Multivariate interpolation

Van Thuan Romoli

Semester 2, 2023/2024

1 Introduction

There are many situations where interpolation needs to be applied to higher dimensional problems, for instance in cartography, image processing, or in simulations of physical systems. Thankfully the process for this is conceptually straightforward; we simply apply the rules of interpolation in 1D in each direction for the higher dimensional problem. The only real trouble is in the application.

The aim of this project is to extend univariate methods of interpolation to higher dimensions. In the first part of the project we consider the case of *bilinear* interpolation, i.e. linear interpolation in two dimensions. In the second part we consider the case of *bivariate polynomial* interpolation, i.e. polynomial interpolation in two dimensions. We derive the interpolation rules by 'convolution', or composition, of the 1D rule in each direction; in our project, by rules we mean using Lagrange basis to obtain a Lagrange interpolant. Lastly, we carry out the error analysis for some bivariate methods.

2 Bilinear interpolation

Consider a function of two variables $f(x, y)$ on some two-dimensional domain $(x, y) \in [a, b] \times [c, d]$ then, for instance, suppose we have sampled data points (x_0, y_0), (x_1, y_0), (x_0, y_1) and (x_1, y_1) at which we have f_{00}, f_{10}, f_{01} and f_{11} defined (i.e. the corners of a rectangle). To obtain an approximation of $f(x_i, y_i)$ where $x_0 < x_i < x_1$ and $y_0 < y_i < y_1$ we can notice that, in the x-direction

$$f(x_i, y_0) \approx f_{i0} = \frac{x - x_1}{x_0 - x_1} f_{00} + \frac{x - x_0}{x_1 - x_0} f_{10}$$
$$f(x_i, y_1) \approx f_{i1} = \frac{x - x_1}{x_0 - x_1} f_{01} + \frac{x - x_0}{x_1 - x_0} f_{11}$$

is simply the Lagrange interpolant as in the linear interpolation case applied at each y value. Then applying linear interpolation in y we see

$$\begin{aligned} f(x_i, y_i) &\approx f_{ii} \\ &= \frac{y - y_1}{y_0 - y_1} f_{i0} + \frac{y - y_0}{y_1 - y_0} f_{i1} \\ &= \frac{y - y_1}{y_0 - y_1} \left(\frac{x - x_1}{x_0 - x_1} f_{00} + \frac{x - x_0}{x_1 - x_0} f_{10} \right) + \frac{y - y_0}{y_1 - y_0} \left(\frac{x - x_1}{x_0 - x_1} f_{01} + \frac{x - x_0}{x_1 - x_0} f_{11} \right) \\ &= A(y - y_1)(x - x_1) f_{00} + (y - y_1)(x - x_0) f_{10} + (y - y_0)(x - x_1) f_{01} + (y - y_0)(x - x_0) f_{11} \end{aligned}$$

where $A = \dfrac{1}{(x_0 - x_1)(y_0 - y_1)}$.

It can be shown that this interpolation formula is equivalent to weighting each sample point by the appropriate area of the rectangle formed by partitioning the grid cell at the evaluation point, Figure 1 illustrates this.

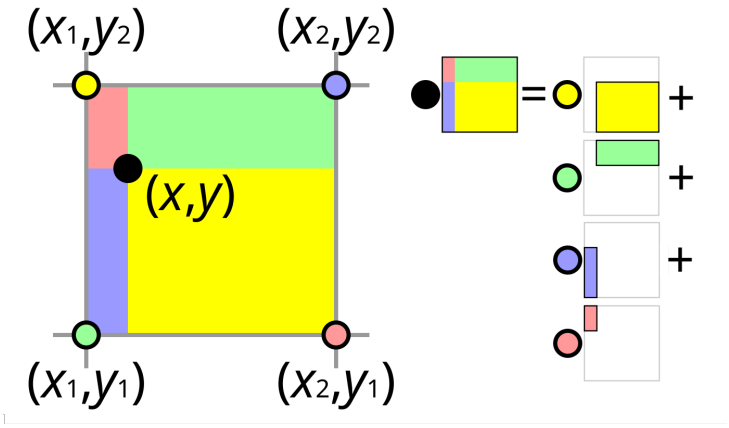


Figure 1: A sample point (x, y) is weighted by the appropriate area of the rectangle formed by partitioning the grid cell at the evaluation point.

