Multivariate Interpolation by Invariant Vandermonde Matrices

Introduction

The method of multivariate interpolation described here has been independently derived from first principles. An effort is made to ensure common terms are used for signposting (e.g., Newton-Gregory divided difference method and Vandermonde matrices). There is no claim to originality here, the purpose is to set out a practical and efficient method of multivariate interpolation with partial derivatives. Any claim to originality would rest in the practicality of the simple python library that implements the method and can be freely used as an open-source library available from PyPi and GitHub. This python library implements an efficient invariant inverted Vandermonde matrix. This is not a new idea – Vandermonde matrices are independent of values - but the extension to the use of inverted Vandermonde matrices as invariant for iterated interpolation is computationally efficient. In the biological application for which this method was derived (an electron density topology application) the matrices are simply saved as datafiles rendering all interpolation and partial derivative calculations simple matrix multiplications. In the abstracted PsuMultivarse python library implementation described here there is the use of a singleton object - the matrices are calculated once and only once in the lifetime of the application when required. The decision to leave them as calculated is made for inspection of the code and for flexibility.

Summary

In those cases of interpolation where the coefficients of a polynomial are required (e.g., for differentiation), the solutions can be efficiently found by a form of traditional interpolation but using an invariant Vandermonde matrix, which substantially improves performance.

For multidimensional curves, a multivariate polynomial can be fitted of the form (for the 3-dimensional case)

$f(x,y,z)= a + bx + cy + dxy + ex^2$…. etc

Or more generally:

$$f(x,y,z) = \sum_{\substack{x=0\\y=0\\z=0}}^{\substack{x=n\\y=n\\z=n}} C_i x^i y^i z^i$$

It can be shown that the solution to this for any curve, in any dimension, to any degree, can be described by a system of simultaneous equations, and that the solution to these for a given degree/dimension is invariant. The fact that it is invariant makes for fast computational algorithms for multivariate interpolation to a high degree.

This paper shows that there is no impediment to high dimension multivariate fitting as the method is equivalent to tri-directional tri-polynomial fitting which avoids the dimension problem inherent in matrices.

Assumptions

This method assumes that the data is mapped onto a 0 indexed unit based orthogonal coordinate system. This is of course not always the case, for example in the case of the probability density function resulting from an x-ray crystallographic experiment. In this case, there exists a function that maps points from real space into coordinate space and back again, so it is simply a case of mapping there prior to interpolation, and then mapping back.

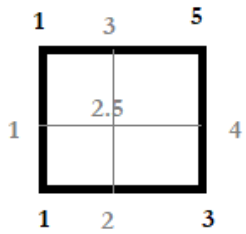
Conventions

We use the python numpy convention of matrices when printed out, so we assume the top left of the matrix to be (0,0). (*and now I need to change everything below because it is all muddled*).

A simple demonstration of the 2d linear case.

If we want to linearly interpolate a square with values as shown, where trivially the centre point by linear interpolation is 2.5:

We can define the values as a matrix, where the matrix is not just a convenient way of storing the data, but the indices in the dimensional directions give us the x/y/z values at that point.



$$VM = \begin{matrix} 1 & 5 \\ 1 & 3 \end{matrix}$$

We know we are expecting a polynomial, this can also be expressed as a matrix, where we also use the dimensional indices, this time the matrix represents the polynomial such that the sum of all the terms is our function.

$$f(x,y) = \begin{matrix} ax^0y^0 & cx^1y^0 \\ bx^0y^1 & dx^1y^1 \end{matrix}$$

Which might look simpler as

$$f(x,y) = \begin{matrix} a & bx \\ cy & dxy \end{matrix}$$

But needs only to expressed as the coefficients, as the variable terms are implied by the position

$$f(x,y) = \begin{matrix} a & b \\ c & d \end{matrix}$$

The calculation of these coefficients will give a unique polynomial fitted to the values.

