Candidate Number: 21777

Introduction

Fractals are infinitely complex shapes that show a level of self-similarity at many different scales and have a non-integer dimension. This dimension is a crucial feature of a fractal as it provides information about how the pattern changes with scale and about its space-filling capacity. The increasing importance of accurate measurements of fractal dimensions is due to the discovery of the huge spectrum of natural objects that can be characterised using a fractal dimension analysis. This investigation uses the box-counting method which involves covering the fractal with a grid and counting the number of boxes that are occupied. By using multiple different box sizes the dimension of the object can be obtained by finding the gradient of the linear portion of a log(N(1/n)) vs log(1/n) graph, where n is the number of boxes along each axis and N is the number of boxes occupied at a given value of n.

Frac 1 Code

Frac 3 Code

Box-Count Algorithm

The box-count algorithm used has been slightly adjusted so that the grid used to perform the box-count can be translated in both the x and y directions. The function takes in two more arguments, sfx and sfy, which represent this random shift. The range of x coordinates is calculated and is set as 100%, the grid is then extended by 5% either side of this range so that the gird now covers -5% to 105% of the original grid. The value sfx dictates a shift between -1% and -5% of the original range while also maintaining a constant grid range of 106%. This process is then repeated for the y direction of the grid.

```
> boxcount := proc(data, N, sfx, sfy)
  local n, xmax, xmin, xrange, ymax, ymin, yrange, dx, dy, i, j, ix, iy, sum, res :
   # no. input points

   n := ArrayNumElems(data) / 2 :
   # determine ranges that span the input data

  xmax := max(seq(data[i, 1], i = 1 ..n)) : xmin := min(seq(data[i, 1], i = 1 ..n)) :
  ymax := max(seq(data[i, 2], i = 1 ..n)) : ymin := min(seq(data[i, 2], i = 1 ..n)) :

  xrange := xmax - xmin :
  xmin := xmin - 0.05· xrange + sfx·0.01·xrange : xmax := xmax + 0.05· xrange - (0.04 - 0.01·sfx)
      ·xrange :
  xrange := 1.06· xrange :

  yrange := ymax - ymin :
  ymin := ymin - 0.05· yrange + sfy·0.01·yrange : ymax := ymax + 0.05· yrange - (0.04 - 0.01·sfy)
      ·yrange :
  yrange := 1.06· yrange :

  # define grid spacing to give an N by N grid.

   dx := xrange/N : dy := yrange/N :

  # define an array to hold the box occupancies

  res := Array(0 ..N-1, 0 ..N-1, 0) :

  # step through data points, identifying which grid cell it belongs in.
  # trunc returns the integer part of a floating point number.

  for i from 1 to n do
    ix := trunc((data[i, 1]-xmin)/dx) : iy := trunc((data[i, 2]-ymin)/dy) :
    res[ix, iy] := 1 :
  end do:

  #  finally, having determined the cells in which all the points lie,
  #  find out & return the total number of cells that are occupied.

  add(add(res[i, j], i = 0 ..N-1), j = 0 ..N-1);
  end proc:
```

Quantisation Error

The box count method relies on finding the minimum number of boxes, of a given size, to cover the image. An arbitrary grid does not guarantee that this minimum number will be found. This over counting leads to a quantisation error which needs to be minimised and calculated, this is achieved by running several box counts for each box size with a translation in x and y of the grid. These translation factors are randomly generated so that any periodicity in the image will not affect the calculations. The minimum box count value is then selected and the difference between this minimum value and the box count value for an untranslated grid gives the quantisation error. The functionality of this procedure could be developed by including several rotational grid translations. The issue with this method is that it increases the number of box counting procedures performed and therefore also increases the computational cost. Both are increased by a factor NOG (the number of grid translations performed at each box size) and so NOG must remain relatively small.

