Optimising a Parametric Wind Turbine Support Frame

Team 3
November 2024
Optimisation Skills 1

Problem Overview

Optimisation skills 1 introduces a small-scale wind turbine support structure. The task is to determine the most cost-effective combination of truss radius and the horizontal distance BC (figure 1). The vertical distance of the truss, forces and material's strength of the support structure are provided.

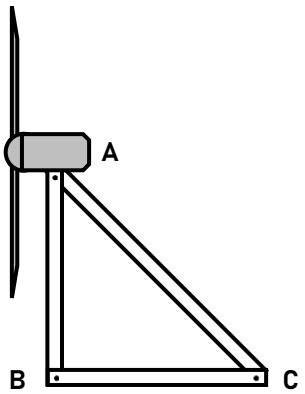


Figure 1: Diagram of turbine support structure.

Numerical Approach

To minimise the cost of the trusses, their outer radius will be reduced to the smallest possible dimension, which are constrained by the physical limits of either yield stress or buckling load. These limits depend on the force applied to each truss, which vary based on the horizontal distance between nodes. Consequently, the problem is simplified from four variable to one variable optimisation, involving only the horizontal distance.

Additionally, a variable R_C, defined as the ratio of the horizontal distance (x_C) to vertical distance (y_A), is introduced. This simplifies the equations for determining the smallest outer radius of each truss.

Truss	Force applied	Radius at yield limit	Radius at buckling limit
AB	$F_{AB} = F_y + \frac{F_x}{R_c}$	$r_{AB} = \frac{F_{AB}}{2\sigma_y \pi t} + \frac{t}{2}$	$F_{AB}\frac{4y_A^2}{\pi^3 E} = 4r_{AB}^3 t - 6r_{AB}^2 t^2 + 4r_{AB}t^3 - t^4$
BC	$F_{BC} = F_x$	$r_{BC} = \frac{F_{BC}}{2\sigma_y \pi t} + \frac{t}{2}$	$F_{BC}\frac{4R_c^2 y_A^2}{\pi^3 E} = 4r_{BC}^3 t - 6r_{BC}^2 t^2 + 4r_{BC}t^3 - t^4$
AC	$F_{AC} = \frac{F_x\sqrt{R_c^2+1}}{R_c}$	$r_{AC} = \frac{F_{AC}}{2\sigma_y \pi t} + \frac{t}{2}$	$F_{AC}\frac{4(R_C^2+1)y_A^2}{\pi^3 E} = 4r_{AC}^3 t - 6r_{AC}^2 t^2 + 4r_{AC}t^3 - t^4$

Table 1: Equations for radius of each truss at yield stress and buckling load limit.

Implementation

Buckling occurs only in trusses under compression, and requires the solution of a cubic equation, as shown in Table 1. To account for this, the MATLAB algorithm incorporated these conditions when calculating the minimum outer radius for each truss.

The algorithm compares the outer truss radii determined by yield stress and buckling load limit, selecting the larger radius to ensure structural integrity. These radii are then used to calculate the cost of each truss based on their truss length and material cost per unit volume. The total cost of all three trusses is calculated and optimised by varying R_C.

In practical applications, trusses cannot be designed exactly at the limits of yield stress or buckling load. To address this, a safety factor is applied to artificially increase the force considered in the calculations. For the subsequent cost calculations, a safety factor of 1.5 is used to ensure an additional margin of safety.

Optimisation Function

The numerical approach simplifies the problem to a single variable, but due to the complexity of the function, gradient-based optimisation methods are unsuitable. Instead, the MATLAB function fminbnd is used, which combines the golden section search and successive parabolic interpolation to find a local minimum efficiently [1].

The golden section search leverages the golden ratio (approximately 1.618) to narrow the search range in a single variable problem. Two intermediate points are calculated at 61.8% and 38.2% of the range, and the range producing the higher value at these points is eliminated, effectively reducing the search space by 38.2% in each iteration.

Successive parabolic interpolation, on the other hand, uses three specified points to estimate the local minimum or maximum of a function by fitting a parabola through these points. The parabola's local minimum or maximum is then used as a new point for the next iteration, replacing one of the initial three points. This iterative process continues until the desired accuracy is achieved.

Solution

Using the numeric approach and fminbnd optimisation function, the minimum total cost of the support structure is calculated to be £22505, achieved with a horizontal distance (x_C) of 3.87m at a safety factor of 1.5. The radii of the trusses AB, BC and AC are 14.2mm, 12mm and 47.3mm, respectively. While trusses AB and BC operate at the safety factor limit for yield stress, truss AC is under compression and operates at the safety factor limit of buckling.

The iterations of the fminbnd optimisation function are shown in figure X, with the red cross marking the final iteration that achieves the minimum cost. The figure has been overlaid with a visualisation curve of the optimisation problem. Figure Y shows a scaled representation of the optimal support structure with dimensions and forces.

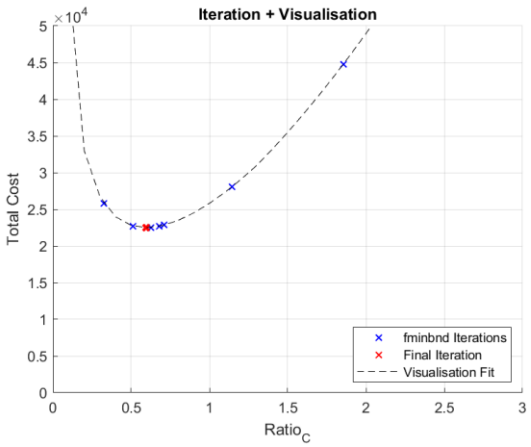


Figure 2: Optimisation iteration with visualisation overlay.

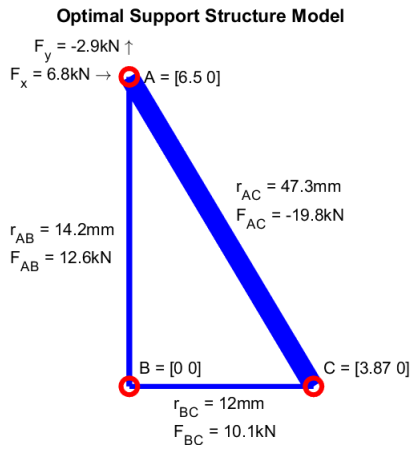


Figure 3: Dimensions of the optimal support structure.