With 4 knowns and 4 unknowns this can be done as a system of simultaneous equations, where, if we substitute in the x and y values at the appropriate places:

```
f(0,0) = 1 = a + b.0 + c.0 + d.0.0 = a
f(1,0) = 1 = a + b.1 + c.0 + d.1.0 = a+b
f(0,1) = 5 = a + b.0 + c.1 + d.0.1 = a+c
f(1,1) = 3 = a + b.1 + c.1 + d.1.1 = a+b+c+d
```

Expressed as a matrix this gives us:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 1 \\ 5 \\ 1 \\ 3 \end{bmatrix}$$

The matrix above can be seen to be invariant, and not dependent on the values for this case.

Definitions:

- SE is the matrix that defines the simultaneous equations.
- V is the vector of values (collapsed from the matrix) that are observed

- CV is the vector of coefficients for the polynomial

Then

$$SE.CV = V$$

So:

$$SE^{-1}.V = CV$$

Therefore, the inverse of this matrix, is invariant for all 2d linear interpolation, which I have calculated as:

$$\begin{bmatrix}1 & 0 & 0 & 0\\ 1 & 1 & 0 & 0\\ 1 & 0 & 1 & 0\\ 1 & 1 & 1 & 1\end{bmatrix}^{-1} = \begin{bmatrix}1 & 0 & 0 & 0\\ -1 & 1 & 0 & 0\\ -1 & 0 & 1 & 0\\ 1 & -1 & -1 & 1\end{bmatrix}$$

Therefore

$$\begin{bmatrix}a\\ b\\ c\\ d\end{bmatrix} = \begin{bmatrix}1 & 0 & 0 & 0\\ -1 & 1 & 0 & 0\\ -1 & 0 & 1 & 0\\ 1 & -1 & -1 & 1\end{bmatrix} \cdot \begin{bmatrix}1\\ 1\\ 5\\ 3\end{bmatrix} = \begin{bmatrix}1\\ 4\\ 0\\ -2\end{bmatrix} = \mathrm{CV}$$

So

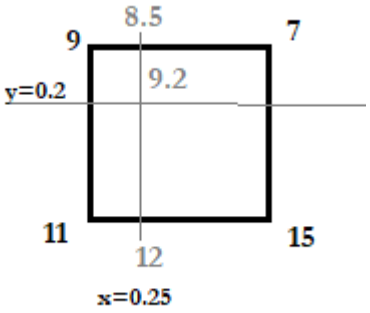
$$\mathrm{CM} = \begin{bmatrix}1 & 4\\ 0 & -2\end{bmatrix}$$

And the function f(x,y) = 1 + 4x - 2xy

If we substitute in (0.5,0.5) we get 1+2 - (2x0.5x0.5) = 2.5 as required.

Verifying the inversion matrix in another simple case

Let us take another case and check we get our expected values:



Therefore

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 1 & -1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 9 \\ 7 \\ 11 \\ 15 \end{bmatrix} = \begin{bmatrix} 9 \\ -2 \\ 2 \\ 6 \end{bmatrix} = \mathrm{CV}$$

So

$$\mathrm{CM} = \begin{bmatrix} 9 & -2 \\ 2 & 6 \end{bmatrix}$$

And the function f(x,y) = 9 -2x + 2y + 6xy

If we substitute in (0.25,0.2) we get 9 - (2x0.25) + (2x0.2) + (6x0.25x0.2) = 9 – 0.5 + 0.4 + 0.3 = 9.2 as required.