The QEReduction procedure has four arguments; the first is NOG, the number of grid translations performed per value of n, the next three indicate the range of n values (lowest and highest) and by what spacing will be used to increment between the two.

```
> QEReduction := proc(NOG, lowBoxn, highBoxn, spacing)
      local gridShiftFactorX, gridShiftFactorY, n, i, tempBox, minBoxCount, noShiftValue;
       global res, errStr, xv, yv;

      #set the data and error lists as empty and set the minimum box count as the highest possible
      #number of boxes that can be counted
      res := [ ] : errStr := [ ] : minBoxCount := highBoxn^2 :

      #cycle through the values of n selected by the user
      for n from lowBoxn by spacing to highBoxn do
            for i from 1 to NOG do #NOG is the number of grid transformations selected by the user

                  #produce and random shift factor for both the x and y directions
                  gridShiftFactorX := rand(0.0 ..1.0);
                  gridShiftFactorY := rand(0.0 ..1.0);

                  #perform a box count with the transformed grid
                  tempBox := boxcount(pts, n, gridShiftFactorX( ), gridShiftFactorY( ));

               #if statement to find the lowest number of boxes counted for each n
                  if tempBox < minBoxCount then
                        minBoxCount := tempBox;
                  end if;

            end do;

                  #perform a box count for an untransformed central grid
                  noShiftValue := boxcount(pts, n, 0.5, 0.5);

                  #check to see if the untransformed box count is the lowest
                  if noShiftValue < minBoxCount then
                        minBoxCount := noShiftValue;
                  end if;

                  #populate list with 1/n and the lowest box count found
                  res := [op(res), [1.0/n, minBoxCount]];
                  #populate list with the quantisation error for each value of n
                  errStr := [op(errStr), (noShiftValue-minBoxCount)];
                  #reset the minimum boc counter to the maximum possible number of boxes
                  minBoxCount := highBoxn^2;

       end do:
      #split res into x and y components for calculations
      xv := [seq(log(res[i][1]), i = 1 ..nops(res))] :
      yv := [seq(log(res[i][2]), i = 1 ..nops(res))] :

      #calculate the error in the gradient due to the quantisation error
      errorCalc(res, errStr);

   end proc:
```

The QEReduction procedure produces a list of the absolute quantisation error for each value of n, this is used as one of the arguments for the errorCalc procedure where the other is the original data. The absolute errors are then converted to the necessary errors needed for a log log plot and these are then used to calculate the maximum and minimum gradients, finally the error in the gradient is calculated.

```
> errorCalc := proc(data, absoluteErr)
      local i, y, dy, lim, minGrad, maxGrad;
      global err, gradError;

      #set the error list as empty and assign the length of the data inputted to lim
      err := [ ] : lim := nops(data) :

      #this loop calculates the error for each point assuming that a log10 has been taken
      for i from 1 to lim do

            y := data[i][2];
            dy := absoluteErr[i];
            err := [op(err), (0.434*dy/y)];

      end do;

      #calculate the maximum and minimum gradients due to the errors
      minGrad := ((log10(data[lim][2])-err[lim])-(log10(data[1][2]) + err[1]))/(log10(data[1][1])
      - log10(data[lim][1]));
      maxGrad := ((log10(data[lim][2]) + err[lim])-(log10(data[1][2])-err[1]))/(log10(data[1][1])
      - log10(data[lim][1]));

      #calculate the resulting error in the gradient
      gradError := (maxGrad-minGrad)*(1/2);
   end proc:
```