Reflection

The optimisation algorithm uses a numerical approach and the fminbnd function to effectively minimise the total cost while calculating the horizontal length, and the outer radii of trusses AB, BC and AC.

In addition to incorporating a safety factor, the algorithm is designed for flexibility, enabling quick parameter adjustments. This allows the wind turbine manufacturer to efficiently optimise similar supports structure for varying turbine weights and thrust forces.

By reducing the optimisation variables from four to one, the algorithms significantly decreases the computational power required for each iteration. This simplification enables the generation of a 2D graph that provides a visual approximation of the model's minimum. The graph is also used to define the lower and upper bounds for the fminbnd function.

[1] MathWorks, 2024. fminbnd [Online]. Available from: https://uk.mathworks.com/help/matlab/ref/fminbnd.html [Accessed 21 November 2024]

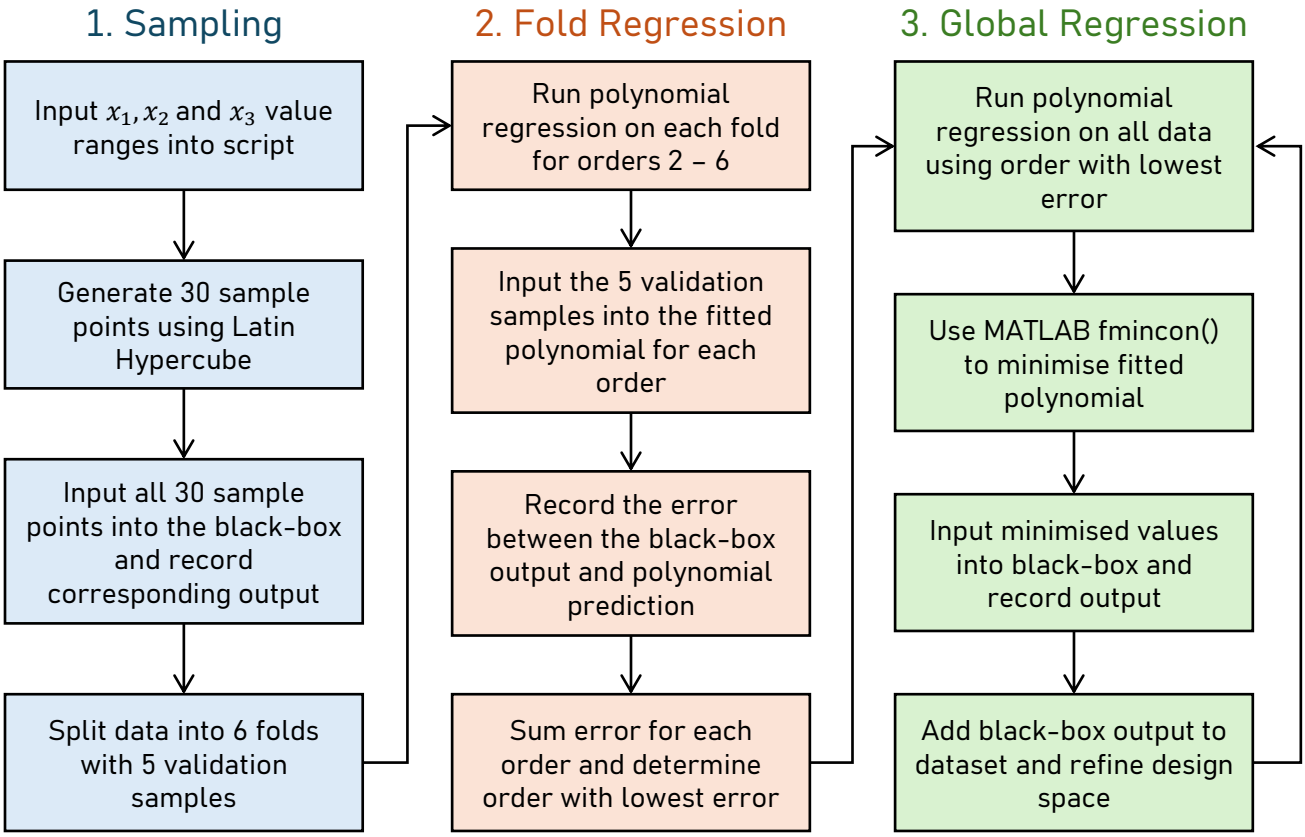
4-Dimensional Black-Box Surrogate Optimisation

Team 3
November 2024
Optimisation Skills 2

Scenario: A black-box takes three inputs x_1, x_2 and x_3 and outputs a numerical value according to an unknown function $f(x_1, x_2, x_3)$. Within 36 input attempts, find the combination of input values which minimise the black-box's output.

Approach: Use 4-dimensional polynomial regression with 6-fold cross validation, refining the design space with test run.

High Level Overview



Creating a Black-Box Simulator

A simulator was created using a variety of four-dimensional toy functions so that the optimisation process could be tested independently from the black-box. A switch-case was used such that a random toy function could be used with every optimisation run. The advantage of this was to ensure the optimisation script was reliable and applicable to a range of optimisation problems rather than falsely assume its robustness by 'tailoring' it to achieve a minimum of a specific function, ultimately allowing for more objective testing.

Polynomial Regression or Radial Basis Function?

To ensure the most reliable function approximator was used on the day, we generated 10 toy functions and tested both function generators to determine which had the most accuracy and repeatability. As seen in the next page, the radial basis function performed very poorly with the scarce data points, therefore polynomial regression was chosen and refined for usage with the real black-box.

Sample Generation

Four sampling strategies were considered: grid, random, Latin Hypercube (LHC) and Halton Sequence (HS). Immediately, grid and random sampling were discarded. When considering the D* metric to evaluate the samples' ability to capture the design space, grid sampling was especially inefficient. Whilst random sampling better captured the design space, there was a high risk of sample points becoming clustered, and thus inefficiently spread. Since an arbitrary fixed number of samples, 30, was used, the LHC was the best option as its quasi-randomness resulted in the lowest D* value and, thus, effective design space capture. The deterministic nature of the HS was not required in this situation. Figure 4 displays how the D* metric varied when taking creating 30 three-dimensional samples over 50 iterations for grid, random and LHC sampling. Figure X then visualises the design space of the sample points used in the black-box session and has also been ruled on figure 5 to show how its D* metric is suitably low compared to the LHS variation.

