Optimal mineral recovery using Genetic Algorithms

Introduction

Separation technologies are widely used to improve the purity of products. These technologies usually take the form of identical or near identical **separation units** (referred to as **units** for brevity) that are arranged in **circuits**. In minerals processing, for instance, the separation units are things like flotation cells or spirals, while in the upgrading of nuclear material the separator unit will be a centrifuge. While the separation units are different, their key property is that they will attempt to recover the "valuable" material to a concentrate stream, while simultaneously attempting to reject the "waste" material to a tailings stream (Fig. 1). The challenge is that individual units are typically inefficient in that they will both misclassify some of the waste material into the concentrate stream and not recover all of the valuable material to concentrate stream (i.e. some valuable material will report to the tailings stream).

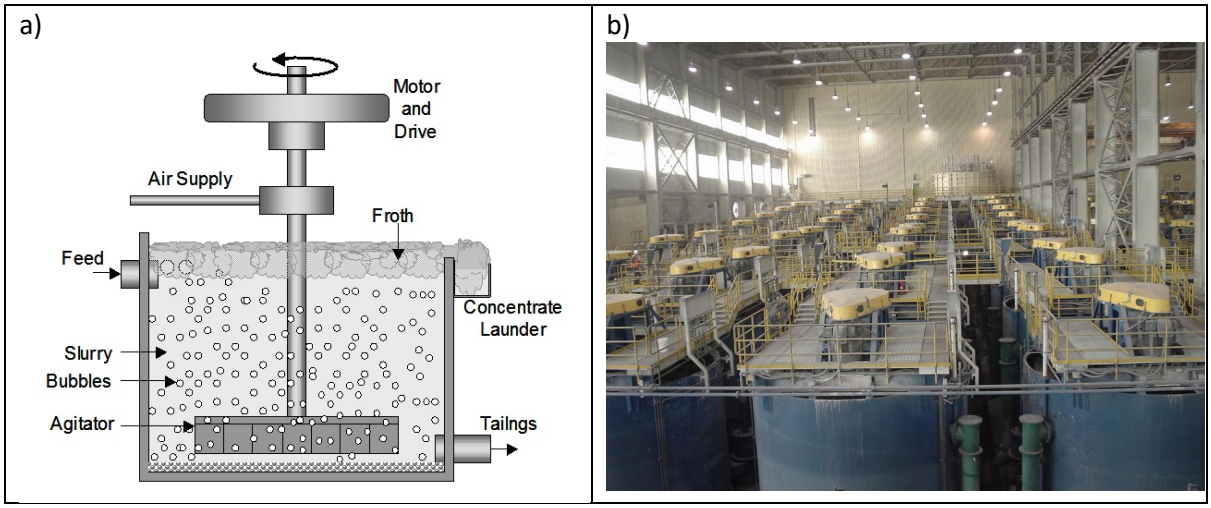

Figure 1: a) Schematic of a froth flotation cell (an example of a separation unit), which produces a concentrate stream via the froth with the rest of the material flowing out of the unit as tailings b) Picture of a large number of flotation cells arranged as a circuit

While a single separation unit in isolation is often inefficient, multiple units can be combined together in circuits that can both enhance recovery of the valuable material and improve the purity of the final product. While some circuit designs will be unambiguously better than others (i.e. produce both better recoveries and purities), for many designs there will be a compromise between the overall recovery (total mass of valuable material recovered) and purity (proportion of valuable material to the total material recovered). In such circumstances the optimum circuit will be an economic decision based on the balance between how much you are paid for the product and penalised for a lack purity.

In this project we are going to restrict ourselves to both units and an overall circuit that produces two products (**streams**), namely a **concentrate stream** and a **tailings stream**. Let us consider extraction of a rare mineral known as gormanium, from its ore gormanite. The success of the circuit will be measured based on the first product, the concentrate stream, with a price paid per kg of the valuable material (gormanium) and a penalty charged per kg of waste material in the concentrate stream. The second product, the tailings stream, will be dominated by waste material and discarded at no cost or penalty. In both individual units and the overall circuit, intermediate products could be produced, but we will ignore this as it dramatically increases the complexity of any potential optimisation.

