BEng Biomedical Engineering
Object-Oriented Programming

Coursework 3

Objective

To gain practical experience of object-oriented design and programming using C++ on a biomedical problem.

Introduction

A common task in biomedical imaging is to spatially realign two datasets acquired from the same subject or sample in different sessions, since the positioning of the subject will in general differ between sessions. This type of problem is typically addressed using so-called *registration* methods, which aim to identify the spatial transformation that best matches positions in one dataset with the corresponding positions in the other dataset. There are many ways to approach this problem, depending on the nature of the data, and many types of transformation that can be used, from rigid-body (i.e. rotation and translation) to full non-linear warping.

In this task, we are provided with the 3D positions of a set of N corresponding landmarks, identified in two datasets, labelled A and B. We need to identify the transformation that provides the best match between the positions of each landmark in A with its corresponding landmark in B. We define 'best matching' as the transformation that minimises the sum of squared Euclidian distances between corresponding landmarks. This can be written as:

$$\theta_{opt} = \min_{\theta} \sum_{i}^{N} \|\mathbf{a}_i - \mathcal{T}_\theta(\mathbf{b}_i)\|^2 \tag{1}$$

where $\mathbf{a}_i$ and $\mathbf{b}_i$ correspond to the 3-vectors of $x/y/z$ coordinates for point i in datasets A and B respectively, and $\mathcal{T}_\theta : \mathbf{x} \rightarrow \mathbf{y}$ is the transformation represented by parameters θ. In other words, $\mathbf{y} = \mathcal{T}_\theta(\mathbf{x})$ maps 3D point $\mathbf{x}$ onto another 3D point $\mathbf{y}$ according to the parameters of the transformation θ.

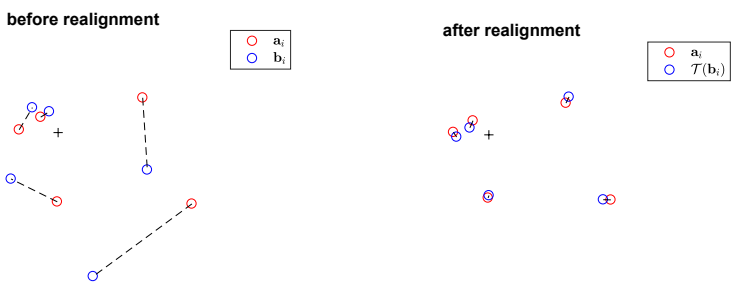

Figure 1: A 2D illustration of the point-matching problem.

We will consider affine transformations, so we will need to estimate the 12 parameters of the affine transformation that provides the best match between A and B. An affine transformation is a linear transformation of the form:

$$\mathcal{T}_\theta(\mathbf{x}) = \mathbf{R}\mathbf{x} + \mathbf{t} \tag{2}$$

where $\mathbf{R}$ is a 3×3 matrix (capable of representing any rotation, scaling, or shear), and $\mathbf{t}$ is a translation vector. Such a transformation can be uniquely represented by the 9 independent components of $\mathbf{R}$ and the 3 independent components of $\mathbf{t}$, giving a total of 12 independent parameters; in other words, the parameter vector $\theta = \{R_{i,j}\} \cup \{t_i\}$.

The registration problem involves solving Eq. (1), i.e. minimising the cost function that quantifies how well the data match, with respect to the parameters of the transformation. This is a classic optimisation problem, for which many algorithms exist. Optimisation methods are typically implemented as general-purpose algorithms to solve problems of the type:

$$\theta_{\text{opt}} = \min_\theta \mathcal{F}(\theta) \tag{3}$$

where $\mathcal{F}$ is the *objective function* (also known as the cost function), which depends on a set of parameters θ. The solution θ_{opt} corresponds to the parameter set that minimises the value of the objective function $\mathcal{F}(\theta)$.

The optimisation will be performed using a simplified *genetic algorithm*, which works as follows. Starting from an initial population of candidate parameter sets, the top ranking candidates are identified, and a new population is generated by mixing their parameters in a manner similar to the way children inherit genetic material from their parents. By only allowing the top ranking solutions to produce the next generation, the range of parameter values in the population will gradually converge to the optimal solution over successive *generations*. Genetic algorithms can be useful in solving complex problems with many local minima that are difficult to solve reliably using other methods.

The algorithm to be implemented in this task will start by initialising a population of N_c candidate sets, with parameters drawn at random from a uniform distribution within specified bounds. The cost function is computed for each of these parameters sets, and the best N_s solutions are then selected. These top solutions are then used to produce the next generation, as follows.

- For each new candidate θ^{new}:
  - Pick two of the parent candidates θ^i and θ^j at random from the pool of top solutions (with uniform probability).
  - For each parameter p:
    - Generate a random number f between 0 and 1.
    - Set the value of the parameter p for the new candidate as the weighted average of the corresponding parameters of its parents:
      $\theta_p^{\text{new}} = f\theta_p^i + (1-f)\theta_p^j$.

This process is repeated for the next generation. The algorithm terminates when the range of parameter values within the population is within a user-specified tolerance; in other words, when all candidate sets have values that are very close to each other. This is determined by finding the minimum and maximum values of each parameter in the population and computing the difference between them. When the range of all paremeters is smaller than the tolerance, the algorithm terminates, and the best candidate in the current population is returned as the solution.

