2D Inverse Distance Weighting Interpolation Model

Devlin Papworth

1 Introduction

This project makes a continuous data set (2D Block Model) by interpolating '*point*' data (Drill Holes) using Inverse Distance^2 Weighting (IDW). The user can define the dimensions of the block model and choose to import an Excel file, make their own data within the code, or use a randomly generated data set (for testing the code). The IDW will then automatically run through the data set and writes out the output block model to an excel file (Figure 1). There is also an option to run a Python script to generate clean graphs (Figure 2); however, this is not my code and is only used to assign a colour value to the raw numerical values stored in each square of IDW_output.csv.

2 User Instructions

Initially, the user is prompted with various setup questions to define parameters, including block model size, power value, number of drill holes, and the preferred method for importing drill hole data. Each user input is checked to be valid before continuing to the next step. The calculation is performed automatically writing out the data to a csv file. Once complete the user can choose to run a python script to create a graphic block model.

3 The Code

The code consists of 4 main functionality files: IDW_functions, ask_usr, Error_handler, data_handle. These files respectively: compute the main maths, ask the user for inputs, handles any errors that occur, and monitors and controls the data being taken in for the calculations. The remaining files contain small non core functionality code.

3.1 Data Storage

The drill hole data is read in from its respective stream and stored in a 2D array called DH_array within the code, for example Table 1 would be stored as:

```
DHarray = {{32,42,0.29},{52,98,0.12},{13,7,0.56},{..,..,..}}
```

X	Y	Grade Value
32	42	0.29 %
52	98	0.12 %
13	7	0.56 %
..	..	.. %

Table 1: An example data set being taken in.

where the number of rows of this 2D array is defined by the number of Drill Holes. This array is stored as pointer to pointers (below).

```
double** DH_array = new double* [num_DH];
for (int c = 0; c < num_DH; c++)
    {
        DH_array[c] = new double[3] {0.0, 0.0, 0.0}; // populates empty drill hole array
    }
```

3.2 Error Handling

Steps are taken to account for user error in entering the values. For example, two key functions that are implemented repeatedly are:

```
bool valid_double(string str);
bool valid_int(string str);
```

These take the user inputs as a string and then for each element of that string checks if it is a valid number input or if has more than one decimal place.[1]. Before running the IDW calculation three checks of the user's data are completed.

```
double** valid_DH_values(double** DH, int num);
double** repeat_DH_xy(double** DH, int& num);
double** grid_greater_than_DH(int& max_x, int& max_y, double** DH, int num);
```

They check for valid integers for coordinates of drill holes, valid doubles between 1 and 0 for the grade values, any repeated drill holes, and also ensures that the output block model includes all drill holes. If the functions find an error in the data they will check with the user for how they want to correct it, modifying the memory at DH_array accordingly. In the case of repeated Drill hole(x,y) coordinates the repeated drill hole is removed. This is done by copying over one drill hole at a time from the old DH_array to new_DH array and checking if the i^{th} Drill hole(x,y) being copied is already inside the new array, if so it is not included.

3.3 The IDW Formula

The base functionality of the code is looping over the IDW formula (below) for every sample grid square of the user defined block model.

$$\text{Sample}(x,y) = \frac{\sum_{i=1}^{N} \frac{\text{Drill Hole Value}}{distance_i^{\text{power}}}}{\sum_{i=1}^{N} \frac{1}{distance_i^{\text{power}}}} \quad \text{where} \quad N \text{ is the number of drill holes.}$$

For a single given sample(x,y) the code loops through all drill hole points. Within this loop, the distance between the target sample square and the i^{th} drill hole(x,y) is calculated and applied to the IDW formula. The output of the numerator and denominator is stored as top and bottom, respectively (see below). Top and Bottom are summed to each time a new drill hole is calculated for that same sample(x,y). Once all drill holes are taken into account for one sample(x,y) the code exits the inner loop and divides top by bottom, outputting the value into IDW_output.csv using the sample(x,y) co-ordinates. This outer loop is repeated for all squares in the block model. When the sample(x,y) is the same as the i^{th} Drill hole(x,y) then the IDW is skipped as that output(x,y) already contains a measured value.

```
    double temp = pow(target_x - sample_x, 2) + pow(target_y - sample_y, 2);
    double sqrt_distance = sqrt(temp);
    top = top + (t_val / (pow(sqrt_distance, Pow))); // sums for each iteration of the inner loop.
    bottom = bottom + (1.0 / pow(sqrt_distance, Pow));
```