In general we will be dealing with a 2D grid of data points, therefore in order to use the formula above we will need to identify the grid cell in which our evaluation point lies; this is equivalent to identifying the correct interpolating polynomial when considering piecewise linear interpolation. To make life easy, here we will assume our data is on a regular grid of points, which makes identifying cells straightforward.

We write a Python function `bilinear(x0,y0,f0)`, to implement bilinear interpolation, using the formula above, and which returns the 2D `interpolant` (a Python function). Our function assumes that the knots are evenly spaced and evaluation points are within the closed interval $[0, 1]^2$. The function parameters `x0` and `y0` represent the coordinates of the knots, `f0` the data values there, and the `interpolant` function itself has as its parameters the coordinates of the evaluation points, say, `xx` and `yy`, and returns the interpolated values on these points.

```
def bilinear(x0,y0,f0):
    '''
    Parameters:
    - x0: array of x coordinates of the knots.
    - y0: array of y coordinates of the knots.
    - f0: 2D array of values of f at (x,y).

    Returns:
    - interpolant: the Python function with parameters xx and yy and returns the
    interpolated values at xx and yy.
    '''

    def interpolant(xx,yy):
        '''
        Parameters:
        - xx: array of x coordinates of the evaluation points.
        - yy: array of y coordinates of the evaluation points.

        Returns:
        - f: 2D array of interpolated values at xx and yy.
        '''

        # This version relies on the data being evenly spaced

        N = np.size(xx)   # new evaluation points
        M = np.size(x0)  # original knot points

        f = np.empty((N,N))

        # calculates spacing assuming interval [-1,1]
        h = (2)/(M-1)
        # making sure we don't overshoot the last subinterval
        k_x = np.minimum(M-2,((xx-x0[0])/h).astype(int))
        k_y = np.minimum(M-2,((yy-y0[0])/h).astype(int))
```

```
        ip = np.arange(N)

        for i in range(N):
            f[i,ip] = (1 / ((x0[k_x[i]]-x0[k_x[i]+1])*(y0[k_y[ip]]-y0[k_y[ip] + 1]))) *
    (
            (yy[ip]-y0[k_y[ip]+1]) * (xx[i]-x0[k_x[i]+1]) * f0[k_x[i],k_y[ip]] -
            (yy[ip]-y0[k_y[ip]+1]) * (xx[i]-x0[k_x[i]]) * f0[k_x[i]+1,k_y[ip]] -
            (yy[ip]-y0[k_y[ip]]) * (xx[i]-x0[k_x[i]+1]) * f0[k_x[i],k_y[ip]+1] +
            (yy[ip]-y0[k_y[ip]]) * (xx[i]-x0[k_x[i]]) * f0[k_x[i]+1,k_y[ip]+1]
            )

        return f

    return interpolant
```

We now provide an example by considering the function of two variables

$$f(x,y) = \sin(x)e^{-y^2} \tag{1}$$

on $(x,y) \in [-1,1]^2$.

We use `bilinear(x0,y0,f0)` to perform bilinear interpolation on a 10×10 grid, evaluating the interpolant(s) on a 100×100 grid on the same domain. We store the interpolated values in a numpy array named `ff1` and the plot of the difference between the exact $f(x,y)$ and these interpolated values on these points is given in Figure 2.

```
def f(x,y):
    '''
    Returns the Python function sin(x) * e^(-y^2).
    '''
    return(np.sin(x) * np.exp(- y ** 2))

# generates knots on a 10x10 grid
x10 = np.linspace(-1,1,10)
y10 = np.linspace(-1,1,10)
X10,Y10 = np.meshgrid(x10,y10,indexing = "ij")
f10 = f(X10,Y10)

# generates the evaluation points on a 100x100 grid
xx = np.linspace(-1,1,100)
yy = np.linspace(-1,1,100)
(XX,YY) = np.meshgrid(xx,yy,indexing = "ij")

# calculates the interpolated values on a 100x100 grid
twodint = bilinear(x10,y10,f10)
ff1 = twodint(xx,yy)

# calculates exact values on a 100x100 grid
fexact = f(XX,YY)

# plots the error between exact and interpolated values on a 100x100 grid
plt.figure(figsize = (8,6))
plt.pcolor(XX, YY, (fexact-ff1), cmap = "coolwarm")
plt.xlabel('x')
plt.ylabel('y')
plt.colorbar(label = "$f(x,y)$ $-$ ff1")
plt.title("Difference between the exact $f(x,y)$ and interpolated \"ff1\" values on a
    100 X 100 grid on $[-1,1]^2$")
plt.show()
```

