Queen Mary
University of London

MTH6150 / MTH6150P: Numerical Computing with C and C++

Examiners: M. Agathos, I. Mainou

This is the Assignment for Coursework 2, counting for 80% of the final grade. The submission deadline is **Friday 17/05/2024 @ 17:00 BST**. Delayed submissions will not be accepted. A late submission will be given 0 marks, so please make sure you give yourself plenty of time ahead of the deadline, to submit your files to the system.
For this Coursework you will need to submit via QMplus a number of source code files with the indicated names and a .pdf document with your answers to the questions that require explanation. The source code files should have a few short and concise comments describing the non-trivial parts of the code. The text in the .pdf should convey your understanding of the problem and its solution, and should include plots where necessary.

In Coursework 2 we will study a set of problems and applications of Numerical Computing that we are familiar with in our everyday lives. We will use this problem set to test our understanding of and skills in

- programming in C++;
- object oriented programming;
- fundamental concepts of Numerical Analysis;
- root finders and their C++ implementation;
- numerical interpolation with C++;
- numerical differentiation with C++;
- solving ordinary differential equations on a computer with C++.

A running tracker app in C++

You like to exercise outdoors in your spare time, whether that's running, cycling or just going for a long stroll. Just like with your University courses, being able to monitor your runs and to get immediate feedback is an important factor in improving your performance. This is why you decided to use a GPS-enabled tracker app on your smartphone. You have set up your profile, that the app uses to keep a record of your activity as well as some of your personal data. Alas, the app is glitchy and badly designed. With some fresh ideas in mind, you are starting to wonder what you would need to write your own running tracker and maybe some day make a business out of it.

Question 1 [20 marks]. First some work on defining the central classes of the project. Create the first couple of files of a C++ library for a running tracker app, with

 Continue to next page

the following features:

(a) A `Run` class, that can keep some basic data for a given run. Implement this class in a file named `run.cpp`. It should have the following `public` member variables:

- `double t_start`: the starting time of the run (in Unix timestamp format[1], in seconds);
- `double t_end`: the end of the run (in Unix timestamp format, in seconds);
- `int duration`: the duration of the run in seconds (excluding pauses);
- `double distance`: the total distance covered in meters;
- `valarray<double> t_data`: an array with the equidistant ($dt = 1$ sec) timestamp data t_i, starting at `t_data[0] = t_start`;
- `valarray<double> lat_data`: an array of the same length as `t_data`, that holds a time series of data for the latitude $\texttt{lat}_i$;
- `valarray<double> lon_data`: an array of the same length as `t_data`, that holds a time series of data for the longitude $\texttt{lon}_i$;

It should also have the following member functions, which you should define:

- a constructor that creates a new object of the `Run` class, with a `double t_s` as its only argument that it uses to initialize the starting time, and immediately calls another member function `startRun()`
- three `public` functions, `void startRun()`, `void pauseRun()` and `void endRun()` which you can leave empty (assume that they are already written and will take care of filling in the data);
- a `public` function `void printRunInfo()` that prints out a structured message summarising the run;
- two `public` functions `double get_avg_pace()` and `double get_fastest_pace()`, that will return the average pace and the fastest pace of a run in units of min/km. Leave these empty; their implementation will be the main task of Question 2.c.

[13]

(b) In a new file named `runner.cpp`, define the `Runner` class, whose purpose is to hold the profile data for a user and maintain a record of their activities. The `Runner` class should have the following members

- `private` variables for: `username` (string), `age` (integer, in years), `weight` (integer, in kg), `height` (integer, in cm), `runList` (list container of `Run` objects [2])

[1] Number of seconds since Jan 01 1970 (UTC)

[2] You will need to include the header `<list>` from the standard library.

- `public` functions `getX()` and `setX()` for X in {`Username`, `Age`, `Weight`, `Height`, `RunList` }. Getter functions should take no arguments and should return a value of the correct type, while setter functions should take an argument of the correct type and have `void` as return type.
- a `public` function `newRun()` that creates a new `Run` object and adds it to the front of the `runList` container.