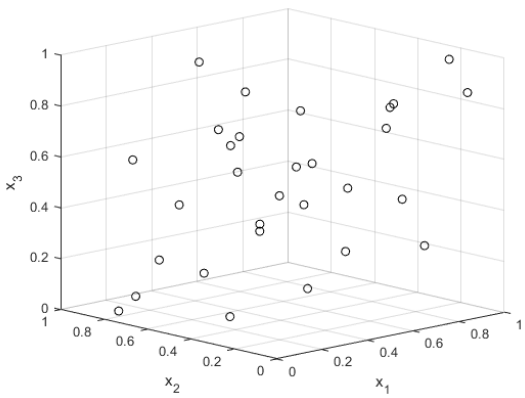


Figure 4: Samples used in black-box session with D* = 0.0954.

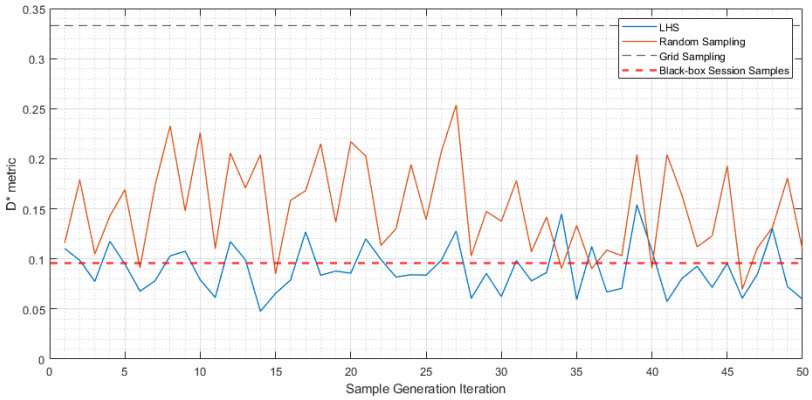


Figure 5: Variation of D* values over 50 iterations of sample generation.

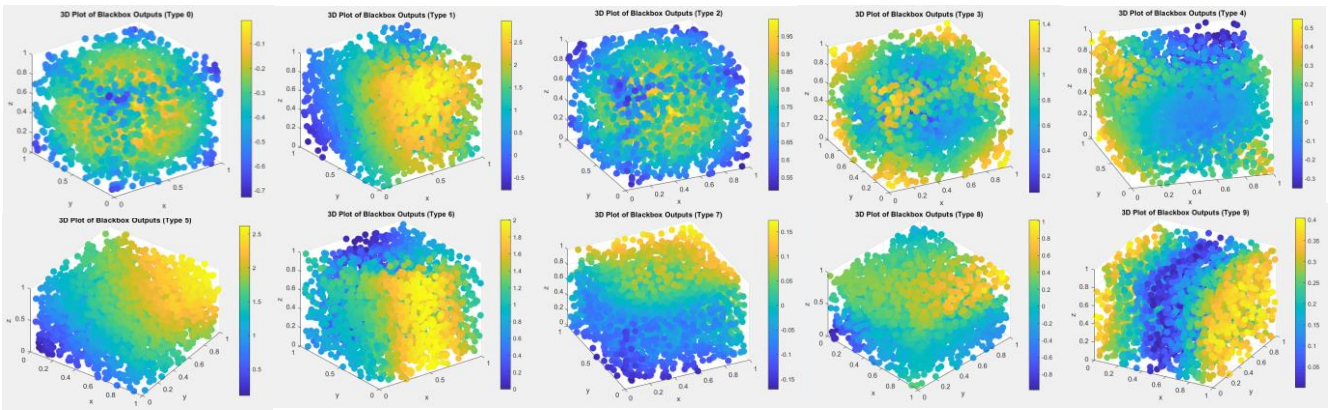
Normalisation and Data Scaling

The optimisation process used two domains: the "optimisation domain" where values are normalised between 0 to 1, and the "global" domain where values reflect the black-box. The optimisation domain is responsible for providing normalised inputs for the optimisation process, and therefore a transformation was required to traverse the domains. For x_1, x_2 and x_3, the dials of the black-box were swept to determine their minimum and maximum values. By taking the range, it was possible to use the following equations to traverse domains:

$$x_{i,\,g} = x_{i,n} * \texttt{range}(x_i) + \texttt{min}(x_i)$$

Toy Functions

In MATLAB, a function named blackbox() would take dial inputs for x_1, x_2, x_3 and the type of function desired. This would then feed into a switch case, applying the relevant function to the data points and returning `f`. These outputs have been visualised with 2,700 LHC inputs below:



Toy Function Selection

Despite all toy functions being generated with the intention of converging within the normalised domain, only toy functions 0, 2 and 3 converged. They all converged at (0.5, 0.5, 0.5), allowing for the accuracy of the radial basis function to be assessed according to this metric.

Radial Basis Function

To implement the radial basis function, 10,000 query points were generated using the Latin Hypercube in the same dimensions as the data (x_1, x_2, x_3 to x, y, z) in the normalised range (0, 1). Each query point was run through a for loop, determining the Euclidian distance between said query point and each sampled point in 3D space. The gaussian and weighted average were computed by merging this value with epsilon, a function of the length scale.

```
rbf = exp(-epsilon * r.^2); rbfValues(i) = sum(rbf .* f) / sum(rbf);
```

Why it was discarded

The radial basis function worked very well when dealing with 2,700 LHC inputs, as shown below with global minima/maxima data points for toy functions 0 and 3 respectively:

```
Maximum value: -0.19242
Coordinates of maximum point: (x, y, z) = (0.50683, 0.48998, 0.4838)
```

For function 0. Ideal maximum coords: (0.5, 0.5, 0.5)

```
Minimum value: 0.63521
Coordinates of minimum point: (x, y, z) = (0.47335, 0.50426, 0.50494)
```

For function 3. Ideal minimum coords: (0.5, 0.5, 0.5)

However, when reduced to 27 data inputs, random fluctuations in LHC sampling were enough to cause the RBF function to zone into the lowest collected data point and claim that the minima/maxima was in that region.