These circuits can have simple rows of units with the tailings or concentrate (but not both) being passed to the next unit along. The circuits can also involve recycles, where a stream is passed back to a unit nearer the beginning of the circuit (but not the same unit). This means that the number of potential circuits increases factorially with the number of units in the circuits. A brute force approach is thus only feasible for circuits consisting of relatively few units (by the time there are 30 units in the circuit there are more potential circuit configurations than there are atoms in the universe!).

The large number of possible circuit configurations necessitates an optimisation algorithm to search for a solution. As the configurations are discrete, standard gradient search algorithms won't work. There are a number of potential algorithms that can be applied to such problems and the one that we will be using is called a **genetic algorithm**.

Methodology

Genetic Algorithms

Genetic algorithms, as their name implies, work in a manner not dissimilar to how natural selection works. The heart of the algorithm is a representation of the problem as a vector of numbers (the "genetic code" of the problem). In this problem, the genetic code represents the connections in the circuit (see fig. 2). To start, a large number of these vectors need to be randomly generated and evaluated, with a single number assigned to represent the success of each vector (i.e., the performance of the circuit). The function that takes in the vector and returns a single performance number is often referred to as the **fitness function**. The convergence of the algorithm will usually be enhanced if it can be ensured that every one of the initial vectors is valid.

The set of random initial vectors will form the **parents** for the next generation of offspring vectors via a combination of two processes:

Mutations – Random changes in the numbers in the parent vector.

Crossover – This is roughly equivalent to sexual reproduction. In this process a portion of one parent vector is swapped with a portion of another parent vector. The motivation for swapping a portion of a parent vector with another rather than swapping individual values randomly is that, over successive generations, values that work well together will end up next to one another in the vector (roughly equivalent to genes); preserving these portions of the genetic code is beneficial. In this problem, for instance, where the values in the vector represent connections in the circuit, a certain set of connections between a few units may be useful in more than one location in the overall circuit.

The steps in a basic genetic algorithm are as follows:

1) Start with the vectors representing the initial random collection of valid circuits
2) Calculate the fitness value for each of these vectors

You now wish to create *n* child vectors

3) Take the best vector (the one with the highest fitness value) into the child list unchanged (you want to keep the best solution).
4) Select a pair of the parent vectors with a probability that depends on the fitness value. In this case you might want to start by using a probability that either varies linearly between the minimum and maximum finesses of the current population. This should be done "with replacement", which means that parents should be able to be selected more than once.
5) Randomly decide if the parents should crossover. If they don't cross they both go to the next step unchanged. If they are to cross, a random point in the vector is chosen and all of the values before that point are swapped with the corresponding points in the other vector.
6) Go over each of the numbers in both the vectors and decide whether to mutate them (this should be quite a small probability). If the value is to be mutated, you should move the value by a random amount (you can decide the step size). In these circuits you can avoid clustering of the results near the minimum and maximum unit numbers by "wrapping" the change (i.e. don't artificially restrict the movement, rather use a modulus to bring it back within range). In the circuit problem, values are essentially completely independent of one another as there is no reason to think that a connection to a unit with a number close to that of the currently connected unit will be better than the connection to any other unit. This means

that the potential step size can be set to be the same as the size of the valid range, though in other optimisation problems it may be useful to preferentially search values close to the current one.

7) Check the validity of each of these potential new vectors and, if they are valid, add them to the list of child vectors.
8) Repeat this process from step 4 until there are *n* child vectors
9) Replace the parent vectors with these child vectors and repeat the process from step 2 until either a set number of iterations have been completed or a threshold has been met (e.g. the best vector has not changed for a sufficiently large number of iterations).