Finally, we also print the maximum error between the interpolant and the exact function.

```
# calculates the maximum error between the interpolant and exact function
Error_part1 = np.max(np.abs(fexact-ff1))
print(Error_part1)

#0.014226140698689549
```

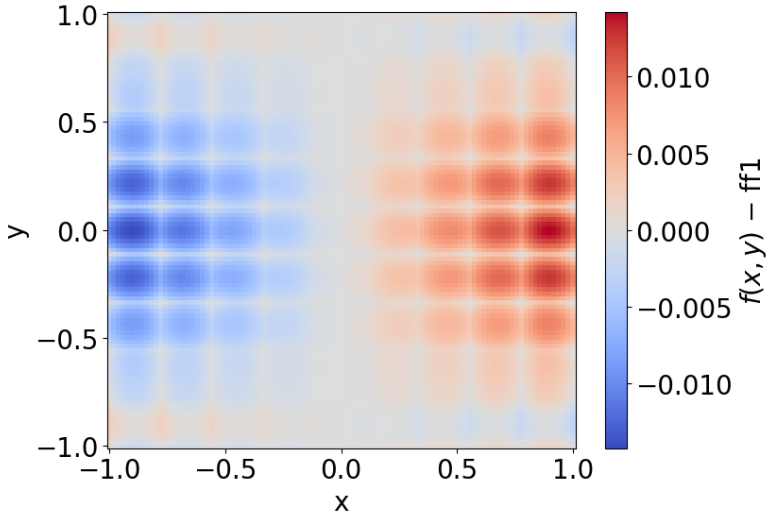


Figure 2: Plot of the difference between the exact $f(x, y)$ and the interpolated `ff1` values on a 100×100 grid on $[-1, 1]^2$.

3 Bivariate polynomial interpolation

Instead of deriving equivalent formulae as the one above, to create higher-order multi-dimensional interpolants we can make use of the composition of the basis functions in each direction, i.e.

$$\begin{aligned} l_j(x) &= \frac{x - x_0}{x_j - x_0} \cdots \frac{x - x_{j-1}}{x_j - x_{j-1}} \frac{x - x_{j+1}}{x_j - x_{j+1}} \cdots \frac{x - x_{n-1}}{x_j - x_{n-1}} \\ &= \prod_{k=0,\, k \neq j}^{n-1} \frac{x - x_k}{x_j - x_k}, \end{aligned}$$

for n distinct knots in each direction $x_0, x_1, ..., x_{n-1}$, and $y_0, y_1, ..., y_{n-1}$, such that the interpolating polynomial is given by

$$p_{n-1}(x, y) = \sum_{j=0}^{n-1} \sum_{i=0}^{n-1} l_i(x) l_j(y) f_{i,j},$$

where $f_{i,j}$ is the data at the knots, the $l_i(x)$ basis functions are in the x-direction (using the knots in x), and the $l_j(y)$ the basis functions in y (using the y knots). If $n = 2$ we recover the bilinear formula above.

