M20 Homework 1

Colin Skinner - 505795313

2022/07/01

1 Oblate Spheroid Calculations

1.1 Introduction

The goal in this problem is to calculate the surface area of an oblate spheroid with user-specified equatorial and axial polar radii. The results will be approximated using a function value and calculated using the exact function. They will be printed, and discrepancies between the approximation and calculation will be discussed.

1.2 Model and Theory

An oblate spheroid is a sphere with different equatorial and polar radii. Thus, a function that determines a specific quality of such a shape will take into account both values. The applicable function for this problem is:

$$A(r_1, r_2) = 2\pi \left(r_1^2 + \frac{r_2^2}{sin(\gamma)} ln \left(\frac{cos(\gamma)}{1 - sin(\gamma)} \right) \right) \tag{1}$$

$$A(r_1, r_2) \approx 4\pi \left(\frac{(r_1 + r_2)}{2} \right)^2 \tag{2}$$

Where

- A = Surface Area
- π = A constant, either the `MATLAB` built-in value (equation 1) or 3.14 (equation 2)
- r_1 = Equatorial Radius
- r_2 = Polar Radius
- $\gamma = arccos\left(\frac{r_2}{r_1}\right)$.

1.3 Methods and Pseudo-code

The `MATLAB` script should carry out the following pseudo-code:

```
1. Prompt user to input equatorial radius to r1 and polar radius to r2
    • Error check user input for viability (e.g. no negative, zero, complex, or pure imaginary numbers)
    • If the code detects the equatorial axis to be shorter, it informs the user of the error
    • If the code detects both radii to be equal (rendering the shape to be a perfect sphere), the code throws an error
2. Define variables for $\pi$
    • let pi be the built-in MATLAB values of $\pi$
    • let pi2 be 3.14
3. Calculate Surface Area using pi via the above equations
4. Calculate Surface Area using pi2 via the above equations
5. Compare the Area values from two calculations
```

1.4 Calculations and Results

With an example equatorial radius of 6378.137 km and polar radius of 6356.752, the resulting output from the script is:

```
Which problem to test?
1
Enter number for Equatorial (larger) radius in km (r1):
6378.137
Enter number for Polar (smaller) radius in km (r2):
6356.752
For the input value of 6378.137 km as r1 and 6356.752 km as r2:
- The surface area calculation (rounded to 5 sig figs) yielded 510070000 km^2
- The surface area approximation (rounded to 5 sig figs) yielded 509500000 km^2
The approximate difference is 570280 km, and the percent difference is 0.11187%
```

When the user inputs the equatorial radius as smaller than the polar radius, the program informs the user of their error (r1 and r2 reversed from previous example):

```
Which problem to test?
1
Enter number for Equatorial (larger) radius in km (r1):
6356.752
Enter number for Polar (smaller) radius in km (r2):
6378.137
Error:  You input the equatorial axis smaller than the polar.
```

With an example equatorial radius of -1 km and polar radius of 6356.752, the resulting output from the script is:

```
Which problem to test?
1
Enter number for Equatorial (larger) radius in km (r1):
-1
Enter number for Polar (smaller) radius in km (r2):
6356.752
Error:  Input non-zero, positive real radii
```

1.5 Discussion and Conclusion

It can be shown that the approximation of the surface area of the oblate spheroid underestimates the magnitude of its true calculated surface area. The calculation and approximation differed by about 500,000 km^2 between the two output values, but given the magnitude of the answers, the difference is not significant. A test of various other radii show that the percent difference varies greatly, particularly when both radii differ by great amounts, resulting in some errors over 100%. These tests were also shown to be underestimates, however.

2 Neighbor Identification

2.1 Introduction

The goal in this problem is to output the neighbors of a user-chosen cell in a 2D linearly-indexed grid with dimensions specified by the user. Various tests to determine the location of the cell in relation to the top, bottom, left, and right will narrow down possible neighbors, and in the end, each neighbor will be printed in increasing order. Finally, a theoretical 3D implementation will be discussed further.

2.2 Model and Theory

A *linear indexing* uses a single number to identify a location in an array rather than multiple numbers (such as x, y, z and x_1, x_2, x_3). In order to identify the number of neighbors, four criteria will be used to determine is the cell is on the top, bottom, left, or right, which will limit the number of neighbors. Initially, the program calls for 3 inputs:

- M = The number of rows of the matrix
- N = The number of columns of the matrix
- P = The cell number to be analyzed

Using the generic values of M and N, we can determine how many neighbors P has.

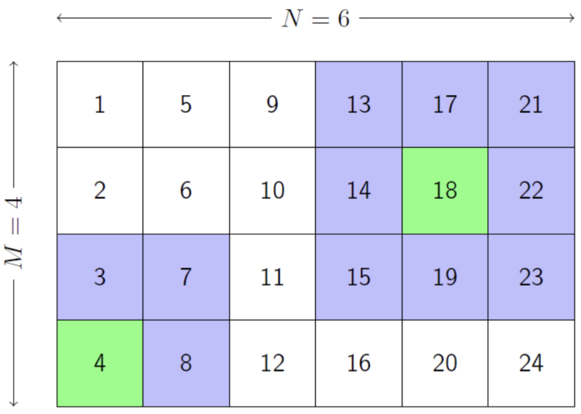

Figure 1: An 4x6 array with cells 4 and 18 analyzed

The above example 4 by 6 array demonstrates the possible neighbors of cells 4 and 18. For each cell, if it is at an extreme, there can be no neighbors further in that direction (e.g. Because cell 4 is on the left, it has no neighbors to the left). Therefore, the code will need to narrow down impossible neighbors of each cell.

