Queens College, CUNY, Department of Computer Science

Numerical Methods

CSCI 361 / 761

Spring 2019

Instructor: Dr. Sateesh Mane

2 Homework set 2

- Please email your solution, as a file attachment, to `Sateesh.Mane@qc.cuny.edu`.
- Please submit one zip archive with all your files in it.
  1. The zip archive should have either of the names (CS361 or CS761):

     `StudentId_first_last_CS361_hw2.zip`

     `StudentId_first_last_CS761_hw2.zip`
  2. The archive should contain one "text file" named "hw2.[txt/docx/pdf]" and one cpp file per question named "Q1.cpp" and "Q2.cpp" etc.
  3. Note that not all homework assignments may require a text file.
  4. Note that not all questions may require a cpp file.

2.1 Functions

- The task is to implement root finding algorithms for nonlinear equations $f(x) = 0$.
- We shall write code to implement the bisection, Newton–Raphson and secant algorithms.
- For bisection and secant, we require only the function value $f(x)$.
- For Newton–Raphson, we also require the derivative $f'(x)$.
- ***But how shall we write functions to implement these algorithms?***
  1. We could write three standalone functions.
  2. That would serve the immediate needs of this homework assignment.
  3. But this course is about something more than writing individual functions.
- We shall code our functions into a simple version of a mathematical software library.

2.2 Math Library

- Let us set up a simple ‘numerical math library’ to perform calculations for this course.
- It is a simple version of real numerical mathematical software libraries.
- It gives you an idea of how to create such a library.
- Our immediate purpose is to write functions for root finding to solve nonlinear equations $f(x) = 0$.
- ***What do we need to do?***
  1. Begin with bisection, it is the simplest.
  2. We wish to write a function to implement the bisection algorithm for a function $f(x)$.
  3. *But the function $f(x)$ is supplied by the user.*
  4. A library routine cannot know the function $f(x)$ in advance.
  5. *How to deal with this?*
- We shall solve the problem by using an **abstract base class.**
- Our abstract base class will contain one or more virtual functions.
- Our library routines will be written using exclusively the abstract base class.
- Users will override our abstract base class, to supply their functions.
- This is an example of a real-life implementation of **polymorphism.**

2.3 Abstract base class

- Here is the code for our abstract base class `MathFunction`.
- Users will override it to supply their mathematical functions, hence its name.
    1. The class contains a virtual function $f(x)$ with return type `double`.
    2. The class contains a virtual function `ffp`.
       Users override it to supply the values of both $f(x)$ and $f'(x)$.
    3. For later use, there is also a virtual function $f(x, y)$ with return type `double`.
       We shall use this function later in this course.
    4. *We are setting up our library to be useful for more than just root finding.*
    5. *We shall add more functionality into our library later in this course.*
- All of the virtual functions are tagged `const` because we give a promise to the user that we will only call the functions to compute the value of $f(x)$, etc., *but we will never change the internal state of the user's object.*
- All of the functions are virtual but they are not pure virtual functions.
- Users do not need to override every virtual function to use our class and library.
- We make the class abstract by making the constructor protected.
- We also write a virtual destructor to support dynamic memory allocation in derived classes.
- In Java, you should be able to figure out how to write an interface to do the job.

```
class MathFunction {
public:
  virtual double f(double x) const { return 0; }
  virtual void ffp(double x, double &f, double &fprime) const {}
  virtual double fxy(double x, double y) const { return 0; }     // for later use

  // virtual destructor for base class
  virtual ~MathFunction() {}

protected:
  // protected constructor to disable creation of objects
  MathFunction() {}
};
```

2.4 Math library interface

- Here are the function declarations for our math software library.
  1. In a real math software library, this declaration would be placed in a header file.
  2. Only the header file would be given to users.
  3. The actual code for the function bodies would be placed in an encrypted file(s).
  4. It is secret proprietary code, for which vendors charge money to customers.
  5. However, we do not need to make things so complicated.
  6. You can write all the function code inline, it will be simpler.
- The library is named `MathLibraryCPP` to indicate it is written in C++.
  1. Similar to a 'final' class in Java, we make the constructor private, so it is not possible to write a derived class and override our functions.
  2. All the functions are static, so we never instantiate a `MathLibraryCPP` object.
  3. This is similar in concept to the `Math` class in Java, which is public and final.
  4. The return type of all the functions is `int`, because the calculations may fail.
     We return 0 for success and 1 if the calculation fails.
  5. The reference `&mf` is `const` to give a promise to the user that we will never change their input object: we only use `mf` to compute $f(x)$, etc.
- You should be able to figure out how to implement a Java version of the library
  `public final class MathLibraryJava`.

```
class MathLibraryCPP {
public:
  static int bisection(const MathFunction &mf, double target,
                       double tol_f, double tol_x, int max_iter,
                       double x_low, double x_high, double &x, int &num_iter);

  static int NewtonRaphson(const MathFunction &mf, double target,
                           double tol_f, double tol_x, int max_iter,
                           double x0, double &x, int &num_iter);

  static int secant(const MathFunction &mf, double target,
                    double tol_f, double tol_x, int max_iter,
                    double x0, double x1, double &x, int &num_iter);

  // we shall add more functions later

 private:
  // private constructor to disable creation of objects
  MathLibraryCPP() {}
};
```