We write a Python function `Lagrange_basis(xp,x)` with parameter `xp` representing an array of size n of coordinates of the knots, and with parameter `x` representing an array of coordinates of size m of the evaluation points. The function returns a $m \times n$ array of Lagrange basis where the i, j-th entry corresponds to $l_i(x_j)$.

```
def Lagrange_basis(xp,x):
    '''
    Parameters:
    - xp: array of coordinates of the knots.
    - x:  array of coordinates of the evaluation points.

    Returns:
    - l: m x n array of Lagrange basis where the i,j th entry is l_i(x_j).
    '''

```

```
    n = np.size(xp)     # number of knots
    m = np.size(x)      # number of evaluation points

    l = np.zeros((n,m))

    for j in range(n):  # loop over knots (a small number compared to m)

        # By using array broadcasting we can do all factors in the product together with evaluation points
        # the product should not include xp[j] so one simple approach is to segment in two parts; <j and >j

        p1 = np.prod((x[:,np.newaxis]-xp[:j])/(xp[j]-xp[:j]),axis=1)
        p2 = np.prod((x[:,np.newaxis]-xp[j+1:])/(xp[j]-xp[j+1:]),axis=1)

        l[j,:] = p1*p2
    return l
```

We provide an example on how to use the `Lagrange_basis(xp,x)` to perform a Lagrange Polynomial fit by sampling function (1) using a 5×5 grid of knots, and evaluating the polynomial on a 100×100 grid. We store the interpolated values in a numpy array named `ff2` and the plot of the difference between the exact $f(x, y)$ and these interpolated values on these points is given in Figure 3

```
def f(x,y):
    '''
    Returns the Python function sin(x) * e^(-y^2).
    '''
    return(np.sin(x) * np.exp(- y ** 2))

# generates the knots on a 5x5 grid
x5 = np.linspace(-1,1,5)
y5 = np.linspace(-1,1,5)
X5,Y5 = np.meshgrid(x5,y5,indexing = "ij")
f5 = f(X5,Y5)

# calculates the lagrange basis on a 5x5 grid
Lagrange_x = Lagrange_basis(x5,xx)
Lagrange_y = Lagrange_basis(y5,yy)

# calculates the interpolated values on a 100x100 grid
ff2 = np.zeros((100,100))
for i in range(5):
    for j in range(5):
        ff2 += (Lagrange_x[i][:,np.newaxis] @ Lagrange_y[j][np.newaxis,:]) * f5[i,j]

# plots the error between exact and interpolated values on a 100x100 grid
plt.rcParams.update({'font.size': 12})
plt.figure(figsize = (8,6))
plt.pcolor(XX, YY, np.abs(fexact-ff2), cmap = "coolwarm")
plt.xlabel('x')
plt.ylabel('y')
plt.colorbar(label = "|$f(x,y)$ $-$ ff2|")
plt.title("Error between the exact $f(x,y)$ and interpolated \"ff2\" values on a 100 X 100 grid on $[-1,1]^2$")
plt.show()
```

We also print the maximum error between the interpolant and the exact function.

```
# calculates the maximum error between the interpolant and exact function
Error_part2 = np.max(np.abs(fexact-ff2))
print(Error_part2)

#0.008263713937677108
```

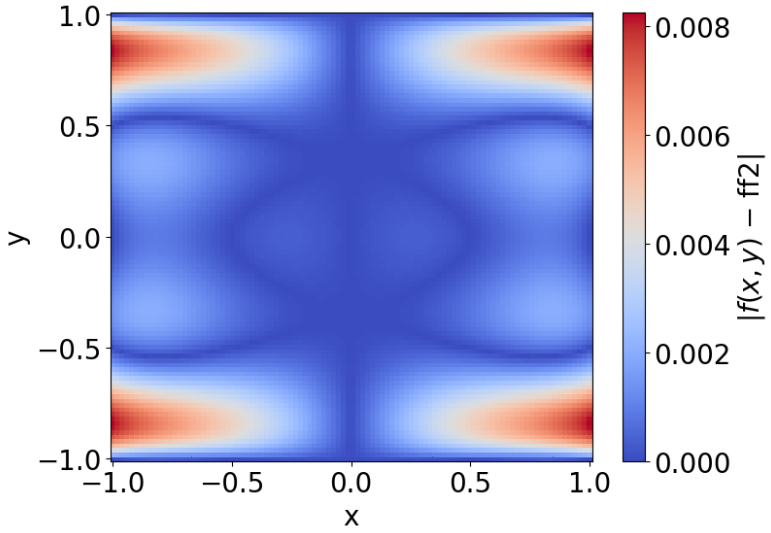


Figure 3: Plot of the difference between the exact $f(x, y)$ and the interpolated `ff2` values on a 100×100 grid on $[-1, 1]^2$.

