Exam project Bi-linear interpolation on a rectilinear grid in two dimensions

This document outlines what project I got, and what goals I choose for this project

NOTE This is a PANDOC FLAVOURED MARKDOWN document, made with the Pandoc notation in mind, that is, it can be converted to pdf using the pandoc utility (PDF version enclosed, do check that out if you prefer), this allows the use of Latex style mathematics.

Locating the project

My Student ID is 201805275, with last two digits 75. There are 23 possible projects, and my project should be $75 mod 23$. I believe $75 = 23 \cdot 3 + 6$ so $75 \equiv 6 mod 23$.

That project is Bi-linear interpolation on a rectilinear grid in two dimensions.

Summary of the project

Part A: **Build an interpolating routine which takes as the input the vectors {xi} and {yj}, and the matrix {Fi,j} and returns the bi-linear interpolated value of the function at a given 2D-point p=(px,py).** Success, 6 points

Part B: **Implement bi-cubic interpolation**. Success, 3 points

Part C: **Implement N-linear interpolation for a grid with arbitrary dimension N.**, Success, 1 point

In total, 10 points.

In all exercises I have verified that all boundary conditions are sattisfied within a relative and absolute precision of 10^{-5}, I have also plotted some demonstrative examples.

Data from published research has been used in part A and B, to demonstrate my procedure.

Below follows a more detailed description of what I have done, what exactly my output is, and what third party data has been used to demonstrate my procedure.

The Data used to demonstrate this

To demonstrate that this is working, I should interpolate some 2D data. I choose to use publicly available terrain elevation data for various features in the continent Tharsis on Mars. The height data used is in Public Domain, (Publisher:

GeoScience PDS Node, Author: MOLA Team, Published: 2014, link to data https://astrogeology.usgs.gov/search/map/Mars/GlobalSurveyor/MOLA/Mars_MGS_MOLA_DEM_mosaic_ The original data is not included anywhere in this repository, as it is more than 2 GB. Instead I have converted the data to a tab-separated list containing the data, which are included in this project.

The examples I use are:

The mountain Olympus Mons, a volcano which is the largest mountain in the solar system (Data from 220 deg East to 233 East, and 12 deg North to 25 deg North), here interpolating between 8 by 8 grid points.

The Valles Marineris, a system of canyons through the Tharsis continent on Mars. (Data from 70 deg East to 130 deg East and 20 Deg South to the Equator NOTE this example is not square, but the output images display it as square, as it better fits inside the image that way, here interpolated between a 48 by 16 point grid).

For part C, I wanted some 3-dimensional data, I simply made a 3D checker-board, which I show different 2D slices of, two verify that the interpolation is working in 3D. I originally considered something like Hydrogen wavefunctions, but they really don't look good when downscaled and interpolated.

The task

Task A

The given task was **Build an interpolating routine which takes as the input the vectors {xi} and {yj}, and the matrix {Fi,j} and returns the bi-linear interpolated value of the function at a given 2D-point p=(px,py).**

The output are 6 png images, with 3 images of 2 different examples: OlympusMons_both.png shows both the original and interpolated data, OlympusMons_int.png shows only the interpolated data as a solid mesh, and OlympusMons_above.png shows the interpolated data from above, with a heatmap and contours marked for the example with Olympus Mons. The three remaining files do the same for the other examples; in all cases the x and y axes are longitude and latitude in degrees (East and North). The txt file outA.txt contains a few tests verifying that the interpolation satisfies the conditions.

This could have been hard, if I did not already have a perfectly working linear equation solver from the homework, which I used to solve the boundary condition equations.

I went with an object oriented approach, where I set up the parameters of the linear interpolation once, and then just call it with my coordinates, that way I don't need to solve each of the n-1 by m-1 the linear equations more than once (where n by m is the grid resolution).

Task B

Implement bi-cubic interpolation

The output are the exact same 6 png images as in A, and the exact same file, verifying that everything worked.

NOTE, bi-cubic interpolations seems to be defined differently in different sources. I do this in a way slightly different from how cubic interpolation was defined in the book, not because I can not implement it, but because it is simply too slow to run.

I think it will be worth going through what I did.