The general case

The general case for creating the simultaneous equation matrix SE and the value vector V is, in python code and C# code resp:

For 3 dimensions, n degrees:

```
import numpy as np
def calcLinearAlgebraSE(degree):
    width = degree + 1
    simul = np.zeros((width*width*width, width*width*width))
    ser = -1
    for i in range(0,width):
        for j in range(0, width):
            for k in range(0, width):
                ser+=1
                sec = -1
                for ic in range(0, width):
                    for jc in range(0, width):
                        for kc in range(0, width):
                            sec += 1
                            seCoeff = math.pow(i, ic)*math.pow(j,jc)*math.pow(k,kc);
                            simul[ser, sec] = seCoeff;
```

```
return simul
```

```
double[] valuesVec = new double[totalLength];
int col = 0;
for (int i = 0; i < _width; ++i)
{
    for (int j = 0; j < _width; ++j)
    {
        for (int k = 0; k < _width; ++k)
        {
            valuesVec[col] = _values[i, j, k];
            ++col;
        }
    }
}
```

The simultaneous equation is invariant for the degree, and the inverse of this matrix is also invariant for the degree. Although matrix inversion can be inefficient, it only needs to be done once, ever, in advance, so is not a bottleneck of this method.

The general case for unwrapping the coefficient vector CV into the coefficient matric CM is, in C# code:

For 3 dimensions, n degrees:

```
double[] ABC = Matrices.multMatrixVector(invertedSim, valuesVec);
int pos = 0;
for (int i = 0; i < _width; ++i)
{
    for (int j = 0; j < _width; ++j)
    {
            for (int k = 0; k < _width; ++k)
            {
                _coeffs[i,j,k] = ABC[pos];
                ++pos;
            }
    }
}
```

Partial Differentiation

The partial differentiated polynomials are easily derived from the coefficient matrix by the simple method of moving in the direction of the variable that you are differentiating wrt and multiplying by the index of the original row.

In our second example case:

v = 9 -2x + 2y + 6xy

$$\frac{dv}{dx} = -2 + 6y \; and \; \frac{dv}{dy} = 2 + 6x$$

$$\mathrm{CM} = \begin{bmatrix} 9 & -2 \\ 2 & 6 \end{bmatrix} => CMdx = \begin{bmatrix} 9 & \mathbf{-2} \\ 2 & \mathbf{6} \end{bmatrix} => \begin{bmatrix} \mathbf{-2} \\ \mathbf{6} \end{bmatrix}$$

$$\mathrm{CM} = \begin{bmatrix} 9 & -2 \\ 2 & 6 \end{bmatrix} => CMdy = \begin{bmatrix} 9 & -2 \\ \mathbf{2} & \mathbf{6} \end{bmatrix} => [\mathbf{2} \quad \mathbf{6}]$$

By the same method of using the dimensional indices, this means that

$$\begin{bmatrix}\mathbf{-2}\\ \mathbf{6}\end{bmatrix} = \begin{bmatrix}-2x^0y^0\\ \mathbf{6}x^0y^1\end{bmatrix} = -2 + 6y$$

And

$$[\mathbf{2} \quad \mathbf{6}] = [\mathbf{2} \quad \mathbf{6}x] = 2 + 6x$$

Rule: move -1 in the direction of the dimension and multiply by the index. Note this continues to work if you want to differentiate twice, or first wrt to x then wrt y, etc.

Partial differentiation as a 1-dimensional derivative

Instead of creating a multivariate function for the above square, we could have performed linear interpolation 2 times to get the solution.

We could have found that when x=0.25

```
V = 8.5 + 3.5y
```

And when y = 0.2

```
V = 9.4 - 0.8x
```

As solutions to the values this is unnecessary, they give the same answer.

But as derivatives:

$$\frac{dV}{dy} = 3.5$$
$$\frac{dV}{dx} = -0.8$$

Evaluating the partial derivatives at (0.25,0.2)

$$\frac{dv}{dx} = 2\text{-}6y = -0.8$$
$$\frac{dv}{dy} = 2+6x = 3.5$$

So, at a given point, the solution to the derivatives for a multi-polynomial solution is the same as the partial derivatives. This extends to the second derivative and n-dimensions and thus facilitates the calculation of the Laplacian (2nd partial derivatives for 3-dimensional data) via a 1-dimensional method. The solution to multiple 1-dimensional polynomial fitting is a recursive algorithm using the Newton-Gregory divided difference method as found in the python library PsuMultivarse class PolySolver. It is less efficient than the multivariate method, but numerically stable to a higher degree.