Instructions

Your task in this coursework is to write a C++ program to compute the parameters of the affine transformation that minimise the mismatch between the landmarks in datasets A and B; in other words, to solve Eq. (1), where the transformation $\mathcal{T}_\theta$ is as written in Eq. (2), using a simplified genetic algorithm solver.

Genetic Algorithm Solver

Your program should implement a genetic algorithm solver. To maximise code re-use, your implementation should be capable of operating on any generic minimisation problem. This can be achieved using either compile-time or runtime polymorphism. Your implementation should allow the user to supply the problem to be solved as a separate C++ class, with a method to compute and return the value of the objective function given a vector of parameters, and a method to report the number of parameters in the problem.

Your implementation will need to maintain a list of N_c candidate parameter sets, along with their associated cost as computed by the problem class for these parameters. These candidates will need to be sorted in order of increasing cost, and the top N_s solutions extracted to be used as parents to form the next generation.

The solver implementation will need functionality to perform these distinct operations:

- initialise its various parameters, including setting the size of the parameter vector as reported by the problem to be solved;
- initialise the population of candidates, along with their cost, by generating random values with uniform probability within an interval defined by appropriate upper and lower bounds (one per parameter);
- perform a single iteration of the algorithm, by creating the next generation of N_c candidates from the set of N_s parents, computing their costs, and sorting this list in order of increasing cost;
- query the maximum range over all parameters, so that convergence can be assessed;
- provide the current best candidate parameter set.

The simplest way to generate random numbers is to use the standard `std::rand()` function: this is a simple pseudo-random number generator. Successive calls to this function return integer values uniformly distributed between 0 and `RAND_MAX` (a constant defined in the `<cstdlib>` header). To produce a random floating-point number between zero and one, use a construct such as `double(rand())/double(RAND_MAX)` (the cast to `double` is necessary to ensure floating-point division, otherwise this will assume integer rules and return an integer). An integer within a range can be obtained using the modulo operator, e.g. `rand() % 100`. See https://en.cppreference.com/w/cpp/numeric/random/rand for full details.

To sort the candidates in order of increasing cost function value, you are encouraged to use the standard C++ STL `std::sort()` function with your own comparator function. Examples of how to do this are available online, for example: https://www.geeksforgeeks.org/sort-c-stl/

The Problem

Your program should define a C++ class representing the matching problem to be solved in this task. The cost itself is to be computed as the sum of squared Euclidian distances between each point in set A and the corresponding *transformed* point in set B, as defined in Eq. (1).

Data input

Your program should read two data files, one for each of the datasets to be matched. You are provided with two such data files, called `A.txt` and `B.txt` respectively. The contents of these files are in the following format:

```
36
   8.4929744e-01   1.3372380e+00   2.8000747e-01
   4.6930718e-01  -1.6163173e+00   8.4605887e-02
  -2.3978170e-01  -4.9181759e-01   7.2911906e-01
   ...
```

The first entry (on the first line) is an integer specifying the number of landmarks contained in the file. The next 3 entries are floating-point values corresponding the $\{x, y, z\}$ components of the position of the first landmark. The next 3 entries correspond to the second landmark, and so on.

Your program should verify that the files are valid (i.e. contain at least one landmark), and contain the same number of landmarks.

The main function

Your program should make use of the functionality defined in the previous sections, and perform the following actions:

- load the data files;
- set up the problem class;
- set up the genetic algorithm solver, using the problem class;
- set up the parameters of the solver, with $N_c = 5000$ and $N_s = 100$;
- initialise the solver with upper and lower bounds set to $[-2, 2]$ for all parameters;
- iterate until convergence, deemed to be when all parameters have a range smaller than 10^{-6};
- extract and print the solution.

Reporting Requirements

You should submit a Code::Blocks C++ project that meets as many of the requirements as possible.

You should also submit a short written report explaining your object-oriented design. The report should include a UML class diagram and a short explanation of the meanings of the class(es), attribute(s) and behaviour(s), as well as a brief description of the object-oriented design process that you went through to produce your design.

To help you in producing this report, a sample model report will be made available to you through the KEATS system (called *Sample model report for coursework 3*).

Your report should not need to be longer than 3 pages of A4, and can be shorter.

Submission will be via the KEATS system.

The submission point will only allow you to upload a single file so you should combine all files into a single *zip* file.

The name of the zip file should be in the format SURNAME-FIRSTNAME-CW3.zip.

The hand-in date is **11 April 2019, 5 pm**.

Late submissions (within 24 hours of this deadline) will be accepted but will be capped at the module pass mark (i.e. 40%).

If your program does not meet all requirements then please submit what you have written by the deadline.

Assessment

Your coursework will be marked on a number of factors:

- Does the program work? Does it meet all requirements? Has it been tested extensively? (50%)
- Program design and appropriate use of C++ language features, e.g. functions, classes, composition/aggregation/inheritance as appropriate, etc. (30%)
- Written report (15%)
- Use of comments, indentation and variable/function names to make code easy to understand (5%)

The overall mark for this coursework will make up 30% of your total mark for this module.

This is an individual assignment. You are not permitted to work together with any other student. Note that general discussions about design decisions and/or coding strategies are permitted, and such discussions can be a useful learning experience for you. But you should not, under any circumstances, share details of designs or code.

If you have any questions about this coursework please contact:

- Donald Tournier (jacques-donald.tournier@kcl.ac.uk)