To be on the edge, the cell can be on the top, bottom, left, or right. On the top and bottom a different method is used than left and right.

- In the 2D case, linear indices first trace out a column, and then trace one last column at the end. Then the rows follow by stacking columns. If the cell P is less than the value of M, or if it is less than M away from the end, then it is on the edge on the left or right respectively.
- Additionally, if the cell is on the bottom or top, the modulus function can be used. If the cell is on the bottom, then its index is divisible by M, since all columns, which are M indices long, end at the bottom. Similarly, if P is at the top, one *less* than P will be at the bottom of the array.

2.3 Methods and Pseudo-code

The `MATLAB` script should carry out the following pseudo-code:

1. Prompt user to input values for M, N, and P
    - Error check user input for viability (e.g. M and N have to be positive, nonzero integers, and P has to be an integer not greater than M*N)
2. Create array of neighbors
    - Top left = P-M-1
    - Middle left = P-M
    - Bottom left = P-M+1
    - Top = P-1
    - Bottom = P+1
    - Top right = P+M-1
    - Middle right = P+M
    - Bottom right = P+M+1
3. Detect if P is at any edge and sets impossible neighbors in array to -1.
    - If P is on the left ($P < M$), remove Top Left, Middle Left, and Bottom Left neighbors from list.
    - If P is on the right ($P > M(N-1)$), remove Top Right, Middle Right, and Bottom Right neighbors from list.
    - If P is on the bottom (P *divisible by* M), remove Bottom Left, Bottom, and Bottom Right neighbors from list.
    - If P is on the top ($P-1$ *divisible by* M), remove Top Left, Top, and Top Right neighbors from list.
4. Removes impossible neighbor elements from neighbors array (value -1)
5. Print cell number and its possible neighbors in increasing order, each separated by a space

2.4 Calculations and Results

With M = 4, N = 6, and P = 4, the resulting output of the script is:

```
Which problem to test?
2
Enter an integer for the number of rows (M):
4
Enter an integer for the number of columns (N):
6
Enter an integer for the cell (P):
4
Cell ID: 4
Neighbors:  3 7 8
```

With M = 1, N = 6, and P = 4, the resulting output of the script is:

```
Which problem to test?
2
Enter an integer for the number of rows (M):
4
Enter an integer for the number of columns (N):
6
Enter an integer for the cell (P):
```

```
4
Error:  Numbers of Rows and Columns has to be greater than 1
```

With M = 4, N = 6, and P = 25, the resulting output of the script is:

```
Which problem to test?
2
Enter an integer for the number of rows (M):
4
Enter an integer for the number of columns (N):
6
Enter an integer for the cell (P):
4
Error:  P must be inside the array
```

2.5 Discussion and Conclusion

The resulting outputs show the value of the neighbors depending on the location of the cell. In practice, it is easy to see why multiple variables are used for array values identification, since each dimension can be analyzed one at a time, but even in 2D, one variable in this problem did help with some things, such as quickly identifying if P is in the array or not.

For 3D implementation, a similar method would be used for calculating whether an index is at an edge. For clarity's sake, let us notate the dimensions as A, B, and C, and that the linear indices expand in one direction A times, then B times, then C times.

In the 2D case, linear indices first trace out columns, but in 3D, they trace out a plane. Then the columns are stacked in rows.

- The first dimension deals with the stacking of planes. If the value of P is less than the number of cells in a plane spanned by A rows and B columns, then it is on an edge along the third dimension C. Similarly, if P is close to the end of the array, within one plane of the end (P less than AB(C-1)), then it is on the edge on the other side.
- In the second dimension, if the cell is on the bottom or top, the modulus function can be used. If the cell is on the bottom, then its index is divisible by A, since all columns, which are A indices long, end at the bottom. If it is on the top, then subtracting one less will place it back at the bottom, which can be determined as previous. This determines if the cell P is on the edge along the first dimension A.
- The last dimension can be distinguished due the simple nature of a 3D space. In order to create the basic plane shape discussed before, if A counts up and down then B counts across to finish the plane, being the direction for this third classification. If the cell P is divisible by B, or P-1 is divisible by B, then it is on an edge in the dimension B.

Thus, these three dimensions each eliminate one direction of neighbors, assuming that the cell touches the edges in these dimensions.
Another approach might deal with classifying a cell as a corner, edge, face, or center piece. In 2D, this system wasn't used due to the nature of my own approach to the problem, but would give corners 3 neighbors, edges 5 neighbors, and middle pieces 8 neighbors. In 3D, corners would have 7 neighbors, edges would have 11, faces would have 17, and middle pieces would have 26.