[7]

Question 2 [30 marks]. Copy the C++ source file `q2.cpp`. We want to add more features to our code, and will use this source file to develop new functions, which we will test on the datafile `test_run_coords.dat` .

The tracker app uses GPS to measure the position (`lon`, `lat`) of the runner at regular time intervals of $1s$, taking a total of N measurements from the start t_0 to the end t_{N-1} of the run. We will always use the Unix timestamp standard as the format of our data (to double precision). For instance, 00:00 1 Jan 2024 had the timestamp 1704067200.0. You can use the online tool https://www.epochconverter.com if you want to convert between timestamps and human-readable time & date format, although this is not necessary for this problem.

The function `read_gps_data()` reads the data from the file and stores all three columns in a `valarray`, while in the `main()` we copy the first column into `t_varr`, the second into `lat_varr` and the third into `lon_varr`. (These play the role of `t_data`, `lon_data` and `lat_data` that we saw in Q1, respectively.)

(a) To draw a smooth path on a map, we need to be able to evaluate (`lon`(t), `lat`(t)) for any $t \in [t_0, t_{N-1}]$, not just on the N steps of our time grid t_i. Do this by independently interpolating the two functions `lon`(t) and `lat`(t). We will use Lagrange interpolation with piecewise quadratic polynomials: using the source code in `interp.cpp` and the header file `interp.hpp` provided, write a function in `q2.cpp` with signature

```
double interp_coords(valarray<double>& t_varr,
                     valarray<double>& coord_varr, double t)
```

that takes as input two data arrays (one for the time grid, one for lattitude or longitude), and the time t at which we want to evaluate the interpolant. The function should return the value of the interpolant at t. Use this interpolation to evaluate the coordinate data on a dense time grid with $dt = 0.1$ s and print the data series to a file with four columns (i, t_i, `lat`$_i$, `lon`$_i$). Use this data to plot the path (`lon`$_i$ vs. `lat`$_i$) in 2D. A simple plot with `gnuplot` or an alternative of your choice will suffice. [7]

(b) Implement a function with the following signature

```
valarray<double> coords_to_distances(valarray<double>& lat,
                                     valarray<double>& lon)
```

that takes two arrays of length $N+1$ and returns an array `dl` of length N. The function should estimate the length $\texttt{dl}_i$ of each segment and should fill in the values of the array `dl` with those values. Do not use a loop! Hint: first create the `valarray` variables for `dlat` and `dlon`, using `slice()`.

Note that (`lat`, `lon`) are curvilinear coordinates (in [degree]), which means that the physical length of a displacement by `dlon` depends on where you are. In particular, the length of a small displacement is calculated as

$$\texttt{dl} = R_\oplus \frac{2\pi}{360} \sqrt{\texttt{dlat}^2 + \cos^2(\texttt{lat})\,\texttt{dlon}^2}\,, \tag{1}$$

where $R_\oplus = 6371000$ m is the (average) radius of the Earth. [3] Assign this value to a constant variable named `earthRadius`. The function should check that $N \geq 1$ and that the lengths of the two input arrays match. It should print an error message and exit (call `exit(-1);`) if this is not the case. **[8]**

(c) Given the data in the array `dl`, use first-order centered finite differences to estimate the speed for each segment. Write a function `valarray<double> dl_to_speed(valarray<double>& dl)` that takes the length displacement data and returns a `valarray` with the speed values. Assume that the data is sampled every 1s and is uninterrupted (no pauses). Use `valarray` variables and `dl_to_speed()` to implement the functions `get_avg_pace()` and `get_fastest_pace()` that were described in Q1.a. **[6]**

(d) If the error of each GPS location measurement is 0.5 m, how does this affect the estimates of instantaneous speed, average pace, fastest pace and total length covered? Would it make sense to decrease dt in order to achieve better accuracy? What would you do to make these estimates more reliable? **[4]**

(e) Your dog is the best running mate, but he occasionally gets distracted or needs to follow the call of nature; we don't want these disruptions to contaminate our data! Describe how you would code up a pause-detection function that pauses the run whenever it senses that the user has stopped moving. **[5]**

[3] Note that the argument of the `cos()` function needs to be transformed to [rad].