Partial differentiation, general formula

General functions for partial differentiation wrt x,y or z are given in C# code:

```
private double[,,] diffWRTx(double[,,] vals)
{
  double [,,] result = new double[vals.GetLength(0),vals.GetLength(1),vals.GetLength(2)];
  for (int i = 1; i < vals.GetLength(0); ++i)
    for (int j = 0; j < vals.GetLength(1); ++j)
      for (int k = 0; k < vals.GetLength(2); ++k)
        result[i-1, j, k] = vals[i, j, k] * i;
  return result;
}
```

```
private double[,,] diffWRTy(double[,,] vals)
{
```

```
  double[,,] result = new double[vals.GetLength(0),vals.GetLength(1),vals.GetLength(2)];
  for (int i = 0; i < vals.GetLength(0); ++i)
    for (int j = 1; j < vals.GetLength(1); ++j)
      for (int k = 0; k < vals.GetLength(2); ++k)
       result[i, j-1, k] = vals[i, j, k] * j;
  return result;
}
```

```
private double[,,] diffWRTz(double[,,] vals)
{
  double[,,] result = new double[vals.GetLength(0),vals.GetLength(1),vals.GetLength(2)];
  for (int i = 0; i < vals.GetLength(0); ++i)
    for (int j = 0; j < vals.GetLength(1); ++j)
      for (int k = 1; k < vals.GetLength(2); ++k)
        result[i, j, k-1] = vals[i, j, k] * k;
  return result;
}
```

Evaluating a function or a derivative

Given the calculations of the values matrix or the derivatives as coefficient matrices as described, the evaluation of the value or derivative at any given point x,y,z is easily performed by the c# code:

```
double getValue(double x, double y, double z, double[,,] coeffs)
{
  double value = 0;
  for (int i = 0; i < coeffs.GetLength(0); ++i)
  {
    for (int j =0; j < coeffs.GetLength(1); ++j)
    {
      for (int k = 0; k < coeffs.GetLength(2); ++k)
      {
        double coeff = coeffs[i, j, k];
        double val = coeff * Math.Pow(z, i) * Math.Pow(y, j) * Math.Pow(x, k);
        value += val;
      }
    }
  }
  return value;
}
```

As a replacement for the Newton-Gregory divided difference method

If we want to fit a 1-dimensional polynomial to a sequence, we can follow this method.

Let's look for a quadratic fit to: 1,2,7

f(x) = a + bx + cx^2

CM = $[a \quad b \quad c]$

```
1 = a
2 = a + b + c
7 = a + 2c + 4b
```

Expressed as a matrix this gives us:

$$\begin{bmatrix}1 & 0 & 0\\ 1 & 1 & 1\\ 1 & 2 & 4\end{bmatrix}\cdot\begin{bmatrix}a\\ b\\ c\end{bmatrix}=\begin{bmatrix}1\\ 2\\ 7\end{bmatrix}$$

$$\begin{bmatrix}1 & 0 & 0\\ 1 & 1 & 0\\ 1 & 2 & 4\end{bmatrix}^{-1} = \begin{bmatrix}1 & 0 & 0\\ -1.5 & 2 & -0.5\\ 0.5 & -1 & 0.5\end{bmatrix}$$

Therefore

$$\begin{bmatrix} a \\ c \\ d \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -1.5 & 2 & -0.5 \\ 0.5 & -1 & 0.5 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 7 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix} = \text{CV}$$

$$\text{So CM} = [1 \quad -1 \quad 2]$$

And the function f(x) = 1 - x + 2x^2 is evidently correct.

As before, this matrix is invariant for the fitting of all quadratic polynomials.

Some precalculated matrices

Note that the code above follows the format that z is the fastest changing axis, then y, then x.