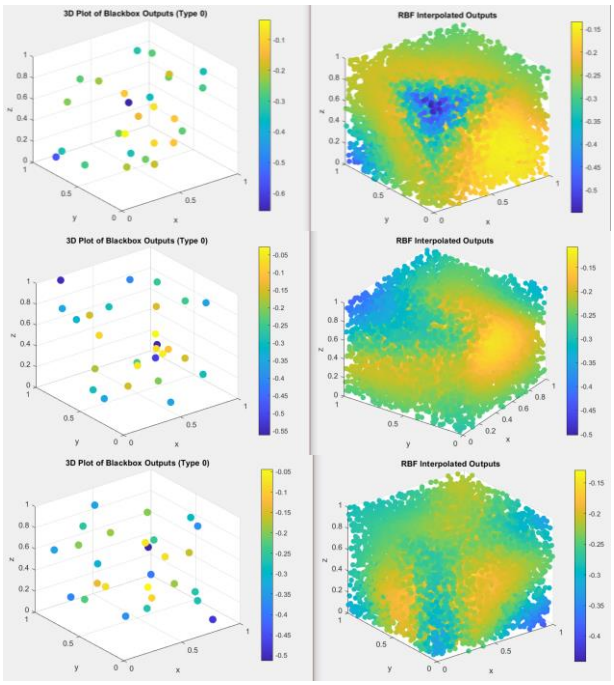
```
Maximum value: -0.21806
Coordinates of maximum point: (x, y, z) = (0.81498, 0.29318, 0.36405)
```

For function 0. Ideal maximum coords: (0.5, 0.5, 0.5)

```
Minimum value: 0.60285
Coordinates of minimum point: (x, y, z) = (0.84972, 0.15898, 0.76461)
```

For function 3. Ideal minimum coords: (0.5, 0.5, 0.5)

Figure 6: LHC sampling of toy function 0, run 3 different times. RBF outputs clearly show vastly different maxima.




```
% Gaussian RBF parameters
L = 0.5; % Length scale of the radial basis function
epsilon = 1 / L^2; % Convert L into epsilon
```

Figure 7: Varying length scale affected gradients, but did not improve accuracy.

Creation of 6-Folds

As 30 samples were to be collected, the data was divided into 6 folds. For each fold, a polynomial regression was performed on 25 samples with the remaining 5 used to evaluate errors. It would have been ideal to perform Leave-One-Out Cross Validation (LOOCV), however this would have increased the number of regression function calls from 36 (6 orders with 6 folds) to 180 (6 orders with 30 folds) and thus was omitted for efficiency.

It is ideal that the 5 validation samples are randomly selected; this is especially important when grid sampling, however concern was raised regarding the arrangement of points generated by the `lhsdesign()` function. MATLAB's documentation clearly states the points generated are randomised [1]; this simplified coding as it was possible to simply iterate over the sample points in steps of 5, retain these values as "test" data and use the remaining samples as "training" data for the polynomial regression. Training and test data were stored using cell arrays and linked by their indexes. For example, testSet{2} would contain the samples from rows 6-10, and trainingSet{2} all other samples. trainingSet{2} would then be used for regression, and testSet{2} for error evaluation.

MultiPolyRegress() Function

Initially, it was desired to use MATLAB's lsqcurvefit() function to perform the regression. Whilst this function can perform regression in N dimensions, it requires the form of the polynomial to be explicitly defined. In the case of the four-dimensional black-box function, a second-order polynomial had 10 coefficients, and a third-order 20 coefficients. Due to this, it was determined too unwieldy for higher orders. Instead, a polynomial regression function developed by Ahmet Cecen available on the MATLAB File Exchange was used [2]. This function, MultiPolyRegress(), significantly simplified coding the regression as it only required three inputs: the black-box input data, the black-box output and the desired polynomial order. The function output a cell array containing a populated polynomial as well as goodness-of-fit data. This meant for each fold a range of polynomial orders could easily be evaluated such that the script could dynamically determine which order had ideal error performance.

Overfitting Prevention

To prevent overfitting of the data, it is imperative that the number of coefficients does not exceed the number of samples. The number of coefficients can be calculated by $\binom{n+d}{d}$ where n is the number of inputs (3 in this case), and d is the polynomial order [3]. Plotting this in MATLAB for orders 1 – 8 yields figure 8, where 30 has been marked by a black line.

The optimisation process was made more efficient by only testing polynomials orders 2 and 3 as overfitting was present above this. Ironically, this meant lsqcurvefit() could have been used as complex equations were not required to be formed for higher order polynomials. However, MultiPolyRegress() was significantly more adaptable and simpler to implement whilst also returning goodness-of-fit data, therefore continued to be used.

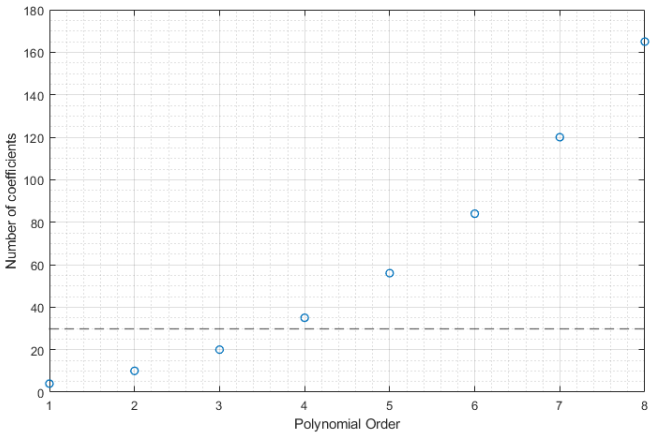


Figure 8: Number of coefficients plotted against polynomial order. Black line defines 30 coefficients.

Goodness-of-Fit & Error Calculation

Second and third order polynomial regressions were performed on the training sets using the sample data. As expected, the third order polynomial had a higher R^2 value between sets and therefore better represented the design space, as shown in figure 9. However, when using the test sets to calculate the cumulative error, the second order polynomial significantly outperformed the third-order as shown in figure 10. Due to this, second order was used for global regression.