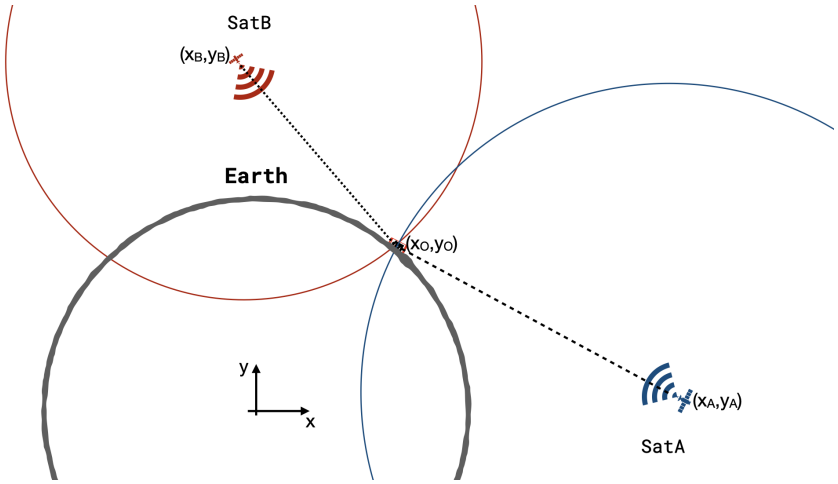
 Continue to next page

GPS geolocation in 2D

Knowing our exact location is considered a given fact in today's world, thanks to the global positioning system (GPS) technology, but there is much tech and computing hidden behind the scenes. GPS is a system that employs satellites orbiting around Earth to determine the position of any point on the planet, with a given accuracy. Each satellite continuously transmits (broadcasts) its position and a timestamp, with a high accuracy, so that an observer that receives multiple of those signals from different satellites can reconstruct their position accurately, by computing the flight times of those signals and triangulating them as shown in the figure below.

Here, we address the problem of computing our mobile device's coordinates, based on the data it receives from GPS satellites orbiting the Earth. To make the problem manageable, we make several simplifying assumptions. First, we study the problem in a 2D setup, constraining our motion and the motion of satellites on a plane, e.g. the one defined by the Greenwich meridian. We take the Earth's surface to be a perfect sphere (perfect circle in 2D) and we place the origin $(0, 0)$ of our coordinate system (x, y) at its centre (so that the equator is at $(R_\oplus, 0.0)$ and the North pole at $(0.0, R_\oplus)$), where $R_\oplus$ is the Earth's radius, as in Q2.

We receive GPS data from two satellites, SatA at (x_A, y_A) and SatB at (x_B, y_B). Our position is at (x_O, y_O) and we will solve the problem of computing our coordinates based on triangulation using GPS data, with the help of a 2D root-finder.



Question 3 [30 marks]. Root-finding our way to the finish line

The principle of computing our coordinates (x_O, y_O) using GPS, as illustrated in the figure above, is the following.

Every 1 second, SatA broadcasts a signal with its exact position (here the two coordinates (x_A, y_A)) and the timestamp. Our device receives the signal with some time delay δt_A: the time it takes for the signal to travel from SatA to our location at the speed of light, that is $\delta t_A = l_{OA}/c$. The same happens for SatB. Let us assume that the GPS receiver in our mobile device has a built-in perfect clock itself, so we can immediately translate the measured times of arrival $\bar{t}_I = t + \delta t_I$ $(I = A, B)$, for a given

 Continue to next page

timestamp t, to a pair of distances:

$$\begin{aligned} l_{OA} &= \delta t_A c \\ l_{OB} &= \delta t_B c\,, \end{aligned}$$

where c is the speed of light. Knowing the distance to SatA places us on a circle of radius l_{OA} around the known location of SatA. Likewise, knowing the distance to SatB places us on a circle of radius l_{OB} around the known location of SatB. We only need to solve the following system of two equations, for the two unknowns (x_O, y_O):

$$\begin{aligned} (x_O - x_A)^2 + (y_O - y_A)^2 &= (t_A c)^2 \\ (x_O - x_B)^2 + (y_O - y_B)^2 &= (t_B c)^2 \end{aligned}$$