4 Error analysis

Let $f(x)$ be a function with polynomial interpolant $p_n(x)$ of degree n. Let $f^{(n)}$ be the n-th derivative of f. Assuming that the size of the derivatives is small, $|f^{(n)}| \ll 1$, we may express the maximum error as

$$\max |f(x) - p_n(x)| = \frac{\max \prod_{k=0}^{n} |x - x_k|}{(n+1)!} \max_{x_0 \le u \le x_n} |f^{(n+1)}(u)|.$$

For equally spaced points, we have an error bound which states

$$|f(x) - p_n(x)| \le \frac{h^{n+1}}{4(n+1)} \max_{x_0 \le u \le x_n} |f^{(n+1)}(u)|,$$

where h is the step size and $f^{(n+1)} h^{n+1} \ll 1$. Hence, we find errors decreasing like $h^{(n+1)}$ (at worst). In other words, the convergence is proportional to $h^{(n+1)}$ and so we can expect increasing improvements on reducing h as the order of the polynomial increases.

A generalisation of this result holds for multivariate polynomials, where the degree of the multivariate polynomial is the highest degree of its monomials.

In our project, the maximum error using bilinear interpolation is `Error_part1` $= 0.01423$ (4 s.f.), and the maximum error using bivariate polynomial interpolation is `Error_part2` $= 0.008264$ (4 s.f.). As expected, bivariate polynomial interpolation is more accurate than bilinear interpolation, since the former polynomial has degree 4 (in x and y), while the latter one has degree 1 (in x and y).

We now wish to empirically evaluate the convergence rates between bilinear and bicubic interpolation. Note that we keep the same 100×100 evaluation points and domain throughout. Rather than implementing the functions ourselves, to save time, we use the `scipy` function `RegularGridInterpolator`.

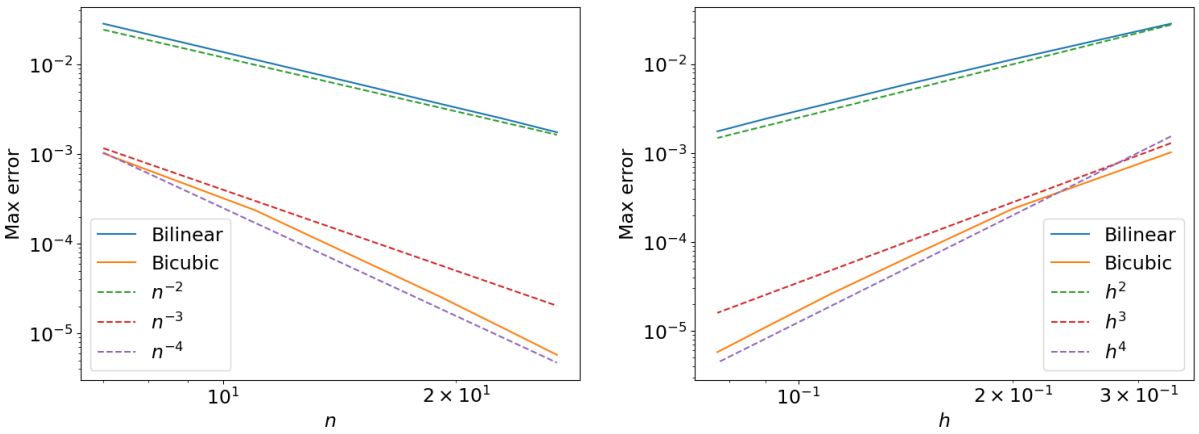
We define a function `maximum_error(n, fexact, method)` which takes the number of knots `n`[1] in each direction, the exact function `fexact`, and the interpolation `method` to be used. The function returns the maximum error between the interpolant and the exact function.

```
1 def maximum_error(n, fexact, method):
2     '''
3     Parameters:
```