2.5 Bisection

- The function signature for our bisection function is (you can write the code inline)

```
static int bisection(const MathFunction &mf, 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) `const MathFunction &mf` (`const` reference to abstract base class), (ii) `double target`, (iii) `double tol_f`, (iv) `double tol_x`, (v) `int max_iter`, (vi) `double x_low`, (vii) `double x_high`.
- The outputs are (viii) `double &x` (the root), (ix) `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 = mf.f(x_low)`.
   This is the polymorphism at work: users will override the function f.
   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 = mf.f(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`.
   The 'solution' may be a root or the location of a discontinuity.

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 = mf.f(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).

2.6 Testing of function

- You will have to write a main program to call and test your library.
- However, you also need an actual example of a function $f(x)$.
- One simple example is a parabola, or rather the square $f(x) = x^2$.
  1. Write a derived class and override `f`.
  2. For later use with Newton–Raphson, also override `ffp`.
  3. There is no need to override any other virtual functions.
  4. *This is the advantahe of our software design: the virtual functions in the abstract base class are NOT pure virtual, hence we do NOT need to override functions which are not relevant for our purposes.*

```
class MySquare : public MathFunction {
public:
  virtual double f(double x) const {
    return x*x;
  }
  virtual void ffp(double x, double &f, double &fprime) const {
    f = x*x;
    fprime = 2.0*x;
  }
  // no need to override anything else
};
```

- The equation is therefore $f(x) = x^2 = \texttt{target}$. 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.
- **Try `x_low=-10.0` and `x_high=0.0`. Try `target =-1.0`.**
- You can devise other tests, for example $f(x) = x^3$ or x^4. Use your imagination.

2.7 Newton–Raphson

- Next let us write a function to implement the Newton–Raphson algorithm.
  This code for this function is in fact shorter than that for bisection.
- The function signature for our Newton–Raphson function is

```
static int NewtonRaphson(const MathFunction &mf, 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) `const MathFunction &mf` (`const` reference to abstract base class), (ii) `double target`, (iii) `double tol_f`, (iv) `double tol_x`, (v) `int max_iter`, (vi) `double x0` (initial iterate).
- 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 $f(x) =$ 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 `mf.ffp(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 only one starting iterate `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.

2.8 Secant method

- The secant method is simply the Newton–Raphson algorithm but we replace the function derivative $f'(x)$ by a numerical approximation

$$f'(x_i) \simeq \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}\,. \tag{2.8.1}$$

- Hence the secant method calls only $f(x)$, like bisection.
- However, unlike bisection, we supply two initial iterates x_0 and x_1, not an initial bracket.
- The function signature for the secant function is very similar to Newton–Raphson:

```
static int secant(const MathFunction &mf, double target,
                  double tol_f, double tol_x, int max_iter,
                  double x0, double x1, double &x, int &num_iter);
```

- The inputs contain two initial iterates x_0 and x_1.
- **Everything else is almost the same as Newton–Raphson.**

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 (same as Newton–Raphson).

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

2. We need variables to hold data for the "previous iteration" x_{i-1} and $f(x_{i-1})$ in eq. (2.8.1).
   (a) At the start, the "previous iterate" is x_0, so declare and initialize a variable `x_prev`

   ```
   double x_prev = x0;
   ```

   (b) Then obviously we need a variable `f_prev` for $f(x_{i-1})$. At the start this is $f(x_0)$.

   ```
   double f_prev = func(x0);
   ```

3. The obvious validation test is to check if $f(x_0)$ is already good enough to meet the tolerance.
4. If `|f_prev - target| <= tol_f`, set $x = x_0$ and return 0 (= success) and exit.
5. Otherwise we must iterate.
6. The "initial iterate" is x_1, **so set `x = x1`** and begin the main iteration loop.

```
x = x1;
for (num_iter = 1; num_iter < max_iter; ++num_iter) {
   // (to be filled in below)
}
```

7. In the loop, *we require only* $f(x)$, so we call `mf.f`, same as for bisection.

   ```
   f = mf.f(x);
   ```

8. As before, calculate `double diff_f = f - target`.

9. 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.

10. **Secant method:**

    (a) Calculate the numerical derivative as follows.

    ```
    fprime = (f - f_prev)/(x - x_prev);
    ```

    (b) **This is the key difference between secant and Newton–Raphson.**

    (c) We require a "division by zero" check. Test if `|f_prime|` $\leq$ `tol_fprime`.

    (d) If yes, this will cause a division by zero and ***the secant method fails.***

    (e) Set `x = 0` and **"return 1" (= fail)** and exit.

11. Calculate `double delta_x = diff_f/fprime` (same as Newton–Raphson).

12. Now test for convergence in x (same as Newton–Raphson).

13. 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.