Let's formulate the problem in a more familiar form, using the vector notation

$$\vec{x} = \begin{bmatrix} x \\ y \end{bmatrix} \quad , \quad \vec{x}_O = \begin{bmatrix} x_O \\ y_O \end{bmatrix} \quad , \quad \vec{x}_A = \begin{bmatrix} x_A \\ y_A \end{bmatrix} \quad , \quad \vec{x}_B = \begin{bmatrix} x_B \\ y_B \end{bmatrix} \quad ,$$

and also rewrite the equations we want to solve, in the form of a function with two components

$$\vec{f}(\vec{x}) = \begin{bmatrix} f_1(\vec{x}) \\ f_2(\vec{x}) \end{bmatrix} = \begin{bmatrix} (x - x_A)^2 + (y - y_A)^2 - (t_A c)^2 \\ (x - x_B)^2 + (y - y_B)^2 - (t_B c)^2 \end{bmatrix} = \vec{0} \tag{2}$$

The Jacobian matrix has a very simple analytical form:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{bmatrix} = 2 \begin{bmatrix} x - x_A & y - y_A \\ x - x_B & y - y_B \end{bmatrix} \tag{3}$$

(a) Create a new C++ source file named `q3.cpp` with a `main()` function. Download the header file `q3-4.hpp` and examine its contents. This header file contains the definitions of constants and declarations of functions that you will need to implement. In `q3.cpp`, write a `void` function `f()` that implements the function in Eq. (2), with the signature given in the header file. The function `f()` takes two arguments: the first one is a reference to an array, whose components the function should fill in with the evaluations of $\vec{f}(\vec{x})$; the second argument is the input array with the coordinates of $\vec{x}$. Write another `void` function `f_Jac()` that implements the 2x2 Jacobian matrix of Eq. (3), again using the indicated signature. **[7]**

(b) Download the source file `newton-raphson.cpp` and modify it to solve a 2-dimensional system of equations, by implementing the 2D version of the Newton-Raphson method. Recall that for systems of equations, the role of the derivative will be played by the Jacobian matrix $\mathbf{J}$. You will need to implement a function `mat_inv_2x2()` that calculates the inverse $\mathbf{J}^{-1}$ of a 2x2 matrix $\mathbf{J}$ given as argument. **[8]**

(c) How many solutions are there to the system of equations given by Eq. (2)? Give a geometric interpretation based on the figure. If there are more than one, describe how you can choose the initial point $\vec{x}_0$ (without having to do any calculations), so that you end up at the correct root that gives your device's location? **[3]**

 Continue to next page

(d) The problem is ill-conditioned when the Jacobian matrix becomes singular. If this happens during the itarative process, the program will crash. Together with the algebraic expression, give a geometric description of whether this can occur in our problem described by Eq. (2) and when? **[3]**

(e) In the `main()` function, declare a variable `double x0[2]` and initialize it to the coordinates of a starting point of your choosing. Declare another variable `double root[2]` which will be filled in by the `newton_raphson()` root-finder. Call the root-finder and print out the result (either in meters or in Earth radii units). Compute and print the components of $\vec{f}$ at the root, to check that they are indeed small. **[4]**

(f) Once all functions are implemented, compile the two source files and run the program from command-line, setting a relative error tolerance of 10^{-6}; the algorithm should converge within less than 10 iterations. What are the coordinates of the root and how many iterations did it take to reach the desired accuracy? **[2]**

(g) Accuracy: if the measurement error for the time of arrival is 10^{-5} s, estimate the error with which we calculate the position, in meters. **[3]**

Question 4 [20 marks]. A satellite in orbit

In the previous question, we treated each satellite's location as fixed. In reality of course, with the exception of satellites in geo-stationary orbits, each satellite will move with respect to the receiving device on Earth. We remain in our simplified 2D problem and we want to evolve the orbital motion of a satellite in a polar orbit (i.e. going around a meridian with fixed longitude).