[1]Note that henceforth n doesn't represent the degree of the polynomial anymore, rather the number of knots.

```
    - n: number of knots in each direction.

    Returns:
    - Error_partn: the maximum error between the interpolant and the exact function.
    '''

    # generates knots on a n x n grid
    xn = np.linspace(-1,1,n)
    yn = np.linspace(-1,1,n)
    XN,YN = np.meshgrid(xn,yn,indexing = "ij")
    fn = f(XN,YN)

    # calculates the interpolated values on a 100x100 grid
    g = RegularGridInterpolator((xn,yn), fn, method=method)
    ffn = g((XX,YY))

    # calculates exact values on a 100x100 grid
    #fexact = f(XX,YY)
    Error_partn = np.max(np.abs(fexact - ffn))

    return Error_partn
```

Using `maximum_error`, we evaluate the maximum errors for various number of knots using the bilinear and bicubic methods. Figure 4 shows a log-log plot of the max error of bilinear and bicubic interpolation against the number of knots n and step size h.



(a) The x-axis is the number of knots n.

(b) The x-axis is the step size h.

Figure 4: log-log plot of the max error of bilinear and bicubic interpolation against the number of knots n and step size h.

Figure 4b shows that the bilinear interpolation (which uses a polynomial of degree 1) has convergence rate proportional to h^2. We also observe that the bicubic interpolation (which uses a polynomial of degree 3) has a convergence rate proportional to h^3 when h is large and h^4 when h is smaller. We expect the convergence rate to be proportional to $h^{\text{degree of polynomial}+1}$. Hence, we expect for the bilinear case a rate proportional to $h^{1+1} = h^2$, and for the bicubic case a rate proportional to $h^{3+1} = h^4$.

The convergence rate of h^3 for the bicubic case when h is large is justified from the following argument: if the distance between knots, h, is large, then the number of knots, n, is small and we have a best case scenario for the convergence rate at small n. As n gets large (and hence h gets small), we tend to get the expected result of a convergence rate proportional to h^4.

5 Conclusion

In this project we extend univariate methods of interpolation to two dimensions. Specifically, we extend the Lagrange basis and Lagrange interpolant rules in one dimension to derive the rules of the *bilinear* and *bivariate polynomial* interpolation.

This project may be extended in many different ways. A non-exhaustive list would be:

- Derive the rules of multivariate interpolation starting with a non-Lagrangian approach, for example using a Vandermode matrix.
- Derive the rules of multivariate interpolation for higher than 2-dimensions.
- Derive the rules of multivariate interpolation for a non-rectangular, non-regular grid of points.
- Derive the rules of multivariate interpolation for different methods, such as piecewise polynomial interpolation or spline interpolation.
- Perform a more thorough error analysis, for example by discussing Runge's phenomenon.
- Improve code by avoiding for loops, for example using array broadcasting.
- Analyse the computational costs and efficiency.

A Extension 1: Using a Vandermode approach to bivariate interpolation

The bilinear interpolant in Section 2 can be expressed as a multivariate polynomial of degree 1:

$$f(x,y) \approx a_{00} + a_{10}x + a_{01}y + a_{11}xy$$

where the coefficients are found by solving the linear system

$$\begin{bmatrix} 1 & x_1 & y_1 & x_1y_1 \\ 1 & x_1 & y_2 & x_1y_2 \\ 1 & x_2 & y_1 & x_2y_2 \\ 1 & x_2 & y_2 & x_2y_2 \end{bmatrix} \begin{bmatrix} a_{00} \\ a_{10} \\ a_{01} \\ a_{11} \end{bmatrix} = \begin{bmatrix} f_{00} \\ f_{10} \\ f_{01} \\ f_{11} \end{bmatrix} \tag{2}$$

where $f_{ij} = f(x_i, y_j)$ are the corners of the square in Figure 1. The square matrix in (2) is a bivariate linear version of the Vandermode matrix.

We may generalise the multivariate polynomial to the n-th degree:

$$f(x,y) \approx \sum_{i,j \leqslant n} c_{ij} x^i y^j$$

where the c_{ij} coefficients can be found by solving an analogous linear system as in the bilinear case. The Vandermode matrix in this case would be a $4 \times (n+1)^2$ matrix where the i-th row contains all possible products of $1, x_i, x_i^2, \ldots, x_i^n, y_i, y_i^2, \ldots, y_i^n$.