Tuning the algorithm – You may notice that there are a number of things that you can tune when running these algorithms. These include:

- the number of offspring *n* that are evaluated in each generation
- the probability of crossing selected parents rather than passing them into the mutation step unchanged (a recommended range is between 0.8 and 1)
- the rate at which mutations are introduced (recommended probabilities of 1% or lower for each value in the vector)

You will need to investigate how these numbers change the rate of convergence achieved. The optimum parameters will depend on both the type of problem being investigated and the size of the problem (they will be different between your test problem used to develop the genetic algorithm and the actual circuit simulation, and they will vary with the number of units in the circuit).

Before the optimum circuit can be determined, though, we need to be able to evaluate the performance of a given circuit.

Modelling a Circuit

There are a number of aspects that need to be covered in terms of calculating circuit performance. The first is representing the circuit as a vector:

Representing the Circuit – In these circuits the separation units can take in as many input streams as desired, but they must have only two output streams each. We can therefore represent the circuit as a vector of stream destinations. We call this the circuit vector and this forms the genetic code representing the circuit. The following are a couple of examples of circuit vectors and the corresponding circuit layouts:

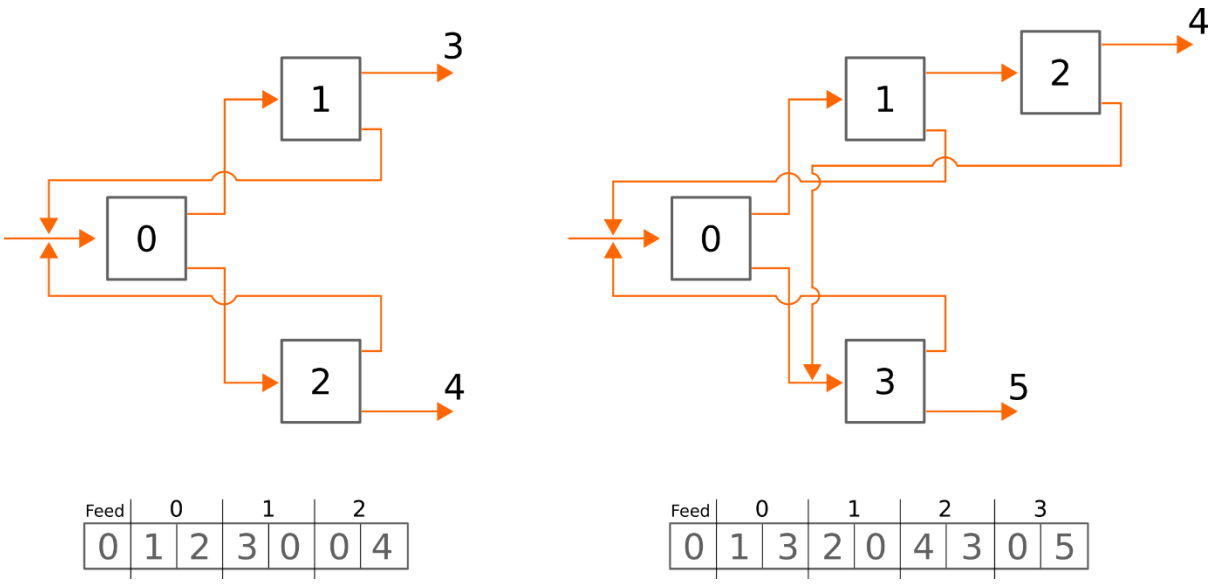

Figure 2: Two simple circuits and their corresponding circuit vector ("genetic code")

Length of the circuit vector: As well as recording the destination unit of each output stream, the genetic code for the circuit must also include the destination unit of the circuit **feed** (the input to whole circuit). This means that if there are *n* units in the circuit, the circuit vector will be 2*n*+1 elements long; one element for the circuit feed, and then two elements for each unit, where the first element is the destination for the unit's concentrate stream and the second element is the destination for the unit's tailing stream.

