `y` and `y_high` are on the same side of `target`.
    Hence `x` and `x_high` are on the same side of the root.
    Update `x_high = x`.

15. *Don't be in rush to iterate!* **There is one more test!**

16. We have tested for convergence in y, now we must test for convergence in x.

17. If `|x_high - x_low|` $\leq$ `tol_x`, *then the algorithm has converged (up to the tolerance).*
    Note that the **tolerance is `tol_x`** in this test (convergence in x).
    Hence "return 0" (= success) and exit.

18. If we have come this far, continue with the iteration loop.

19. If we exit the iteration loop after `max_iter` steps and the calculation still has not converged, then set $x = 0$ and `num_iter = max_iter` and "return 1" (= fail) and exit.

20. We have reached the end of the function. By now either we have a "good enough" answer ("return 0" = success) or not ("return 1" = fail).

2.3 Testing of function

- You will have to write a main program to call and test your function.
- However, you also need an actual example of a function $f(x)$.
- One simple example is a parabola.

```
double func(double x)
{
  return x*x;
}
```

- For later use with Newton–Raphson, also write the following function.

```
void func(double x, double &f, double &fprime)
{
  f = x*x;
  fprime = 2.0*x;
}
```

- The equation is therefore $f(x) = \texttt{target}$. The solution is $x = \pm\sqrt{\texttt{target}}$.
- Try `target = 4.0`. If your function works correctly, it should output $x = 2.0$ or -2.0.
- Set `tol_f = tol_x = 1.0e-6` and `max_iter=100`.
- If `x_low=0.0` and `x_high=5.0`, the algorithm should converge to $x = 2.0$.
- If `x_low=-5.0` and `x_high=5.0`, the algorithm should return 1 and fail.
- You can devise other tests. Use your imagination.
- Try `x_low=-10.0` and `x_high=0.0`. Try `target =-1.0`.

2.4 Newton–Raphson

- Next let us write a function to implement the Newton–Raphson algorithm.
  This function is in fact shorter than the one for bisection.

- The function signature for our Newton–Raphson function is

```
int root_NR(double target, double tol_f, double tol_x, int max_iter, double x0,
            double & x, int & num_iter);
```

- The return type is "`int`" not `void`, because the algorithm might not converge.
  If the calculation succeeds, we return 0. It it fails, we return 1.

- The inputs are (i) `double target`, (ii) `double tol_f`, (iii) `double tol_x`, (iv) `int max_iter`, (v) `double x0` (initial iterate).

- The outputs are (vi) `double & x` (the root), (vii) `int & num_iter` (number of iterations).

- Obviously `target` is the target value. We wish to solve $f(x) = \text{target}$.

- We also need tolerance parameters.
  Remember that an iterative algorithm needs a cutoff parameter to stop iterating.
  We use `tol_f` for convergence along the y-axis and `tol_x` for convergence along the x-axis.

- As a safety check, we also input an upper limit `max_iter` on the number of iterations, in case the computations take too long.

The following pseudocode describes the steps. You can use it as the basis to write a working function.

1. Declare some useful variables at the start of the function.

```
const double tol_fprime = 1.0e-12;
double f = 0;
double fprime = 0;
```

2. The const parameter "`tol_fprime`" is to guard against division by zero.
   The value of `1.0e-12` is arbitrary.

3. We do not need any preliminary tests.
   Set `x = x0` (= initial iterate) and begin the main iteration loop.

```
x = x0;
for (num_iter = 1; num_iter < max_iter; ++num_iter) {
   // (to be filled in below)
}
```

4. We could begin with `num_iter = 0` but this is not important.

5. In the loop, call `func(x,f,fprime)`.

6. Also calculate `double diff_f = f - target`.

7. If `|diff_f|` $\leq$ `tol_f`, *then we are done.*
   The value of `f` is within the tolerance.
   We have found a "good enough" value for x.
   Hence "return 0" (= success) and exit.

8. Next test if `|f_prime|` $\leq$ `tol_fprime`.
   If yes, then this will cause a division by zero and ***the Newton–Raphson iteration fails.***
   Set `x = 0` and **"return 1" (= fail)** and exit.

9. Calculate `double delta_x = diff_f/fprime`.

10. Now test for convergence in x.

11. If `|delta_x|` $\leq$ `tol_x`, *then the algorithm has converged (up to the tolerance).*
    Note that the **tolerance is `tol_x`** in this test (convergence in x).
    Hence "return 0" (= success) and exit.

12. If the convergence tests have not passed, update the value of x (note the minus sign).

```
x -= delta_x;
```

13. Continue with the iteration loop.

14. If we exit the iteration loop after `max_iter` steps and the calculation still has not converged, then set $x = 0$ and `num_iter = max_iter` and "return 1" (= fail) and exit.

15. We have reached the end of the function. By now either we have a "good enough" answer ("return 0" = success) or not ("return 1" = fail).

