PH20105, problem sheet on applications, 2016

0 Interactive class tasks

Task 1*: Plotting data*

The C programming language is not ideally suited for visualization and quick plotting of results. This aspect is typically covered better by dedicated plotting software like GNUPLOT or by a language like PYTHON that can be used for quick scripting. Before delving into numerical methods, let's ensure that we have some means of plotting functions. First, we will write a C program that writes the outcome of the function

$$F(x) = \sin(x) - 2\log(x+1) + \frac{x^2}{5}, \tag{1}$$

to a text file (let's say "data.txt"). We will write 100 entries of type `double`, `double`, with the left column containing x and the right column $F(x)$. The entries should be evenly spaced between 0 and 5. The program code is as follows, your task is to not just copy the program but to make sure that you understand every line:

```
// pre-compiler inclusion commands
#include <stdio.h> // include access to the printf statement and file I/O
#include <math.h> // for sin, log command. Compile with the -lm option

double F(double x)
{
  // return the requested function
  return sin(x) - 2. * log(x + 1.) + x * x / 5.;
}

void write_file(char *filename, double x0, double x1)
{
  // This function prints 101 values of the function F to file. The function
  // evaluations lie between x0 and x1, the filename is the argument filename

  int i; // define an integer i to use as loop counter
  double x; // define x, which we will use to denote the current position
    // within the interval.
  FILE *p_file; // a pointer to a file.

  // open the file for writing
  p_file = fopen(filename, "w");

  // perform a loop using the loop counter
  for (i = 0; i <= 100; i++)
  {
```

```
    // set x to current position within the interval
    x = x0 + i * (x1 - x0) / 100;
      // step size Delta x = (x1 - x0) / 100, for 100 steps

    fprintf(p_file, "%e, %e\n", x, F(x)); // print in scientific notation two
      // values, the first the value of x, the second the return value of
      // the function that can be found in memory starting at the location
      // pointed at by p_func
  }

  // don't forget to close the file afterwards
  fclose(p_file);
}

int main()
{
  // call the function to write the file
  write_file("data.txt", 0., 5.);

  return 0; // return 'all is well' when exiting program
}
```

After having written the text file with data, please try one of the following methods to plot the outcome.

For Python:

```
import matplotlib.pyplot as plt
import numpy as np

data = np.loadtxt("data.txt", delimiter = ', ')

x = np.array(data[:,0])
F = np.array(data[:,1])

plt.plot(x, F)
plt.draw()
plt.show()
```

For Gnuplot:

- Type `gnuplot` on your command line, assuming you're in the same directory as your data file.

- Type `plot "data.txt" u 1:2 w l`
- Type `exit` to exit gnuplot

For maple, in case the above does not work.

- Launch Maple. If there's a desktop short-cut on the Windows machine you can use that. You can also launch it via the terminal by typing `xmaple &`

Task 2*: Random Numbers*

We will not write our own random number generator, and we are going to avoid the question whether the current random number generator, provided by whatever C compiler version is available on the network, is sufficient. Instead we will import an external random number generator from a freely available package of *libraries* for use with C (but not part of the base package): *the GNU Scientific Library or (GSL)*. Please copy and compile the following code (as usual, no need to copy comments):

```
#include <stdio.h>
#include <gsl/gsl_rng.h> // the Random Number Generator GSL sub-library

int main (void)
{
  // Demonstration of GSL random numbers. There are different random
  // number generators in GSL. These are enumerated in global
  // variables declared in the GSL libraries. Since these variables
  // do not change over the course of the program (they're labels,
  // after all), these global variables get the additional indication
  // 'const' in front of them.

  // declare pointer to a constant variable of specific type
  const gsl_rng_type * T;

  // a pointer to our current random number generator for use in the
  // program
  gsl_rng * r;

  int i, n = 10; // declare loop counter i, will count for n steps

  gsl_rng_env_setup(); // set up environment for random number generation

  // set our pointer to a random number generator type to point
  // at the label associated with the default random number generator
  T = gsl_rng_default;
```

```
  r = gsl_rng_alloc (T); // initialize our current random number generator

  gsl_rng_set(r, 1); // set a random number sequence 'seed'

  // print n random numbers evenly spaced between 0 and 1, lower bound
  // inclusive, upper bound exclusive
  for (i = 0; i < n; i++)
    {
      double u = gsl_rng_uniform (r);
      printf ("%.5f\n", u);
    }

  // free memory allocated for our current random number generator
  gsl_rng_free (r);

  return 0; // return zero, i.e. "all is well"
}
```

Some of the program code might well appear obscure to you, which is unfortunate but inevitable when working with packages such as GSL. Please compile and run this program. In order to properly link the additionally required libraries, please add `-lgsl -lgslcblas -lm` to your compile command.

Although we will ignore this fact during the remainder of the session, by introducing GSL we effectively short-circuited our lecture. It provides far more than just random number routines, and in fact covers a comprehensive suite of computational analysis tools that can be used in professional research.