Contents of the circuit vector: Each element of the circuit vector defines the unit number (index) for the destination of that stream. In addition to the *n* units, streams can also be directed to the final tailings stream or the final concentrate stream. The final concentrate stream is the product to be evaluated. The first element of the circuit vector, which represents the circuit feed, must take a value between 0 and *n-1*, as the feed must go into one of the *n* units in the circuit. The remaining elements of the vector can be the index of a destination unit in the circuit (0 to *n*-1) or the index of the final concentrate stream (*n*) or the index of the tailings stream (*n*+1).

Modelling the performance of a Unit – The performance of a separation unit operation will depend on a number factors, including the feed composition and feed rate. We will assume that there are only two components in the feed, namely a waste and a valuable component (in many systems there will actually be a range of valuable and waste components with different separation performances).

We will use the simplest possible model in which we will assume that the fraction of each component that is recovered to each of the streams is the same in each unit and does not change with the feed rate. Each unit will recover 20% of the valuable material to the concentrate together with 5% of the waste. So, if the feed into a unit comprises 10 kg/s gormanium and 100 kg/s waste material, the concentrate stream output from the unit will be 10 x 0.2 = 2 kg/s gormanium and 100 x 0.05 = 5 kg/s waste; the tailings stream will therefore contain 10 – 2 = 8 kg/s gormanium and 100 – 5 = 95 kg/s waste.

Modelling the Circuit – To combine the behaviour of each of the individual units into an overall circuit simulation we must calculate the mass flow rate of each component in each stream. To do this we will assume that the circuit is at steady state, which implies that the flows in each stream do

not change with time and that there is no accumulation (i.e. the total feed into a unit is equal to the sum of the flow out of the unit through the two output streams).

As these circuits can (and will typically) involve recycles—that is one or more of the output streams can feed back into an earlier unit in the circuit—the circuit performance will need to be solved iteratively. (Note that, as this particular problem has a linear relationship between feed and product, you could solve the circuit directly by matrix inversion, though this would not be possible for more complex unit performance models). Successive substitution of the component mass flows in the streams is guaranteed to converge in these types of problems (you can look up the proof if you wish) and so this is the approach you can start with. It is possible to have quicker convergence using a more complex convergence algorithm, though you do need to ensure stability of convergence.

The following is a simple successive substitution algorithm that is guaranteed to converge if a valid solution exists. Lack of convergence is thus a test of circuit validity, though a computationally expensive way to evaluate circuit validity (NB: If you develop a better test of circuit validity, you should still check for lack of convergence as there are some pathological circuit configurations that you might not think of in the validity testing):

1) Give an initial guess for the feed rate (mass per second) of both components (gormanium and waste) to every cell in the circuit (as an initial guess you can assume the same feed rate everywhere; 10 kg/s gormanium; 100 kg/s waste)
2) For each unit, use the current guess of the feed (input) flowrate of each component to calculate the output flowrate of each component via both the concentrate and the tailings streams
3) Store the current value of the feed of each component into each cell as “old” feed values and then set the current value of the feeds for each component to zero
4) For the cell receiving the circuit feed, set the feed of each component equal to the flowrate of the circuit feed: i.e., 10 kg/s for the gormanimum feed and 100 kg/s for the waste feed
5) Then, for the current unit, consider first the concentrate stream. Add the flowrates of the components in this stream (calculated in step 2) to the flowrate of the relevant component in the feed going into the destination unit (or final concentrate stream) at the end of the concentrate stream, based on the linkages in the circuit vector. Repeat this procedure for the tailings stream, which will increment the feeds of gormanium and waste to a different unit in the circuit (or the final tailings stream).
6) Move to the next unit and repeat step (5) for each unit in the circuit. You do not need to do this in any particular order as long as each unit is visited once and once only and thus you can simply loop through the list of units in unit number order. After visiting all units, a new estimate for the feed of gormanium and waste into each unit is determined.
7) For each component, check the difference between the newly calculated feed rate and the old feed rate for each cell. If any of them have a relative change that is above a given threshold (1.0e-6 might be appropriate) then repeat from step 2 (you should also leave this loop if a given number of iterations has been exceeded or if there is another indication of lack of convergence).
8) Based on the flowrates of gormanium and waste through the final concentrate stream calculate a performance value for the circuit. If there is no convergence you may wish to use the worst possible performance as the performance value (the flowrate of waste in the feed times the value of the waste, which is usually a negative number).