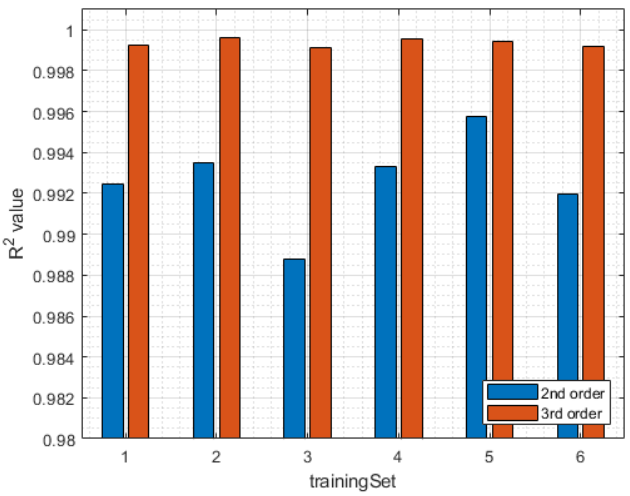


Figure 9: R^2 value of each training set with 2nd and 3rd order polynomial regression

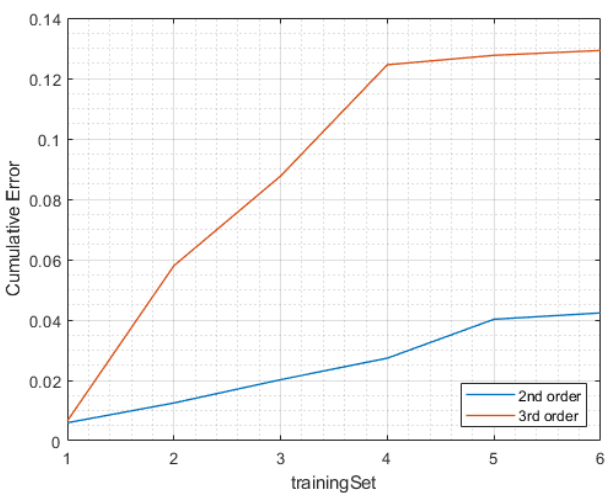


Figure 10: Cumulative error of each training set with 2nd and 3rd order polynomial regression

Polynomial Minimisation

A second order polynomial regression was then performed on all 30 samples (R^2 = 0.9922), and this polynomial expression was fed into MATLAB's fmincon() function for minimisation. The lower and upper bounds were set to 0, 0, 0 and 1, 1, 1 respectively as they were in the normalised optimisation domain, and no constraints were set as none were identified. Initial values of 0.5, 0.5, 0.5 were chosen for x_1, x_2 and x_3 as suggested by MATLAB documentation [4]. The optimisation could then be presented in standard notation:

minimise $f(x_1, x_2, x_3) =$ [generated second order global polynomial]

by varying $0 \leq x_i \leq 1 \quad i = 1, 2, 3$

After converting the normalised minimised values to the global domain, the script predicted:

$x_1 = 0.34$
$x_2 = 0.0516$
$x_3 = 0.150$
$f(x_1, x_2, x_3) = -21.3664$

Finally, when input into the black-box a value of -21.5700 was returned. These values were then added to the design space, and it was now possible to home-in using refinement.

Design Space Refinement

The design space, now consisting of 31 points with a minimum validated against the black-box, was refined by considering samples solely below zero in the global domain. Figure 11 visualises the design space where a cyan bounding box represents the refinement window.

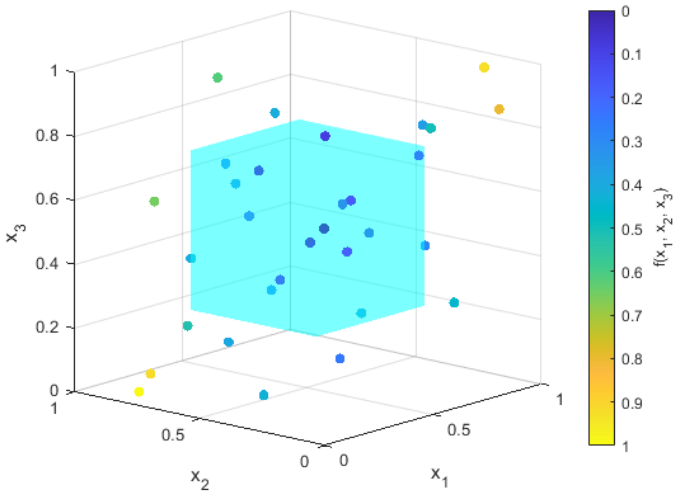


Figure X: Design space after first optimisation run. Cyan box indicates refinement window.

A major error was retroactively identified regarding the refinement technique; no validation of the refined data was performed. During the black-box session, the refined samples were fed directly back into the MultiPolyRegress() function, forgoing any validation. Due to the lower number of samples, this would have been an excellent use of LOOCV. Additionally, the fold generation process was erroneously tailored to specifically use 30 samples, ergo k-fold validation could not have been used. It was assumed that a second order polynomial would produce the lowest error for the refined design space. Despite this, a second order regression was performed on the refined design space now consisting of just 10 samples.

The second order regression on the refined design space and subsequent minimisation (with the starting points modified to match the minimised x_1, x_2 and x_3 values) yielded a predicted minimised objective function of -24.53, an improvement on the previous -21.5700. Inputting these variables into the black-box resulted in -27.3917. Now with four attempts left, further refinement of the design space was unfruitful as minimised input values hit their bounds. Some of these input values were tested and as can be predicted, did not improve the output. A second order regression of the available 11 samples resulted in a black-box output of -21.9454, even when adjusting the polynomial order. With one sample left, an "educated guess" was made by observing that x_1, x_2 and x_3 appeared to follow a periodic pattern at low outputs. Manually determining the input values resulted in the lowest black-box output of **-30.4052 with $x_1 = 1.27, x_2 = 0.0716$ and $x_3 = 0.210$.**

Reflection

The presented surrogate modelling workflow was effective in determining a refined design space; however, it was ultimately human intervention to identify trends in the sample data that resulted in the lowest black-box output.

The script was particularly apt at performing global regression using 6-fold validation to identify the polynomial order as the initial x_1, x_2 and $x_3$values produced a black-box output lower than anything previously seen during sampling, indicating that some level of optimisation was occurring. As discussed, the major issue with the script revolved around its lack of validation of the refined design space; the script was hard-coded to perform cross validation on only the initial 30 samples. Accuracy could significantly be improved in this by implementing LOOCV with the lower sample sizes of the refined design space. At no point when running the script was there any noticeable lag; the output was printed to the console almost instantly, suggesting the script was well optimised.

When conceptualising the process, it was not realised how important infilling was, and therefore as the design space became increasingly refined, it also became increasingly data scarce. It would have been ideal to reserve more of the 36 attempts for refinement as data became significantly more valuable. An iterative approach could be used rather than performing regression on as much data as possible. By taking, say, 15 sample samples, and then continually varying the optimiser's starting point with each optimisation run, the script would be able to perform much more intricate optimisations as samples are more densely packed around the minimisation area. This process has been presented in figure 12.