Task 3*: Bisection method*

Let's try to find a root for the function

$$F(x) = \log(x) - \frac{3}{2} + \frac{1}{2}\sin(\frac{x}{2}) \tag{2}$$

between e.g. 1 and 10. First, plot this function using one of the methods from task 1, so that we have some reason to believe the function is indeed well-behaved in this interval. Now copy the code below and run, in order to see a root-finder in action:

```
#include <stdio.h>
#include <math.h>

double F(double x)
{
  return log(x) - 3./2. + 1./2. * sin(x / 2.);
```

```
}

double find_zero(double x0, double x1)
{
  double x_mid = 0.5 * (x1 - x0) + x0;
  int i;

  double f0, f_mid, f1;

  f0 = F(x0);
  f1 = F(x1);

  for (i = 0; i < 100; i++)
  {

    f_mid = F(x_mid);

    if (f0 * f_mid < 0)
    {
      f1 = f_mid;
      x1 = x_mid;
    }
    else
    {
      f0 = f_mid;
      x0 = x_mid;
    }

    x_mid = 0.5 * (x1 - x0) + x0;
  }

  return x_mid;
}

int main()
{
  printf("find zero outcome: %e\n", find_zero(1., 10.));
  return 0;

}
```

How many function evaluations of $f(x)$ are there per iteration? Why is it generally a good idea to try to minimize the number of function evaluations?

This program will run for 100 iterations regardless of how close we are to zero.

We can improve upon this by adding an additional check for $|f(x)| < \varepsilon$. This can be implemented using an additional `return` statement in the right place that gets executed if the check is passed. You can square the equation to work around the absolute value requirement. Please add this improvement, setting an $\varepsilon = 10^{-4}$. How many iterations were needed (answer might require adding a print statement to the code)?

Task 4*: Differentiation*

From the set of equations

$$\begin{aligned} F(x-h)-F(x) &= (-h)F' + (h^2/2)F'' + (-h^3/6)F''' + O(h^4), \\ F(x+h)-F(x) &= (h)F' + (h^2/2)F'' + (h^3/6)F''' + O(h^4), \\ F(x+2h)-F(x) &= (2h)F' + (2h^2)F'' + (4h^3/3)F''' + O(h^4), \end{aligned}$$

derive an equation for $F'(x)$. If you know how to derive it, but did not manage after a valiant attempt due to repeated little mistakes, you can find the answer at the end of this problem sheet.

Add a function `double dFdx(double x, double h)` to your source code from task 3. This function should implement your derived equation for $F'(x)$, and allow the user to provide a size h. print the derivative as $x = 2$, using a small value for h. It should be around 0.635.

Task 5*: Integration*

Let us expand our source code from task 3 even further, now with an integration routine as well. Write a function `double int_F(double x_0, double x_1, int N)` that implements the extended Simpson's rule

$$\begin{aligned} \int_{x_1}^{x_N} f(x)dx &= h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{2}{3}f_3 + \ldots + \frac{2}{3}f_{N-2} + \frac{4}{3}f_{N-1} + \frac{1}{3}f_N\right] \\ &\quad + O\left(\frac{1}{N^4}\right) \end{aligned} \tag{3}$$

Your function is basically a `for` loop, but keep in mind that the outer ends should be treated different from the rest (they have $1/3$, instead of the alternating $2/3$, $4/3$).

Task 6*: Remember pointers?*

modify your routines to work with pointers to functions, instead of having function calls to $F(x)$ hardwired. You will also need to add an extra *argument* to your root finding, differentiation and integration functions: a pointer to the function that you want these to act on. Test your code by declaring a pointer-to-function `p_F` in your main routine, setting that to point to the function `F`, and providing that as an argument to function calls for each of the three (root finding, differentiation, integration)

1 Problems

Question 1a: *Newton-Rhapson approach to root finding*
Instead of the bisection approach, please write a Newton-Rhapson root finder to find a zero within an interval x_0, x_1. Apply it to the same function as before in task 3, but be careful to pick a smaller interval that does not include the extremum of the function within the original interval. Plot the function again to refresh your memory of its shape, if you need to.

In principle a full description of the Newton-Rhapson approach can be gathered from the lecture slides (i.e. it's shown graphically in the figure). But feel free to use *google* if you need additional information on the Newton-Rhapson algorithm. Just make sure that you understand yourself any code that you enter. Like we did with the bisection approach, just start with taking 20 steps.

Question 1b: *Checking for convergence*
Improve your algorithm in such a manner that it not just stops at 20 steps, but can also stop earlier if a desired accuracy is reached. For example, when the difference between two estimates for x becomes smaller than 0.0001.

Question 2a: *Ordinary Differential Equations*
So far we implemented root finding, differentiation and integration, but not yet a solver for ODE's. Write code to implement the fourth-order Runge-Kutta algorithm

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \\ k_3 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2) \\ k_4 &= hf(x_n + h, y_n + k_3) \\ y_{n+1} &= y_n + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4 + O(h^5) \end{aligned}$$

such that the user can provide a number of steps N, which then implies a stepsize h (given a user-provided range x_0, x_1). Use it to integrate $F(x)$, that we previously integrated via different means.

Question 2b: *Adaptive stepsize*
Take a stab at providing an adaptive stepsize extension to your ODE solver. Do not worry too much about efficiency at first (i.e. avoiding repeated evaluations of a function at the same point). Use an absolute accuracy criterion to judge whether the difference between taking one big step, and two small steps is sufficiently small that it is not necessary to further decrease the stepsize.

Answer to task 4

The equation you're looking for is

$$F' = \frac{-2F(x-h) - 3F(x) + 6F(x+h) - F(x+2h)}{6h} + O\left(h^4\right). \tag{4}$$

HJvE, Dec-2016, March 2019