Checking Circuit Validity – A key step before running a simulation is to ensure that the circuit is a valid one. For the circuit to be valid a few conditions must be met:

- Every unit must be accessible from the feed. I.e. there must be a route that goes forward from one unit
- Every unit must have a route forward to both of the outlet streams. A circuit with no route to any of the outlet streams will result in accumulation and therefore no valid steady state mass balance. If there is a route to only one outlet then the circuit should be able to converge, but there will be one or more units that are not contributing to the separation and could therefore be replaced with a pipe.
- There should be no self-recycle. In other words, no unit should have itself as the destination for either of the two product streams.
- The destination for both products from a unit should not be the same unit.

Note that this is not an exhaustive list of how circuits can be invalid or obviously sub-optimal. You should still check if the circuit simulation is diverging as this will indicate an invalid circuit (have a maximum number of iterations allowed in the circuit mass balance convergence).

When doing these checks you should note that the circuit takes the form of a directed graph. It is often easiest to write recursive functions to traverse the graph, though you should note that, because recycle is allowed, you do need to ensure that you don't get stuck going around a recycle loop. The easiest way to do this is to mark the units that you have already visited. A simple generic function for using recursion to traverse the circuit is outlined in the section below.

Traversing the Circuit using Recursion

Recursive functions are the easiest way to traverse a tree or graph. In this short code snippet I will demonstrate a function which marks every unit which is accessible from a given unit (i.e. every unit that product from a given unit can potentially reach). I will assume that the data for each individual unit is stored in a class:

```
class CUnit
{
public:
        //index of the unit to which this unit's concentrate stream is connected
        int conc_num;
        //index of the unit to which this unit's concentrate stream is connected
        int tails_num;
        //A Boolean that is changed to true if the unit has been seen
        …other member functions and variables of CUnit

};
```

I will assume that there is an array of these units:

```
int num_units;
…set a value to num_units
vector<CUnit> units(num_units);
```

The following function is recursive, which means that it calls other instances of itself within the function.

```
void mark_units(int unit_num)
{
        if (units[unit_num].mark)          //If we have seen this unit already exit
                return;
        units[unit_num].mark = true;     //Mark that we have now seen the unit

        //If conc_num does not point at a circuit outlet recursively call the function
        if (units[unit_num].conc_num<num_units)
                mark_units(units[unit_num].conc_num);
        else
                …Potentially do something to indicate that you have seen an exit

        //If tails_num does not point at a circuit outlet recursively call the function
        if (units[unit_num].tails_num<num_units)
                mark_units(units[unit_num].tails_num);
        else
                …Potentially do something to indicate that you have seen an exit
}
```

To use this function in the code you need to follow the following steps:

…Use the specification vector to set the *conc_num and tails_num values for every unit in the units array*

```
for (int i=0;i<num_units;i++)
        units[i].mark = false;

//Mark every cell that start_unit can see
mark_units(start_unit);

for (int i=0;i<num_units;i++)
        if (units[i].mark)
                …You have seen unit i
        else
                …You have not seen unit i
```

Your Base Case Circuit Specification

Your base case circuit should contain 10 units. It should have a total circuit feed of 10 kg/s of the gormanium (valuable material) and 100 kg/s of the waste material (note that, as our unit's performance does not have a feed rate dependency, it is only the relative amount of the two components that will influence the optimum configuration). Each unit will recover 20% of the valuable material to the concentrate together with 5% of the waste. You will be paid £100 per kg of gormanium in the product (final concentrate stream) and charged £500 per kg of the waste material in the same stream.