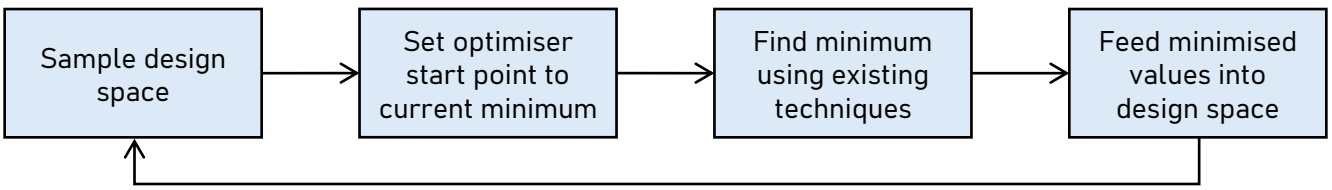


Figure 12: Alternative optimisation scheme using iterative refinement process.

To conclude, an evaluation of the optimisation process has been presented below.

Process Advantages

- 6-fold validation successfully identified lowest order with lowest error
- Use of MultiPolyRegress() allows adaptability to higher orders if required
- LHC allowed for excellent design space filling properties

Process Disadvantages

- Inefficient use of samples around refined design space
- No cross-validation of refined design space
- Input values tend to bounds if design space is inadequate

References (Optimisation Skills 2)

[1] MathWorks, 2024. *lhsdesign* [Online]. Available from: https://uk.mathworks.com/help/stats/lhsdesign.html [Accessed 14 November 2024]

[2] Cecan, A., 2024. *Multivariate Polynomial Regression* [Online]. Available from: https://uk.mathworks.com/matlabcentral/fileexchange/34918-multivariate-polynomial-regression [Accessed 15 November 2024]

[3] Hourieh, A., 2013. *Number of coefficients of a multivariable polynomial* [Online]. Available from: https://math.stackexchange.com/questions/380116/number-of-coefficients-of-a-multivariable-polynomial [Accessed 16 November 2024]

[4] MathWorks, 2024. *fmincon* [Online]. Available from: https://uk.mathworks.com/help/optim/ug/fmincon.html [Accessed 16 November 2024]

Optimising a Wind Turbine Array

Problem Overview

Optimisation Skills 3 explores the design of a linear wind turbine array to maximise energy efficiency and minimise costs. By optimising turbine radius, position, and height, the study addresses wake effects and power losses, aiming to identify configurations that balance performance with cost-effectiveness. The findings will provide insights into improving turbine layouts for greater reliability and energy output in real-world wind farms.

Turbine Cost Calculation

First, the radius data of the turbines are used to calculate the cost of manufacturing the wind turbines, which is dependent on the turbine radius and number of turbines with the same radius.

The position, radius and height data of the turbines are used to calculate the velocity deficit. The Weibull distribution is used for calculating the probability of initial wind speed, but is not representative for the wind speed after deficit caused by wind turbines upstream. Therefore, the Weibull distribution has been modified to calculate the probability of wind speed subjected to a deficit.

Figure 13 on the right shows the Weibull distribution with a deficit of 0 and 0.333.

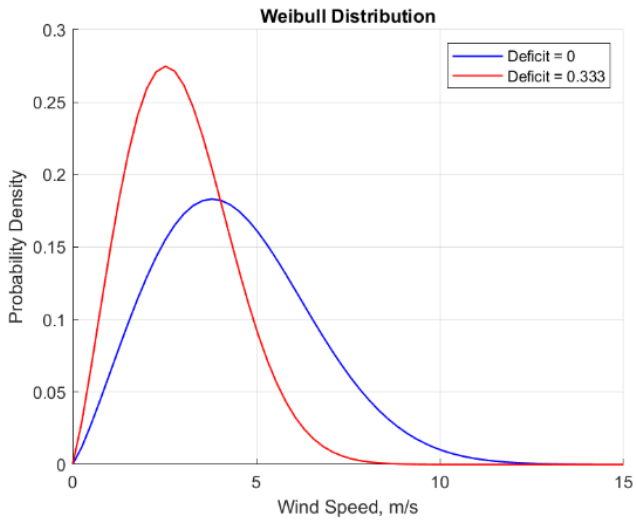


Figure 13: Weibull distribution

The modified Weibull distribution is combined with the power model to determine the annual energy production of each turbine by the integral of the combined function, which already accounted for deficit and probability variations. The annual energy production is converted to profit if the energy produced is sold back to the grid, which is at an average rate of 12p/kWh [1].

The objective function for the algorithm is defined as the net income of the turbine configuration after operation of 5 years.

$$Cost\ index = Turbine\ cost - AEP \times Conversion\ rate \times Years\ of\ operation \times\ Operational\ time$$

The objective function assumes a 90% operational time for maintenance, and the cost of shipping, installation and maintenance is constant for all turbine configurations.

Implementation Rules

A minimum distance between turbines was set to 150m to allow for rotation at the base of the turbine in the case of side winds, resulting in a minimum gap of 50m when two 50m radius turbines are rotated sideways.

The turbine data of distance, radius and height was tested at intervals of nearest meter, nearest 0.1 meter and nearest 0.1 meter, respectively. These intervals were set to the reasonable accuracy in a practical application.

Gradient Descent

Initially using a gradient-based algorithm was considered for this task due to its low compute time and high accuracy for problems with smooth, differentiable objective functions.

Unfortunately, these algorithms are very prone to getting stuck in local minima, along with the fact that the cost function may not be differentiable due to discontinuities in wake effects.

Genetic Algorithm

This is a nature-inspired algorithm that imitates evolution in a population. Each tested data point is equivalent to an individual specimen and undergoes simulated natural selection, gene crossovers with high fitness partners and random gene mutations in offspring.

This algorithm is very potent at finding local minima in functions that are non-linear and that may have multiple local minima. Specifically due to the ability of this algorithm to explore irregular solution spaces due to the varying mutational coefficient, the "Optimization Toolbox" from MathWorks was installed to use the `ga` function.