Pearson Correlation Coefficient

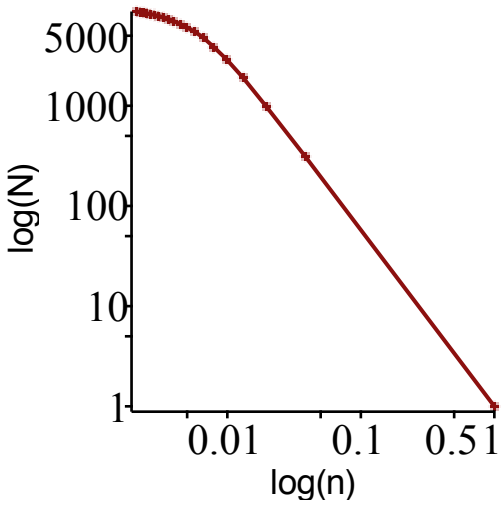
The Pearson correlation coefficient (PCC) is a measure of the linear correlation between two variables, it has a value between -1 and 1 where 0 is no linear correlation, 1 is a complete positive correlation and -1 is a complete negative correlation. This value can be used to find the most linear region in a set of data. This is useful for the box-count method as the fractal dimension is obtained by finding the gradient of the linear portion of the log(N(1/n)) vs log (1/n) graph. The procedure has two arguments; the first is a list that contains the coordinates of the points which you would like to analyse and the second is the minimum number of points that would be acceptable to create the line. It calculates the PCC for every combination of points, where the number of points is greater than the set minimum, and saves the combination with the highest absolute PCC. When calculating the fractal dimension this procedure does not guarantee an increase in accuracy, it merely finds the most linearly related combination of points. For this reason the procedure is only used when the curvature at large values of n can visibly be observed.

```
> PearsCoeff := proc(data, minNPoints)
      local xv, yv, i, j, test, pccN, pccD, pccTemp, L, avgX, avgY;
      global S1, S2, finalSol, pccFinal;

      pccFinal := 0.0; #initially set the Pearson correlation coefficient to be 0

      #the two for loops cycle through every possible combination of points
      for i from 1 to nops(data) do
            for j from 1 to nops(data) do
                  #only calculate the coefficient if the range contains enough points, this is set by the user
                  if j-i ≥ minNPoints then

                        #create x and y lists for the data between the two selected points
                        xv := [seq(log(data[k][1]), k = i .. j)];
                        yv := [seq(log(data[k][2]), k = i .. j)];

                        L := nops(xv); #the lenght of xv (equal to the length of yv)

                        #find the average values of x and y
                        avgX := (add(xv[i], i = 1 .. L))/L;
                        avgY := (add(yv[i], i = 1 .. L))/L;

      #calculate the numerator and denominator of the Pearson coefficient and then perform the division
                        pccN := (add((xv[i]-avgX)*(yv[i]-avgY), i = 1 ..L));
```

```
                    pccD := (add((xv[i]-avgX)^2, i=1..L)*add((yv[i]-avgY)^2, i=1..L))^(1/2);
                    pccTemp := evalf(pccN)/evalf(pccD);

                    #calculate the line of best fit for these points
                    test := Fit(m*x + const, xv, yv, x, output = [leastsquaresfunction,
    standarderrors]);

                    #maximise the Pearson coefficient to obtain the best straight line
                    if abs(pccTemp) > abs(pccFinal) then

                        pccFinal := pccTemp; #save the best Pearson coefficent value
                        finalSol := test; #save the best fit line for the optimum points
                        S1 := i;
    #save the start and end point of the optimum line to be viewed by the user
                        S2 := j;

                    end if;
                end if;
            end do;
        end do;
    end proc:
```

The following test shows the log(N(1/n)) vs log(1/n) graph for frac1 (approximated with 10,000 data points) for values of n between 25 and 500 with a box count performed every 25. The graph shows a significant level of curvature for large values of n; as it is only the linear section that provides information on the fractal dimension the PearsCoeff procedure can be used to find this region. A minimum of 6 points were chosen to form this linear region, S1 and S2 show that the region obtained by PearsCoeff is between points 1 and 7 and pccFinal shows the PCC in this region. originalSol shows the solution that would have been obtained without applying PearsCoeff and finalSol shows the solution in the region isolated by the procedure.

```
> frac1(10000):
> res := [seq([1.0/n, boxcount(pts, n, 0.5, 0.5)], n=1..500, 25)]:
> l := loglogplot(res): p := loglogplot(res, style=point, size=[200, 200]):
> display(l, p, labels = ["log(n)", "log(N)"], labeldirections = [horizontal, vertical], labelfont = ["Helvetica",
      10])
```




```
> PearsCoeff(res, 6)
> S1; S2
```

$$1$$
$$7 \qquad \textbf{(6.1)}$$

```
> pccFinal
```

$$-0.9992840247 \qquad \textbf{(6.2)}$$

```
> xv := [seq(log(res[i][1]), i=1..nops(res))]: yv := [seq(log(res[i][2]), i=1..nops(res))]:

> originalSol := Fit(m·x + const, xv, yv, x, output=[leastsquaresfunction, standarderrors]);
```

$$originalSol := \left[0.821275972778351 - 1.43637948194782\,x,\ \left[\ 0.365251122795240 \quad 0.0699972505663444\ \right]\right] \qquad \textbf{(6.3)}$$