3 dimensional linear multivariate fit:
f(x,y,z) = a + bx + cy + dz + exy + fxz + gyz + hxyz
(nb the a,b,c etc are randomly assigned)

```
[[ 1.  0.  0.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.  0.]
 [-1.  0.  1.  0.  0.  0.  0.  0.]
 [ 1. -1. -1.  1.  0.  0.  0.  0.]
 [-1.  0.  0.  0.  1.  0.  0.  0.]
 [ 1. -1.  0.  0. -1.  1.  0.  0.]
 [ 1.  0. -1.  0. -1.  0.  1.  0.]
 [-1.  1.  1. -1.  1. -1. -1.  1.]]
```

Cubic 1-dimensional fit (spline):
f(x) = a + bx + cx^2 + dx^3

```
[[ 1.      0.         0.         0.    ]
 [-1.8333  3.        -1.5        0.3333]
 [ 1.     -2.5       2.        -0.5   ]
 [-0.1667  0.5      -0.5        0.1667]]
```

5-degree 1-dimensional fit:

```
[[ 1.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00   0.0000e+00  0.0000e+00]
 [-2.2833e+00  5.0000e+00 -5.0000e+00  3.3333e+00  -1.2500e+00  2.0000e-01]
 [ 1.8750e+00 -6.4167e+00  8.9167e+00 -6.5000e+00   2.5417e+00 -4.1667e-01]
 [-7.0833e-01  2.9583e+00 -4.9167e+00  4.0833e+00  -1.7083e+00  2.9167e-01]
 [ 1.2500e-01 -5.8333e-01  1.0833e+00 -1.0000e+00   4.58333e-01 -8.3333e-02]
 [-8.3333e-03  4.1667e-02 -8.3333e-02  8.3333e-02  -4.16667e-02  8.3333e-03]]
```

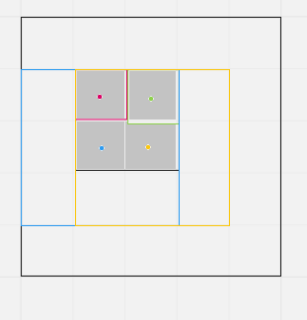
Interpolation

To interpolate over a set of data a multivariate or tri-directional solver needs to be created for every value required for the surrounding points.

Computationally inefficient but memory efficient, an algorithm asks for the value at a given point, and the multivariate solver is created and returned at that point.

Better computationally but with memory implications, the complete set of points that require interpolation is passed to the algorithm and a set of multivariate or tri-directional solvers is created for each mid-point. These can be used repeatedly for all points within a grid square, making a finer grained interpolation more efficient.

Where the degree is greater than expanse of the grid, it can be reduced. See the figure for a depiction of the multivariate solvers that would be created to interpolate the shaded square. The interpolation grids are drawn around each midpoint.



Both methods are implemented in PsuMultivarse in the Interpolator class.

Conclusion

1. The dimensionality is not an impediment to high dimensional multivariant interpolation as the multivariate function is equivalent to a tri-directional tri-polynomial interpolation requiring only the repeated calculation of 1d polynomials over the space, removing the dimensionality problem inherent in matrices.
2. The speed of the 1d polynomial fit is much improved by the consideration that the matrix solution is invariant, and the solutions can be saved up to a required degree or calculated on the fly if greater than. Or, the matrix solutions can be stored as a singleton object, calculated only the first time they are used (see python library PsuMultivarse)
3. The multivariate functions can be fitted using an invariant matrix so the speed problem inherent in calculating the matrix inverse for the simultaneous equations' solution is not an impediment to speed of calculation – only the matrix size itself.
4. The 2nd partial derivative of the multivariate function d2v/dx2 is equivalent to the 2^{nd} derivative of the final x solution at a given point etc, so the partial derivatives of the multivariate function are equivalent to the set of 3 derivatives using the tri-directional method.
5. A limit is the handling of floating points in computers. In the python implementation in PsuMultivarse the decimal places cause a break down for a degree of 7 degrees in 3d (512x512) and 20 degrees in 1d, (21x21).