16. Probably the best test is to use the same main program that you wrote to call the bisection algorithm and use it to also call the Newton-Raphson function. Give the same target and tolerance parameters to both functions. For bisection you have to input `x_low` and `x_high` whereas for Newton–Raphson you have to input a starting oterate `x0`. If they both converge, the output value of x should be the same in both cases (up to the tolerance). Also print out the number of iterations `num_iter` from both function calls. In general, Newton-Raphson should converge (much) more quickly.

2.5 Cumulative Normal Distribution

- A more interesting function is the **cumulative normal distribution**.
- The probability density function for the normal probability distribution is (with mean = 0 and variance = 1)

$$p(x) = \frac{e^{-x^2/2}}{\sqrt{2\pi}} \qquad (-\infty < x < \infty)\,. \tag{2.5.1}$$

- The **cumulative normal distribution** is defined via the integral

$$N(x) = \int_{-\infty}^{x} \frac{e^{-t^2/2}}{\sqrt{2\pi}}\,dt \qquad (-\infty < x < \infty)\,. \tag{2.5.2}$$

- The difficulty is, how do we compute $N(x)$ for arbitrary values of x?
  The above formula is an integral, which by itself is complicated to compute.
- Fortunately C++ (also Excel/Visual Basic) and other languages have a function $\mathrm{erf}(x)$.
  It is known as the "error function" (hence "erf") as is defined as

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,dt\,. \tag{2.5.3}$$

- Hence we obtain $N(x)$ via

$$N(x) = \frac{1 + \mathrm{erf}(x/\sqrt{2})}{2}\,. \tag{2.5.4}$$

- **The following functions will serve our needs.**

```
double cum_norm(double x)
{
  const double root = sqrt(0.5);
  return 0.5*(1.0 + erf(x*root));
}
double func(double x)
{
  return cum_norm(x);
}
void func(double x, double &f, double &fprime)
{
  const double pi = 4.0*atan2(1.0,1.0);
  f = cum_norm(x);
  fprime = exp(-0.5*x*x)/sqrt(2.0*pi);
}
```

- Use this and run some tests with both bisection and Newton–Raphson.
  Unlike a quadratic, it is not simple to solve the equation `N(x) = target`.
  By construction, $N(-\infty) = 0$ and $N(\infty) = 1$ and for finite values of x, then $0 < N(x) < 1$.
  Hence to obtain convergence, the target value must lie in the range `0 < target < 1`.

2.6 Multiple test functions

- **Write only one bisection function and one Newton–Raphson function.**
- Do not write `root_bisection_quadratic()` and `root_bisection_cum_norm()`, etc.
- Also do not write a new cpp file every time you write a new test function.
- Although this will work for a homework assignment, in a real software library function we cannot write a new function or new file for each new test function. What will happen if the client wants to use $f(x) = x^3$ and solve $x^3 = \text{target}$? Or some other function?
- A real software library root-finding function would take an extra input argument for a function pointer. I decided this is too complicated and too much programming burden on you. A lot of programming formalism, which has nothing to do with the subject matter of this course.
- *What to do?*
- Use `#ifdef` preprocessor directives. Here is a sample code of what I mean. It is not pretty but it works. I added a third selection "cubic" for $f(x) = x^3$ to illustrate the basic idea. Uncomment one of the `#define` statements to select a test function.
- Then `root_bisection()` etc. calls only `func()` and not a different function for each test case.
- It is true that you must recompile the program every time you change the choice of test function. It is not a pretty solution, but it works.

2.6.1 Preprocessor code

uncomment one of the `#define` choices below to select a test function

```
//#define func_quadratic
//#define func_cubic
#define func_cum_norm

#ifdef func_quadratic
double func(double x){
    return x*x;
};
void func(double x, double &f, double &fprime){
    f=x*x;
    fprime=2.0*x;
};
#endif  // func_quadratic

#ifdef func_cubic
double func(double x){
    return x*x*x;
};
void func(double x, double &f, double &fprime){
    f=x*x*x;
    fprime=3.0*x*x;
};
#endif  // func_cubic

#ifdef func_cum_norm
double cum_norm(double x)
{
    const double root = sqrt(0.5);
    return 0.5*(1.0 + erf(x*root));
}
double func(double x)
{
    return cum_norm(x);
}
void func(double x, double &f, double &fprime)
{
    const double pi = 4.0*atan2(1.0,1.0);
    f = cum_norm(x);
    fprime = exp(-0.5*x*x)/sqrt(2.0*pi);
}
#endif  // func_cum_norm
```

2.6.2 Main program pseucode

```
// header files etc

int main()
{
   // set inputs
   // call root_bisection(...)
   // call root_NR(...)
   // check if the results from bisection and NR match (up to tolerance)

   // CALL THE TEST FUNCTION USING X = ROOT (bisec and NR roots)
   // IS THE VALUE F(X_ROOT) CLOSE TO ZERO?

   // in other words, did the algorithm converge correctly?

  return 0;
}
```