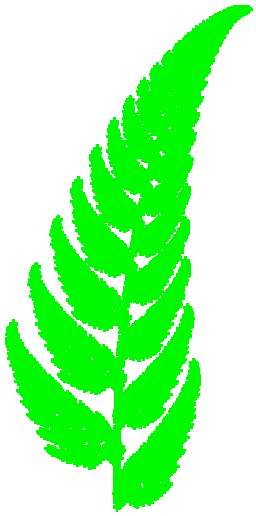
```
> finalSol
```

$$\left[0.0762284787392280 - 1.70801349731388\,x,\ \left[\begin{array}{cc} 0.117063249156014 & 0.0289203467974629 \end{array}\right]\right] \quad \textbf{(6.4)}$$

Analysis of Frac 1

In this model the fractals being analysed are approximated as a controllable number of points. The problem with this approximation is that at large values of n the boxes can occupy the space between points which therefore produces an underestimation in the number of boxes occupied. A system of linear interpolation, as is used in many box counting methods of natural fractals, such as root systems, could remove this uncertainty. Unfortunately, it was not possible to implement this during the investigation and so alternate methods were required to minimise this uncertainty. Our algorithms allow this to be controlled in three different ways; firstly, the range of n can be controlled such that the largest values produce grid spacing that is larger than the distance between points. This control can also be used to increase the minimum size of n because the noise at small values results in a saturation of the curve. Secondly, as previously discussed, if the graph displays visible curvature the PearsCoeff procedure can be applied to the data points to find the most linear region. Finally, the number of points used to approximate the fractal can be increased, however this also increases the computational cost of every box-count procedure called. It was found that an increase in the order of magnitude of the number of points used to define the fractal also increased the computational time by an order of magnitude (1,000 points, T = 0.006s; 10,000, T = 0.056s; 100, 000, T = 0.520; 1,000,000, T = 6.968s). Using the Sierpinski Triangle (with 12 iterations), which has a known fractal dimension of 1.585, it was also found that an increase, by an order of magnitude, in the number of points resulted in an increase of one significant figure in the precision of the fractal dimension (1,000 points, FD = 0.365; 10,000, FD = 1.380; 100,000, FD = 1.540; 1,000,000, FD = 1.581). This highlighted the importance of achieving the correct balance between the computational cost and the desired precision of the calculated fractal dimension. The computational cost required to calculate each fractal dimension depended on three factors; the first, as mentioned above, is the number of points used to define the fractal. This affects the time taken to complete each call to the box-count procedure itself, the second and third factors affect the number of times that the box-count procedure is called. The second is the number of grid translations (NOG) used per value of n to try and reduce the quantisation error. The third is the number of points used on the logN(1/n) vs log(1/n) graph, this is determined by the range of n values and the spacing between each value.

```
> frac1(10000);
> plots[listplot](pts, style=point, symbol=cross, symbolsize=1, color=green, scaling=constrained, axes
      =none, size=[200, 200])
```



The above image shows frac1 defined using 10,000 points. Testing on a combination of the Koch curve and the Sierpinski triangle showed that for a fractal defined by 100,000 points a range in n of 25 to 300 produced the optimum results without a need for the PearsCoeff procedure. For a fractal defined by 1,000,000 points the optimum range of n was found to be 25 to 500. The first test shows the result for n between 25 and 300 with a spacing of 25 between each point,0 grid translations and frac1 being defined by 100,000 points.

```
> frac1(100000) :
> QEReduction(0, 25, 300, 25) : test1 := Fit(m·x + const, xv, yv, x, output=[leastsquaresfunction,
      standarderrors]) :
> -(diff(test1, x))[1]; test1[2][2];
```

$$1.70676707701856$$
$$0.00946289042804671 \quad \textbf{(7.1)}$$

This test produced a fractal dimension of 1.707±0.009 with a computational time of 10.5 seconds where the error is the standard error due to the least squares analysis. The second test is identical to the first however for every point there is now 5 grid translations, this allows the quantisation error to be calculated.

```
> QEReduction(5, 25, 300, 25) : test2 := Fit(m·x + const, xv, yv, x, output=[leastsquaresfunction,
      standarderrors]) :
> -(diff(test2, x))[1]; test2[2][2]; gradError;
```