We write a function `vandermonde_matrix(x, y, degree)` which takes an array of `x` coordinates and `y` coordinates of the knots and the `degree` of the polynomial; and which returns the Vandermode matrix for the n-th degree polynomial.

```
def vandermonde_matrix(x, y, degree):
    '''
    Parameters:
    - x: array of x coordinates of the knots.
    - y: array of y coordinates of the knots.
    - degree: integer for the degree of the polynomial.

    Returns:
    - V: the vandermode matrix
    '''
    n = len(x)
    V = np.zeros((n, (degree + 1) ** 2))
    col = 0

    # ranges over all combinations of x^i * y^j in each row and column
    for i in range(degree + 1):
        for j in range(degree + 1):
            V[:, col] = (x ** i) * (y ** j)
            col += 1
    return V
```

We provide an example on how to use the `vandermonde_matrix(x, y, degree)` to perform interpolation by sampling function (1) using a 5×5 grid of knots, and evaluating the polynomial on a 100×100 grid. We store the interpolated values in a numpy array named `ffa` and the plot of the difference between the exact $f(x, y)$ and these interpolated values on these points is given in Figure 5.

```
# calculates the coefficients using the Vandermode matrix on a 5x5 grid
degree = 4
f5 = f(X5.flatten(),Y5.flatten())
V = vandermonde_matrix(X5.flatten(),Y5.flatten(),degree)
coeff = np.linalg.lstsq(V, f5, rcond=None)[0]

# calculates the interpolated values on a 100x100 grid
ffa = np.empty((100,100))
for i in range(100):
    for j in range(100):
        V = vandermonde_matrix(np.array([xx[i]]),np.array([yy[j]]),degree)
        ffa[i, j] = np.dot(V, coeff)

# plots the error between exact and interpolated values on a 100x100 grid
plt.rcParams.update({'font.size': 20})
plt.figure(figsize = (8,6))
plt.pcolor(XX, YY, np.abs(fexact-ffa), cmap = "coolwarm")
plt.xlabel('x')
plt.ylabel('y')
plt.colorbar(label = "|$f(x,y)$ $-$ ffa|")
plt.title("Error between the exact $f(x,y)$ and interpolated \"ff4\" values on a 100 X
    100 grid on $[-1,1]^2$")
plt.show()
```

We also print the maximum error between the interpolant and the exact function.

```
# calculates the maximum error between the interpolant and exact function
Error_parta = np.max(np.abs(fexact-ffa))
print(Error_parta)

#0.008263713937678385
```

As expected, the errors of the bivariate polynomial interpolant is the same regardless of whether the method of the Lagrangian interpolation in Section 3 or the method of the Vandermode matrix in this section is used.

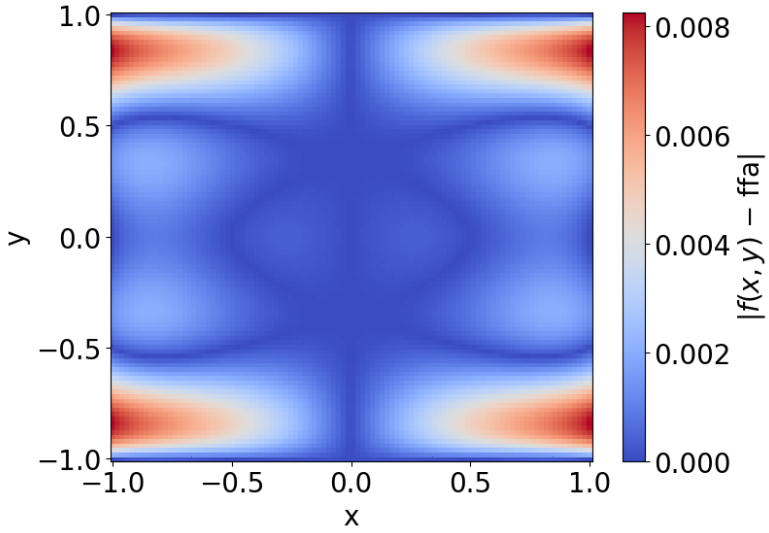


Figure 5: Plot of the difference between the exact $f(x, y)$ and the interpolated `ffa` values on a 100×100 grid on $[-1, 1]^2$.