14. If the convergence tests have not passed, then we must do *two* things.

    (a) **Update the values of `x_prev` and `f_prev`. *This is important.***

    ```
    x_prev = x;
    f_prev = f;
    ```

    (b) *After the above,* we update the value of x (same as Newton–Raphson).

    ```
    x -= delta_x;
    ```

15. Continue with the iteration loop.

16. 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.

17. 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).

18. **Testing:** Use the same tests employed for bisection and Newton–Raphson. The answer should be the same in all cases (up to tolerance). Also print the number of iterations. The secant method usually requires a few more iterations than Newton–Raphson.

2.9 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 $=1$)

$$p(x) = \frac{e^{-x^2/2}}{\sqrt{2\pi}} \qquad (-\infty < x < \infty)\,. \tag{2.9.1}$$

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

$$N(x) = \int_{-\infty}^{x} \frac{e^{-t^2/2}}{\sqrt{2\pi}}\,dt \qquad (-\infty < x < \infty)\,. \tag{2.9.2}$$

- 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.
  Excel actually supports a function for the cumulative normal distribution.
- C++ (and Java) have a function erf(x), the "error function" (hence "erf") defined as

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,dt\,. \tag{2.9.3}$$

- Hence we obtain $N(x)$ via

$$N(x) = \big[\,1 + \mathrm{erf}(x/\sqrt{2})\,\big]/2\,. \tag{2.9.4}$$

- **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));
}

class MyCumNorm : public MathFunction {
public:
  virtual double f(double x) const {
    return cum_norm(x);
  }
  virtual void ffp(double x, double &f, double &fprime) const {
    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`.

2.10 Relative tolerances

- **This is FYI, nothing for you to code here.**
- In this course, we shall not employ relative tolerances.
- Up to now we have implemented only absolute tolerances for f and x.
- Suppose we also have relative tolerance parameters `rel_tol_f` and `rel_tol_x`.
  1. The relative tolerance test for f is
  $$\frac{|f - f_{\text{target}}|}{|f_{\text{target}}|} \le \texttt{rel\_tol\_f}\,. \tag{2.10.1}$$
  2. To avoid a divide by zero weakness, we reformulate the test as
  $$|f - f_{\text{target}}| \le \texttt{rel\_tol\_f} * |f_{\text{target}}|\,. \tag{2.10.2}$$
- For x there are different tests for bisection, Newton–Raphson and secant.
- **Bisection**
  1. The relative tolerance test for x is
  $$\frac{|x_{\text{high}} - x_{\text{low}}|}{|x|} \le \texttt{rel\_tol\_x}\,. \tag{2.10.3}$$
  2. To avoid a divide by zero weakness, we reformulate the test as
  $$|x_{\text{high}} - x_{\text{low}}| \le \texttt{rel\_tol\_x} * |x|\,. \tag{2.10.4}$$
- **Newton–Raphson and secant**
  1. The relative tolerance test for x is
  $$\frac{|\delta x|}{|x|} \le \texttt{rel\_tol\_x}\,. \tag{2.10.5}$$
  2. To avoid a divide by zero weakness, we reformulate the test as
  $$|\delta x| \le \texttt{rel\_tol\_x} * |x|\,. \tag{2.10.6}$$

2.11 Absolute & relative tolerances

- **This is FYI, nothing for you to code here.**
- In this course, we shall not employ relative tolerances.
- The overall set of tests is:

$$
\begin{aligned}
|f - f_{\text{target}}| &\le \texttt{tol\_f}\,, && \text{(2.11.7a)}\\
|f - f_{\text{target}}| &\le \texttt{rel\_tol\_f} * |f_{\text{target}}|\,, && \text{(2.11.7b)}\\
|x_{\text{high}} - x_{\text{low}}| &\le \texttt{tol\_x}\,, && \text{(2.11.7c)}\\
|x_{\text{high}} - x_{\text{low}}| &\le \texttt{rel\_tol\_x} * |x|\,, && \text{(2.11.7d)}\\
|\delta x| &\le \texttt{tol\_x}\,, && \text{(2.11.7e)}\\
|\delta x| &\le \texttt{rel\_tol\_x} * |x|\,. && \text{(2.11.7f)}
\end{aligned}
$$

- If any of the tests pass, we say "converged" and return with a success status.
- This is an "either/or" implementation: if either the absolute or relative tolerance tests pass, we declare success.
- We can customize our tolerance tests.
  1. If we set the relative tolerances to zero, they will never pass (except by accident), so our convergence tests will be based on the absolute tolerance.
  2. If we set the absolute tolerances to zero, they will never pass (except by accident), so our convergence tests will be based on the relative tolerance.
  3. We can employ absolute tolerance for x and relative tolerance for f, etc.
- **What if we want *both* the absolute and relative conditions to be met?**
  1. We would require a logical AND for the tests:

     ```
     if ((absolute tolerance) && (relative tolerance)) then (success)
     ```
  2. This is a stricter condition and a different way of doing things.