1.70736222835777

0.00924261820313488

0.0002375876698 **(7.2)**

This test produced a fractal dimension of 1.707±0.009 with a computational time of 62.0 seconds and shows that the error due to the least squares analysis is an order of magnitude larger than the quantisation error calculated. The way that the error in the gradient is calculated, due to the quantisation error, relies solely on the error in the first and last point. This simplification results in the calculated values of the error due to quantisation to be underestimated. For the lowest value of n the absolute quantisation error is likely to be much smaller and so heavily relying on this error lowers the overall value. To provide a more useful result the algorithm needs to be improved to consider the absolute errors of all the points along the line however, this system still provides information about the order of magnitude of the quantisation error and by selecting the lowest values of N(1/n) it increases the accuracy of the fractal dimension obtained. The third test shows the impact of increasing the number of grid translations per point to see how this impacts the quantisation error.

```
> QEReduction(10, 25, 300, 25) : test3 := Fit(m·x + const, xv, yv, x, output = [leastsquaresfunction,
      standarderrors]) :
> -(diff(test3, x))[1]; test3[2][2]; gradError;
```

1.70810057114282

0.00916338416539536

0.0007930521182 **(7.3)**

This test produced a fractal dimension of 1.708±0.009 with a computational time of 122.0 seconds and showed a slight increase in the quantisation error however it is still an order of magnitude smaller than the error produced from the least squares analysis. The next variable to test is how the number of points along the line affects the least squares analysis error, to test this the spacing is reduced from 25 down to 5.

```
> QEReduction(5, 25, 300, 5) : test4 := Fit(m·x + const, xv, yv, x, output = [leastsquaresfunction,
      standarderrors]) :
> -(diff(test4, x))[1]; test4[2][2]; gradError;
```

1.69800767656909

0.00359568190214071

0.0001187587545 **(7.4)**

This test produced a fractal dimension of 1.698±0.004 with a computational time of 296.0 seconds. This highlights the expected result that increasing the number of points along the line will decrease the error due to the least squares analysis. The issue is that to achieve a reduction in the error of approximately 1/2 it required a computational cost 5 times larger. The final test is on the affect that the number of points used to define the fractal has on the fractal dimension and each of the associated errors, the number of points used to define frac1 is increased from 100,000 to 500,000.

```
> frac1(500000) :
> QEReduction(5, 25, 300, 25) : test5 := Fit(m·x + const, xv, yv, x, output = [leastsquaresfunction,
      standarderrors]) :
> -(diff(test5, x))[1]; test5[2][2]; gradError;
```

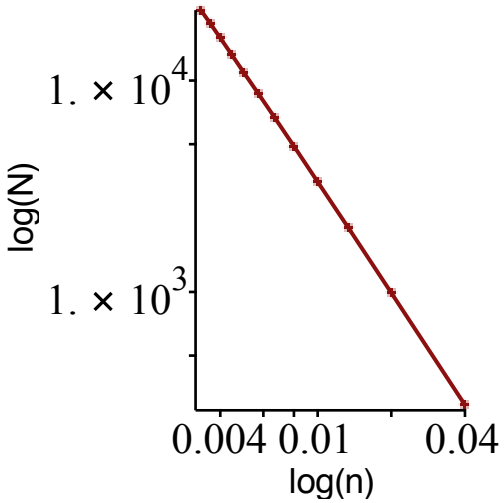
1.73960893132918

0.00403580395183004

0.002899923892 **(7.5)**

This test produced a fractal dimension of 1.740±0.004 with a computational time of 304.0 seconds. Again this test produced a reduction in the least squares analysis error of approximately 1/2 while also producing a 5 times increase in the computational cost. This test also raised the quantisation error by over an order of magnitude. This increase can be explained by the fact that increasing the number of points will increase the number of boxes being counted for large values of n and therefore the difference in the number of boxes for different grid orientations is likely to be larger as well. The results from these tests show that a balance needs to be found between all the contributing factors to aim to minimise the error, increase the accuracy while also maintaining a reasonable computational cost. An increase in the number of grid translations did not influence the fractal dimension obtained or the resulting error however to obtain the lowest number of boxes counted for each value of n a reasonable number of translations needs to be performed so this is set to 5. The resulting increase in the computational cost of greatly increasing the number of points on the graph and the number of points used to define the fractal to obtain relatively small decreases in error means that a large change in each is not necessary. Despite this an increase in the number of points used to define the fractal has proven to be beneficial to obtaining more precise results and so an increase from the initial 100,000 points to 250,000 will be implemented. The range of n will remain at 25 to 300 with a spacing of 25.

```
> frac1(250000):
> QEReduction(5, 25, 300, 25):
> l := loglogplot(res): p := loglogplot(res, style=point, size=[200, 200]):
> display(l, p, labels = ["log(n)", "log(N)"], labeldirections = [horizontal, vertical], labelfont = ["Helvetica",
      10])
```




```
> solution1 := Fit(m·x + const, xv, yv, x, output = [leastsquaresfunction, standarderrors]):
> -(diff(solution1, x))[1]; solution1[2][2]; gradError;
```