When running large data sets the code would run, without any update of progress. In response to this I added a progress percentage which replaces itself when updating. This is achieved by taking the i^{th} loop increment (which iterates through all block model squares), dividing it by the total number of grid squares, and rounding the result to display the progress as a percentage.

```
cout << "\rProgress: " << trunc(progress + 0.5) << "%" << flush;
```

3.4 Limitations and Room for Development

This model assumes:

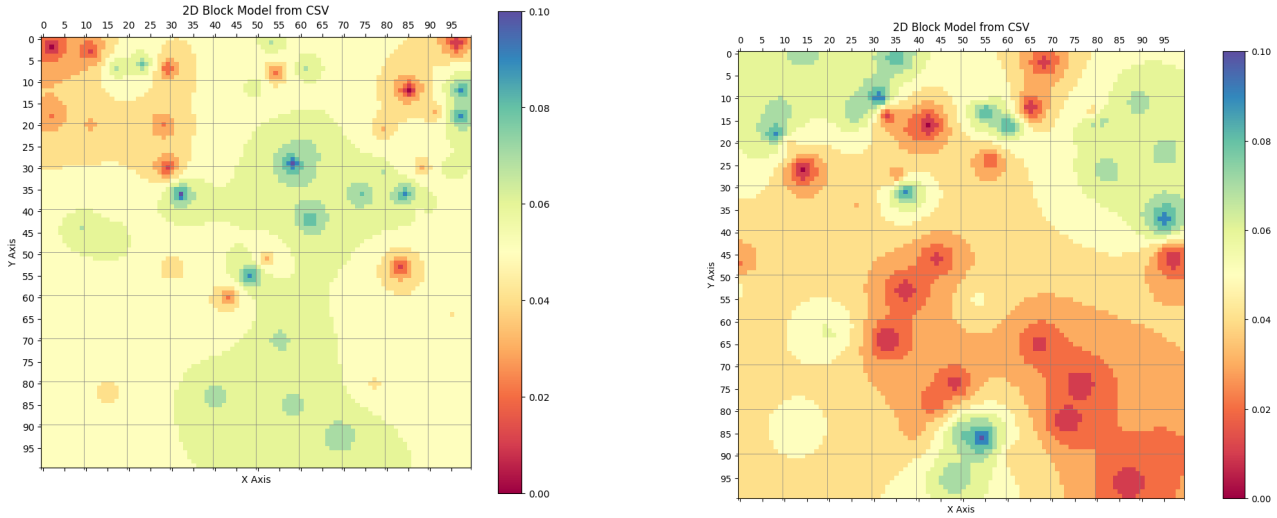
- isotropic influence and does not account for structural trends in the data, such as bedding or fault planes.
- that a drill hole value represents an entire block in the model, without considering the actual width of the drill hole or variations within the block.
- The current implementation only interpolates in two dimensions (X and Y) and does not fully account for depth variations in a 3D space.

[1]* Acknowledgement to Stephen Neethling for their guidance in developing this.

4 Appendix

	A	B	C	D	E	F	G	H	I	J
1	0.06	0.04	0.07	0.07	0.07	0.07	0.07	0.06	0.06	0.06
2	0.06	0.06	0.06	0.07	0.07	0.08	0.07	0.07	0.09	0.06
3	0.06	0.06	0.06	0.09	0.07	0.09	0.06	0.06	0.04	0.06
4	0.05	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05	0.05
5	0.05	0.05	0.05	0.05	0.04	0	0.04	0.04	0.05	0.05
6	0.05	0.06	0.05	0.04	0	0.03	0.03	0.03	0.04	0.04
7	0.1	0.06	0.05	0.04	0.04	0.04	0.03	0	0.03	0.04
8	0.08	0.06	0.05	0.05	0.05	0.04	0.03	0.03	0.03	0.04
9	0.06	0.06	0.06	0.06	0.07	0.04	0.01	0.03	0.04	0.04
10	0.06	0.06	0.05	0.05	0.05	0.04	0.03	0.04	0.04	0.04

Figure 1: Example of the output excel file that the code would produce.



(a) Output with low power value (1.5).

(b) Output with high power value (1.9).

Figure 2: Comparison of two Block Diagrams with different power values. 100x100 grid with 50 random drill holes.