What is required of each team?

Tasks

1) Create software capable of using a genetic algorithm to optimise a system represented by a specification (circuit) vector where the performance is based on the evaluation of a fitness function that takes in the vector and returns a single number to be maximised.
2) Write a mass balance simulator which can take in a circuit vector and calculate the mass flows of each component (gormanium and waste) in every stream.
3) Write a validity checking function that takes in a circuit vector and returns true or false based on its assessment of the circuit validity.

Note that you should ideally write the implementation of both genetic algorithm and the circuit simulators in a modular fashion. In this way you could either use the genetic algorithm on a different problem or apply a different optimisation algorithm to the circuit simulator.

4) Obtain the optimum circuit configuration for the base case specifications. How quickly does the algorithm converge on the optimum and how does this change with the genetic algorithm parameters such as the number of child vectors used and the mutation rate?
5) Investigate how the optimum circuit changes as the various model parameters change. Note that you will usually have to do large changes in these parameters to drive significant changes in the optimum circuit configuration. Are there any circuit design heuristics that you might recommend based on the observed trends in the optimum configuration?
6) The fact that the genetic algorithm requires the evaluation of a large number of independent circuit configurations at each iteration means that it is readily parallelised. It is up to you whether you do shared memory openMP parallelisation or distributed memory MPI parallelisation. You could also write a script to test different parameter settings by running different instances of your serial code in parallel, though this would get less credit than the implementation of a truly parallel code.

Note that you are not expected to complete all of these tasks, though you should complete tasks 1-4 as a minimum. These tasks also carry the greatest weighting in terms of the marking scheme.

Recommendation for how to initially proceed

There are three challenging aspects to this project that can be worked on independently before they are brought together in the final code. Before any of these tasks can be attempted, though, a data structure for the circuit vector needs to be agreed. Additionally, for testing of the circuit validity and carrying out circuit simulation it would be useful to agree data structures for the individual units, streams, etc.

The following are three tasks that can be independently tackled:

Circuit Simulation – For a given circuit vector you need to be able to calculate the mass balance over all the units and thereby the composition of both of the outlet streams as this is required to produce a single fitness value. You can develop this simulator without validity checks or the genetic algorithm by manually creating a few test vectors. This can be done by drawing a circuit that you can see is valid and then writing out the corresponding circuit vector. In drawing the test circuits, you should design complex circuits with lots of recycles rather than circuits you think may be efficient in order to have a good test of the circuit simulator's convergence behaviour.

You might want to use the following form for the circuit performance/evaluation function definition. This form assumes that you are representing your individual vectors as a simple arrays of integers:

*double Evaluate_Circuit(int *circuit_vector, double tolerance, int max_iterations)*

Circuit Validity – Having valid circuits is important for their ability to have the mass balance converge. While you could use non-convergence of the mass balance as an indicator of an invalid circuit, it is much better to explicitly check circuit validity. Checking circuit validity can be done without having the ability to carry out a mass balance on it and can therefore be done as an independent task (how these circuits might be invalid is discussed in an earlier section).

The following function definition can be used for this function. The form of the circuit vector must be the same as that used by the sub-group developing the circuit simulator.

*bool Check_Validity(int *circuit_vector)*

The Genetic Algorithm – This can be developed without the circuit simulator as you simply need a fitness function that gives back a value based on a given number vector. While this will ultimately be the value of the concentrate stream as calculated in the circuit simulator, in order to develop the algorithm a simple test case is to choose a random vector as the answer and have the fitness function return a number that represents the difference between this answer vector and test vector. The advantage of this approach is that you know if the answer returned is the correct one. Don't rely on this test problem to tune for the correct parameters to use in the genetic algorithm as this is a dramatically simpler optimisation than the actual one and therefore should converge much more quickly.