1.73121513845173

0.00532688331069644

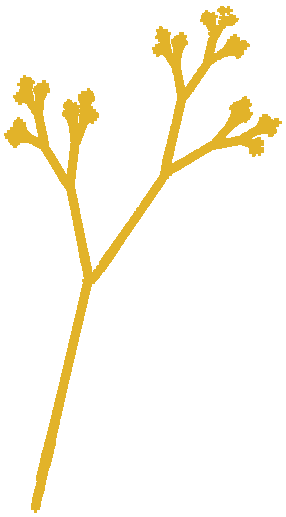
0.003441152460 **(7.6)**

This final solution produced a fractal dimension of 1.731 ± 0.005 in a computational time of 154.5 seconds. The graph highlights that a correct range of n values has been chosen and therefore the PearsCoeff procedure was not required.

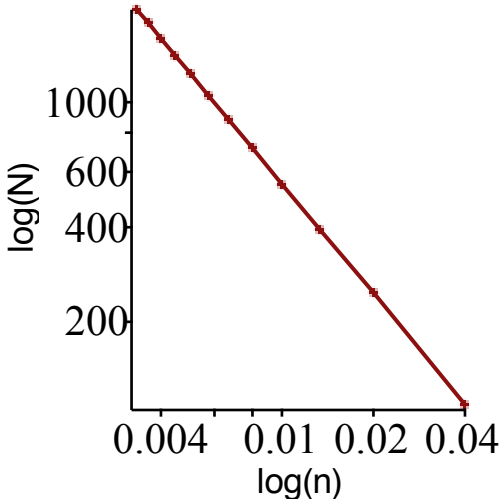
Analysis of Frac 3

The image below shows frac 3 defined using 10,000 points. Using the information gained from running tests on the Koch curve, Sierpinski triangle and the tests shown above the fractal dimension of frac 3 will be determined by using 250,000 points to define it, the values of n will range from 25 to 300 in spaces of 25 and there will be 5 grid transformations performed per value of n.

```
> frac3(10000):
> plots[listplot](pts, style=point, symbol=cross, symbolsize=1, color="Goldenrod", scaling
      =constrained, axes=none, size=[200, 200])
```



```
> frac3(250000):
> QEReduction(5, 25, 300, 25):
> l := loglogplot(res): p := loglogplot(res, style=point, size=[200, 200]):
> display(l, p, labels = ["log(n)", "log(N)"], labeldirections = [horizontal, vertical], labelfont = ["Helvetica",
      10])
```




```
> solution2 := Fit(m·x + const, xv, yv, x, output = [leastsquaresfunction, standarderrors]):
> -(diff(solution2, x))[1]; solution2[2][2]; gradError;
```

1.16397269417452

0.00254315392603072

0.005311928740 **(8.1)**

This final solution produced a fractal dimension of 1.164 ± 0.005 in a computational time of 140.0 seconds. Once again the graph shows that the range of n values chosen is suitable for this particular fractal without requiring the PearsCoeff procedure. The only difference with this particular fractal is that the quantisation error is larger than the error from the least squares analysis and hence the quantisation error is the one quoted.

Conclusion

The box count method developed to find the fractal dimension produced results to four significant figures within 3 minutes of computational time. The values used to calculate the fractal dimension of frac1 and frac3 have shown to be a good balance between computational effort and the accuracy and precision of the results. These values, however, are not guaranteed to work as well for every fractal. For example, fractals that have a higher level of detail would require more points to define it accurately and so to maintain the same level of computational time other factors would need to be reduced. Although, coupled with the PearsCoeff procedure these values represent a strong foundation.

This method could be improved by implementing a more advanced version of the algorithm used to minimise and calculate the quantisation error. This could involve using grid rotations as well as translations and would need to implement a more rigorous way of calculating the minimum and maximum gradients so that the errors of all the points are considered. Creating a form of linear interpolation to connect adjacent points would also improve the program as it would allow higher values of n and therefore a wider range of n values.