To good approximation, the orbital dynamics are governed by the Earth's gravity

$$\vec{a} = \vec{x}'' = \frac{\vec{F}_G}{M} = -\frac{GM_\oplus}{r^2}\hat{x} \quad , \quad r = |\vec{x}|$$
$$\implies \quad \vec{x}''(t) = -\frac{GM_\oplus}{|\vec{x}(t)|^3}\vec{x}(t) \tag{4}$$

where again we placed the origin of the coordinate system at the centre of the Earth. This is a second order system of ODEs, which we can reformulate as

$$\vec{x}\,'(t) = \vec{v}(t) \tag{5}$$
$$\vec{v}\,'(t) = -\frac{GM_\oplus}{|\vec{x}(t)|^3}\vec{x}(t) \tag{6}$$

The system of equations governing the orbital dynamics is thus given by these four first order ODEs

$$\begin{bmatrix} x'(t) \\ y'(t) \\ v_x'(t) \\ v_y'(t) \end{bmatrix} = \begin{bmatrix} v_x(t) \\ v_y(t) \\ -\frac{GM_\oplus}{(x(t)^2+y(t)^2)^{3/2}}x(t) \\ -\frac{GM_\oplus}{(x(t)^2+y(t)^2)^{3/2}}y(t) \end{bmatrix} \tag{7}$$

(a) Create a new file `q4.cpp` with a `main()` function. Copy the source files

(i) `euler.cpp` implementing Euler's method and

(ii) `rk4.cpp` implementing the Runge-Kutta method RK4

to your current directory, to solve this system of ODEs. First implement the vector of functions on the right hand side (RHS) of Eq. (7). The initial value problem (IVP) is complete once we provide the inital data for the satellite's position $\vec{x}_0 = (x(t=0), y(t=0))$ and velocity $\vec{v}_0 = (v_x(t=0), v_y(t=0))$. Set these to the following values (in units of m and m/s respectively):

$$x(t=0) = 0.0\,, \quad y(t=0) = 26378100.0\,, \quad v_x(t=0) = 3887.3\,, \quad v_y(t=0) = 0.0\,. \tag{8}$$

In the function `main()`, set up the initial conditions and call the ODE solver. Finally, add print statements to display the results. [**5**]

(b) Compile the code and run it with Euler's method, to evolve the orbit for 10000 seconds, using $N = 100$ and print out the components of position and velocity to a data file. Plot the resulting coordinate data on the (x, y) plane using `gnuplot` or an alternative of your choice. Repeat the process using the RK4 integrator and plot the resulting orbit on top of the one obtained with Euler's method. Make a qualitative comparison between the two (zoom in if needed). [**3**]

(c) Repeat the process to evolve the orbit for a full day (86400 seconds), and print out the data of coordinates and velocities every 60 seconds. Again plot the results for the two ODE integration methods and compare. According to Newtonian dynamics, this should be a closed orbit, i.e. the satellite should return to its initial position and velocity after a full orbit. Are the results consistent with this expected property? [**3**]

(d) To simplify the problem, we have neglected many secondary effects, among which is the gravitational pull of the Moon. Assume the Lunar position to be fixed at $(x_L, y_L) = (384400000.0, 0.0)$ [m]. Add its gravitational effect as an **additional** $-\frac{GM}{r^3}\vec{r}$ term on the RHS of the ODE, replacing

$$M \to M_L \quad \text{and} \quad \vec{r}(t) \to (x(t) - x_L, y(t) - y_L)\,.$$

Define the value of the lunar mass to $M_L = 7.342 \times 10^{22}$kg. Evolve the ODE for a full day using the RK4 method and plot the orbit as before. Quantify the difference. [**5**]

(e) Increase the initial velocity in steps of 10% and keep doing so, until your plot shows the satellite shooting beyond the Moon during its first orbit. Increase the total integration time to a few days when necessary. Note down the initial velocity and submit the plot of the orbit. [**4**]

(f) Bonus [5 marks]: Give a brief description of how you would modify the system of ODEs to better represent the full dynamics of the Earth-Moon-Satellite system (still in 2D). Take into account that both the Moon and the Earth are moving under each other's gravitational pull. What would be the dimensionality of the problem, what would be the dynamical variables, what would you include in the RHS and how would you set up initial data? Since the Earth is wobbling around due to the Moon's gravitational pull, is there a better point to place the origin of the coordinate axes?

 Continue to next page