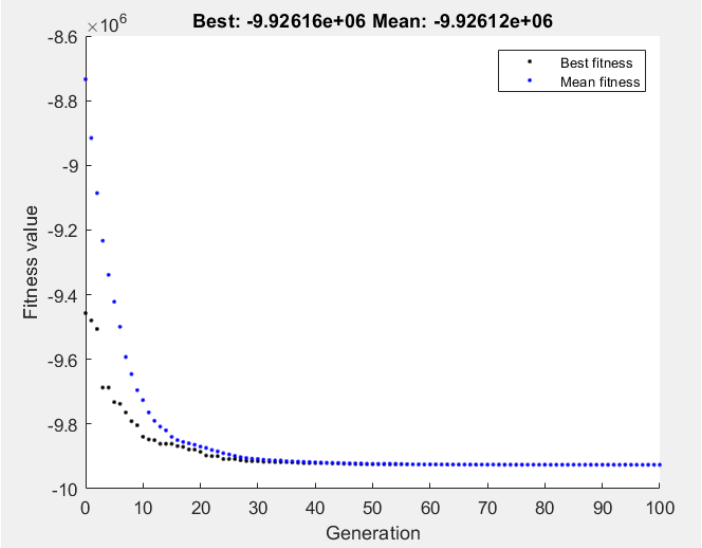


Figure 14: Genetic algorithm being used for 100 generations to determine lowest cost on an older version of Cost_Index_Cal() with 0.27 £/kWh rate.

Multi-Objective Evolutionary Algorithm

The usage of a MOEA, such as NSGA-II (Non-dominated Sorting Genetic Algorithm II) was tested due to its similarities with the standard genetic algorithm and ease of implementation (as the framework was already set up for integrating `ga` with the cost calculator, using `gamultiobj` would be simple). The key advantage this algorithm has is the ability to form a Pareto front that compares cost to build 8 turbines along with energy produced, allowing us to pick which balance of the 2 we would want.

This added functionality turned out to be irrelevant, due to our ability to transform the energy produced into cost by multiplying it by the conversion rate and years of operation.

By merging these 2 different dimensions into a singular cost dimension, the `ga` function was deemed ideal.

Figure 15 on the right shows the Pareto front between annual energy production and rotor cost. This was computed using `gamultiobj`.

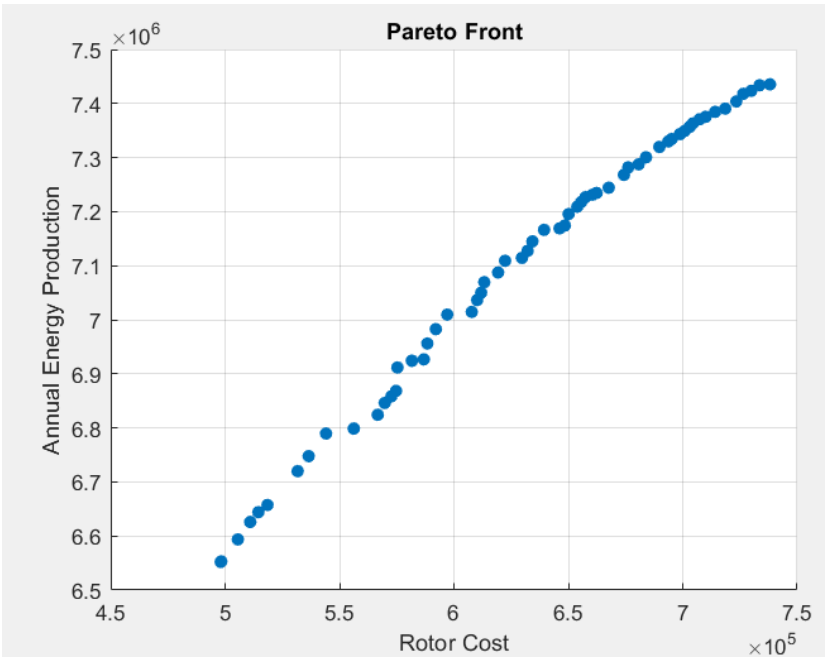


Figure 15: Pareto front showing annual energy consumption and rotor cost

Implementation

Initially, the Cost_Index_Cal() function was created with the ability to take positions (distance each turbine is from 0m), radii (of respective turbine rotors) and turbine height, run the variables through a deficit calculator, a rotor cost calculator, an annual energy production calculator and set some parameters such as simulation years and conversion rates to output a final cost of a turbine configuration with those input parameters.

5 years were simulated as an arbitrary number of years in operation, however changing the value to 10 or 20 years would lead to near identical optimal design parameter outputs as none of the variables were affected in a non-linear fashion by time.

A function called Optimise_Cost_Index() was created, detailing the bounds for the turbine radius and height (30m to 50m and 90m to 110m respectively) along with the number of turbines and said turbine bounds.

Figure 16 on the right shows the console outputting the final results.

```
>> Optimise_Cost_Index

Single objective optimization:
22 Variable(s)
7 Nonlinear inequality constraint(s)

Options:
CreationFcn:       @gacreationuniform
CrossoverFcn:      @crossoverscattered
SelectionFcn:      @selectionstochunif
MutationFcn:       @mutationadaptfeasible

                                Best       Max        Stall
Generation  Func-count         f(x)     Constraint  Generations
    1           9910   -3.82601e+06            0      0
    2          19620   -3.82622e+06            0      0
    3          29330   -3.82625e+06            0      0
    4          39040   -3.82625e+06            0      0
    5          48750   -3.82625e+06            0      1
Optimization terminated: average change in the fitness value less than optio
 and constraint violation is less than options.ConstraintTolerance.
Best Cost: 3826251.22
Optimal Position: [0 387.9 711.5 996.4 1279 1564 1850 2000]
Optimal Radius: [49 50 49 50 49 49 50 50]
Optimal Height: [102.2 98.75 98.64 97.57 106.6 104.6 109.5 107.2]
```

Figure 16

```
% Genetic Algorithm Options
options = optimoptions('ga', ...
'PopulationSize', 200, ...
'MaxGenerations', 50, ...
'CrossoverFraction', 0.9, ...
'MutationFcn', {@mutationadaptfeasible}, ...
'Display', 'iter', ...
'PlotFcn', @gaplotbestf); % Plot best fitness per generation
```

Figure 17

Once optimisation variable bounds had been declared, optimisation options were set. The adapting mutation function was an important addition to prevent the result converging on a local minima. Code shown in Figure 17.