Once again, we just need a linear set of equations, which my linear equation solver can solve. One way of defining the expression for the interpolated function value is:

$$f^{k,l}(\Delta x, \Delta y) = \sum_{i=0}^{3}\sum_{j=0}^{3} \Delta x^i \Delta y^j a_{ij}^{(k,l)} \tag{1}$$

Where k, l denotes that $x_k \leq x \leq x_{k+1}$ and $y_l \leq y \leq y_{l+1}$, for k, l from 0 up to $m-1$ and $n-1$ respectively. It is considerably easier to work with if we write the interpolated functions in terms of Δx,Δy instead of x and y. We have 16 free parameters per the $(n-1)(m-1)$ rectangles, the first conditions to fulfill are, of course, the functions values at each corner which is:

$$f^{k,l}(0,0) = a_{00}^{(k,l)} = z_{l,k}, \tag{2}$$

$$f^{k,l}(w_k,0) = \sum_{i=0}^{3} w_k^i a_{i0}^{(k,l)} = z_{l,k+1}, \tag{3}$$

$$f^{k,l}(0,h_l) = \sum_{j=0}^{3} h_l^j a_{0j}^{(k,l)} = z_{l+1,k}, \tag{4}$$

$$f^{k,l}(w_k,h_l) = \sum_{i=0}^{3}\sum_{j=0}^{3} w_k^i h_l^j a_{ij}^{(k,l)} = z_{l+1,k+1}. \tag{5}$$

Where w_k and h_l are the width and height of each rectangle, this give us 4 linear equations per rectangle. Leaving us 12 conditions (or 3 conditions for each of the corners) short. A common choice seems to be using the derivatives with respect to x, y, x and y and x which, in general, are:

$$\frac{d}{dx}f^{k,l}(\Delta x, \Delta y) = \sum_{i=1}^{3}\sum_{j=0}^{3} i\Delta x^{i-1}\Delta y^{j} a_{ij}^{(k,l)}, \tag{6}$$

$$\frac{d}{dy}f^{k,l}(\Delta x, \Delta y) = \sum_{i=0}^{3}\sum_{j=1}^{3} j\Delta x^{i}\Delta y^{j-1} a_{ij}^{(k,l)}, \tag{7}$$

$$\frac{d^2}{dxdy}f^{k,l}(\Delta x, \Delta y) = \sum_{i=1}^{3}\sum_{j=1}^{3} ij\Delta x^{i-1}\Delta y^{j-1} a_{ij}^{(k,l)}, \tag{8}$$

We can calculate the derivatives in all the corners. Similarly, we can find the higher order derivatives in all the corners.

If we, for a moment, pretend that we have $d/dxz_{k,l}$,$d/dxz_{k,l}$ and $d^2/dxdyz_{k,l}$ given, we have exactly enough equations. We obviously don't have that. In our book, we used the condition that the derivatives should be continuous in the internal grid points, and second derivatives 0 at the borders. This would also give us enough equations to solve the system… but it would be absurdly slow, as our equations would couple all the coefficients of all the rectangles to one another, we would need to diagonalize a $16(n-1)(m-1)$ matrix, which my code actually can do … but it quickly becomes far too slow (It can be done with the 8 by 8 example, but the 16 by 48 example is not happening this century).

Therefore, I do an evil trick: I will first calculate the derivatives in all the corner points numerically, from the neighbouring points, and then use these derivatives as conditions (except for the edge points, where I use the second derivative being 0 as condition instead). Then I have $(n-1)(m-1)$ independent systems of 16 by 16 matrices to diagonalize, which is far far faster.

This is not ideal, I know, and you may consider removing some or all points for this part. Even so, the result looks considerably better than the bi-linear interpolation.

Task C

Implement N-linear interpolation for a grid with arbitrary dimension N.

The output is the text file OutC.txt, which verifies the same tests as always, and the 5 images checker0.png to checker4.png, which shows the 3D checkerboard at various evenly spaced slices.

The example I chose highlights that midway between two extremes, I get half the two values, as I should, and that at all points the function value changes linearly, as it should.

This should have been easy, but generalizing the formalism to work with arbitrary dimensions proved anything but. And it did require some evil binary trickery for me to loop through all the corners of a N-dimensional hypercube.

I do believe my implementation works for a genereal N-dimensional rectangular grid, though I struggle to visualize the results in a plot. I used evenly spaced slices, and showed these as a heat-map.

One alternative for visualizing 3D functions could be using the Marching cube algorithm to turn it into a solid object at various thresholds – essentially a 3D version of a contour plot – or using techniques such as Raytracing to render the function as a translucent cloud – in much the same way smoke and fog is rendered in modern CG effects. This would, technically, work, and I have previously succeeded in implementing the raytracing method (though in C++ with OpenGL), but I do feel like such plots would be harder to read than 2D slices shown as heatmaps.