When developing the genetic algorithm (i.e. before the other sub-groups have completed their tasks) you might want to use the following test versions of the above functions:

```
bool Check_Validity(int *circuit_vector)
{
        return true;
}
```

```
double Evaluate_Circuit(int *circuit_vector, double tolerance, int max_iterations)
{
	double Performance =0.0;
	for (int i=0;i<vector_size;i++)
	{
		//answer_vector is a predetermined answer vector (same size as circuit_vector)
		Performance+=(20-abs(circuit_vector[i]-answer_vector[i])*100.0;
	}
	return Performance;
}
```

Assessment

Your group project will be assessed in four ways:

Software (50 marks)

Software will be assessed based on functionality (25/50 marks) and sustainability (25/50 marks).

Functionality (25 marks): Your software will be assessed on its ability to perform the required tasks. Up to 15 marks will be awarded for successfully completing tasks 1-4; 10 marks being available for the more challenging later tasks. Marks will be deducted for (a) inaccuracy in the solution; (b) bugs or mistakes in implementation; (c) computational inefficiency.

Sustainability (25 marks): As with all software projects, you should employ all the elements of best practice in software development that you have learned so far. A GitHub repository will be created for your project to host your software. The quality and sustainability of your software and its documentation will be assessed based on your final repository and how it evolves during the week. Please refer to the ACSE-4 handbook for more information about the assessment of software quality. You should have test cases for the different aspects of the code, including the genetic algorithm, the circuit simulator and the circuit validity checking. Other important aspects of the sustainability of the project include sufficient and clear documentation of the code, clear descriptions of the code's usage, as well as examples and test cases.

Technical report (30 marks)

By the deadline (Friday 22nd March, 12:00 noon), you must produce a technical report that presents the following elements, in the form of either a PDF or a Jupyter notebook (in your GitHub repo):

1. A brief description of your solution algorithms. This should include descriptions of the implementation of the genetic algorithm, the circuit simulation and the validity checking. You should not rewrite the instructions, but rather describe the specifics of your implementation and, especially, your approaches to optimisation and the overcoming of implementation challenges. (10 marks)

2. The optimum circuit you found for the base case configuration. This should include both the vector for the optimum circuit, as well as figure showing this optimum configuration. (3 marks)

3. A description of the convergence behaviour of the genetic algorithm and how this varies with the different algorithm parameters. (4 marks)

4. A description of how the optimum circuit configuration changes as the model parameters change, as well as a discussion of why this might be the case and any circuit design heuristics that you might recommend (5 marks)

5. Describe the parallelisation of the code and how the performance of the algorithm varies with the number of nodes used (parallel scaling) (4 marks).

6. Layout and presentation of the Technical Report (4 marks).

Presentation (10 marks)

On Friday afternoon, you must present your software and analysis to the class. You will have 20 minutes, plus questions, for your presentation. Your presentation should cover all of the completed tasks. We recommend that your presentation is supported by slides made in Powerpoint or a similar tool. Your presentation should also cover a summary of how you managed the project and divided the tasks.

Teamwork (peer assessment; 10 marks)

After the presentations, you will complete a self-evaluation of your group's performance. This, together with observations made during the week by staff, will inform the teamwork component of your mark. Please refer to the ACSE-4 guidelines for more information about the assessment of teamwork.

Technical requirements

You should use the assigned GitHub repository exclusively for your project

Your software should be predominantly written in C++, though you can write some modules in Python if you wish.

Your program should be written in ANSI standard C++ so that it is able to compile under both Windows and Linux.

Your program should be able to be run from the command line without any user input so that it can be submitted to run on cx1 via a PBS script.

You should use Travis and the GitHub flow for any automated testing that you implement