Initial Results

The initial results were promising, however due to a lack of proper constraints, the “optimal” positions of the 8 turbines had distances that broke the 150m separation rule, being far too close for safe usage.

```
  100             4753      -9.926e+06      -9.926e+06        0
Optimization terminated: maximum number of generations exceeded.
Best Cost: -9926163.82
Optimal Position: [617.7 374.8 1273 268.5 625.5 381.9 1057 1444]
Optimal Radius: [50 50 50 50 50 50 50 50]
Optimal Height: [102.1 106.7 97.19 97.74 90.71 108.7 94.55 98.29]
```

Figure 18

As seen in Figure 18, some 4 turbines were less than 10m apart from each other. This would lead to a catastrophe if any of these turbines turned at even a slightly different angle to its pair.

```
Optimization terminated: average change in the fitness value less than options.
 and constraint violation is less than options.ConstraintTolerance.
Best Cost: -9924330.91
Optimal Position: [0 286.5 595.1 1011 1230 1452 1804 2000]
Optimal Radius: [50 50 50 50 50 50 50 50]
Optimal Height: [93.07 100.9 107.4 101.5 107.4 90.04 109.4 96.21]
```

Figure 19

Figure 19, left, was the first result after solving the constraints issue. It still suffered from a negative value for cost (profit generated by 5 yrs) and had an outdated 0.27 £/kWh rate.

Figure 20, right, fixed the previous issues and gave insights into the optimised f(x) in each generation. This insight was invaluable, clearly showing that the function was now mistakenly trying to reach a minimum profit output, generating the worst design possible.

```
                                Best       Max        Stall
Generation  Func-count         f(x)     Constraint  Generations
    1           2500   3.11482e+06            0      0
    2           4950   3.00683e+06            0      0
    3           9703   2.96345e+06            0      0
    4          14456   2.95517e+06            0      0
    5          16906   2.95517e+06            0      1
Optimization terminated: average change in the fitness value less than options.
 and constraint violation is less than options.ConstraintTolerance.
Best Cost: 2955167.94
Optimal Position: [0 329.5 541.3 822.5 1109 1537 1797 2000]
Optimal Radius: [30 30 30 30 30 30 30 30]
Optimal Height: [95.37 107.2 101.5 103.2 94.89 110 97.02 105.9]
```

Figure 20

Tweaks and Improvements

By returning the output values of the cost function to be negative, the genetic algorithm could continue to effectively find the global minima.

The simple workaround of just changing the negative output to a positive gave the desired outcome of design parameters with the maximal profit.

Initially, the genetic algorithm was being run for 500 generations, however the change in fitness function would be so minute by generation 5 that it would end. To ensure that local minima weren't being reached instead of global minima, the algorithm was run multiple times with increasing mutation rates.

They kept converging around the same values, which suggests that the algorithm is likely finding the global minima.

```
    1          9910  -1.46021e+06            0       0
    2         19620  -1.47005e+06            0       0
    3         29330   -1.4727e+06            0       0
    4         39040  -1.47404e+06            0       0
    5         48750  -1.47643e+06            0       0
Optimization terminated: average change in the fitness value less than options
 and constraint violation is less than options.ConstraintTolerance.
Best Cost: 1476427.66
Optimal Position: [0 371.5 682.1 1016 1221 1541 1850 2000]
Optimal Radius: [40 30 30 30 30 30 30 34]
Optimal Height: [105.6 109.6 101.9 90.03 110 105.8 110 90.11]
```

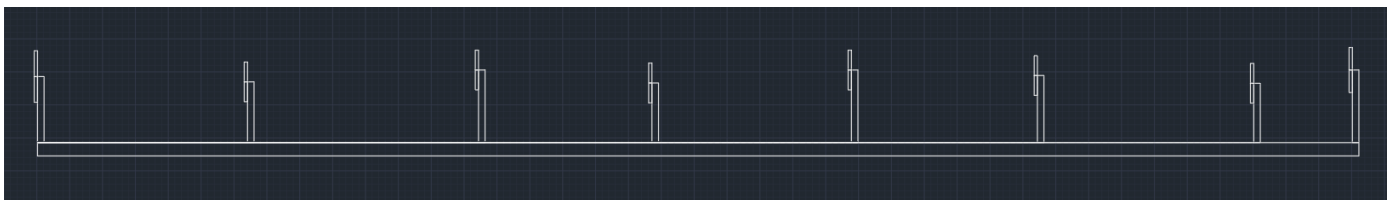
```
    1          9910  -1.45915e+06            0       0
    2         19620   -1.4676e+06            0       0
    3         29330  -1.47096e+06            0       0
    4         39040  -1.47347e+06            0       0
    5         48750  -1.47549e+06            0       0
Optimization terminated: average change in the fitness value less than options
 and constraint violation is less than options.ConstraintTolerance.
Best Cost: 1475485.34
Optimal Position: [0 319.4 670.9 934.7 1238 1521 1850 2000]
Optimal Radius: [39 30 30 30 30 30 30 34]
Optimal Height: [100.2 92.02 110 90.04 110 101.8 90.02 110]
```

Figure 21: Genetic algorithm re-run, resulting in near-identical cost functions and optimal positions.

Final Results

The final result, as shown in Figure 21, indicated that with the turbine radius of 30-39, turbine height fluctuating massively between 90 and 110 and turbines configured with more density near the back, the maximum profit can be generated in a 5-year span, being roughly £1.465 million.

The proposed turbine configuration is visualised below, to scale.



As can be seen in Figure 22 above, there appears to be an increasing density near the back, the last 2 turbines being exactly 150m apart (which would no doubt be closer if it weren't for the constraints). The rotor radius stays at the maximum, intuitively making sense as this allows for more wind energy to be collected per turbine. The heights fluctuate greatly, often going from 90m straight to 110m to avoid the wake pattern of the turbine in front.

By understanding these patterns, we manually entered a radius of 30 for all the radiuses, fluctuating heights perfectly going from 90m to 110m, and with the turbines placed the same way as simulated. We achieved an even greater profit of £1.51 million. Figure 23 below shows the console.

```
Height = [90 110 90 110 90 110 90 110];
Radius = [30 30 30 30 30 30 30 30];
Position = [0 387.9 711.5 996.4 1279 1564 1850 2000];
```

```
Cost_Index =

   1.5164e+06
```

References (Optimisation Skills 3)

[1] The Renewable Energy Hub, 2024. *How Much is a Wind Turbine Likely to Make me and Over What Period? UK* [Online]. Available from: https://www.renewableenergyhub.co.uk/main/wind-turbines/how-much-is-a-wind-turbine-likely-to-make-me-and-over-what-period [Accessed 20 November 2024]