Contents

1 2DChebClass implementation for different shapes

1.1 The interval (1D)

In order to construct a line in the 2DChebClass library, three inputs have to be given; the endpoints of the interval and the number of discretization points $N+1$.

1.1.1 Discretization points, boundaries, normals

Pseudospectral methods on non-periodic domains are based on polynomial interpolation on non-equispaced points. Typically, Chebyshev points $\{x_j\}$, $j = 0, .., N$, are chosen as collocation points on $[-1, 1]$, which are defined as:

$$x_j = \cos\left(\frac{j\pi}{N}\right), \quad j = 0, 1, ..., N, \tag{1}$$

see [1]. These points are clustered at the endpoints of the interval, and sparse around 0. Using this approach, the points are distributed from 1 to -1, which is counter-intuitive. Therefore, in the code library 2DChebClass [2], which is used in producing the results of this report, the Chebyshev points are automatically flipped back to run from -1 to 1. All computations in 2DChebClass are done on this computational domain $[-1, 1]$. A vector $\vec{x}$ on $[-1, 1]$ can then be mapped onto a physical domain $[a, b]$ of interest via the following linear map

$$\vec{y} = a + \frac{(\vec{x}+1)(b-a)}{2}. \tag{2}$$

The inverse map is

$$\vec{x} = -1 + \frac{2(\vec{y}-a)}{b-a}. \tag{3}$$

The boundary of the domain is found by creating a vector of size N, which contains ones at the boundary points (i.e. x_{min} and x_{max}) and zeros everywhere else. Note that the boundary points are found by evaluating the computational minimum and maximum, rather than the physical one. The normal vectors for the line are simply defined as the one dimensional outward normals $n_1 = -1$, located at y_{min} and $n_2 = 1$ at y_{max}.

1.1.2 Interpolation

A function of interest, f, is evaluated at the Chebyshev points $\{x_j\}$ and a grid function, $f_j := f(x_j)$, is defined. There exists a unique polynomial of degree $\leq N$ that can be used to interpolate a function f on the grid points x_j. The polynomial p_N satisfies, by definition, the following relationship

$$p_N(x_j) = f_j, \tag{4}$$

so that the residual $p_N(x_j) - f_j$ is zero at these points. Therefore, this method is called a collocation method, see [3]. Interpolation on the Chebyshev grid is done using barycentric Lagrange interpolation, derived in [4]. The barycentric formula is

$$p_N(x) = \frac{\sum_{k=0}^{N} \frac{\tilde{w}_j}{x - x_j} f(x_j)}{\sum_{j=0}^{N} \frac{\tilde{w}_j}{x - x_j}},$$

where the weights are defined as

$$\tilde{w}_j = (-1)^j d_j, \quad d_j = \begin{cases} \frac{1}{2} \text{ for } j = 0,\ j = N, \\ 1 \text{ otherwise,} \end{cases}$$

see [4] and [2]. In the code library 2DChebClass, this is implemented as a matrix-vector product, interpolating from the set of points $\{x_j\}$, onto another set of points $\{x_i\}$, where $i = 0, ..., M$ and $j = 0, ..., N$. The interpolation matrix is of the form

$$\text{Interp}_{ij} = \frac{1}{\omega_i} \left(\frac{\tilde{w}_j}{x_i - x_j} \right), \qquad \omega_i = \sum_{k=0}^{N} \frac{\tilde{w}_k}{x_i - x_k}.$$

Generally, the set $\{x_j\}$ lies in computational space $[-1, 1]$, while the second set of points can be customised by the user, to be any M points in $[-1, 1]$. There is another function in 2DChebClass, which takes a set of M points $\{x_j\}$ in physical space as the set to be interpolated onto. This set of points is then first mapped onto the computational domain, using a linear map, before the interpolation matrix is computed, as described above. This method can then be applied to interpolating an arbitrary set of values $\{f_j\}$ onto the new set of points $\{f_i\}$.

1.1.3 Differentiation

The derivation of the Chebyshev differentiation matrices is described below, following the presentation in [1]. Given a polynomial p_N (++definitely same as above??++), where condition (4)

holds, it can be differentiated so that $f'_j = p'_N(x_j)$, which can be rewritten as a multiplication of f_j by a $(N+1) \times (N+1)$ matrix, denoted by D, as follows

$$f'_j = Df_j,$$

using (4). A $(N+1) \times (N+1)$ differentiation matrix D has the following entries, compare with [1]

$$\begin{aligned}
(D)_{00} &= \frac{2N^2+1}{6}, \\
(D)_{NN} &= -\frac{2N^2+1}{6}, \\
(D)_{jj} &= -\frac{x_j}{2(1-x_j^2)}, \quad j = 1, ..., N-1, \\
(D)_{ij} &= \frac{c_i}{c_j}\frac{(-1)^{i+j}}{(x_i - x_j)}, \quad i \neq j, \quad i, j = 0, ..., N,
\end{aligned}$$

where

$$c_i = \begin{cases} 2, & i = 0 \text{ or } N, \\ 1, & \text{otherwise.} \end{cases}$$

The second derivative is represented by the second differentiation matrix D_2, which can be found by squaring the first differentiation matrix; $D_2 = D^2$, and more generally the j^{th} differentiation matrix D_j is defined to be

$$D_j = D^j.$$

However, in [2], the exact coefficients, derived in a similar way as above for D, are used to compute D_2, since it is more accurate than squaring D. Furthermore, all differentiation matrices are derived in computational space and then mapped to physical space using the Jacobian transformation of the linear map (2), which is defined to be

$$J = \text{diag}\left(\tilde{J}\right), \quad \tilde{J}_j = \frac{dy_j}{dx_j}, \quad j = 0, ..., N. \tag{5}$$

Then the physical differentiation matrix D_y is defined as

$$D_y = JD.$$

Higher order differentiation matrices are mapped to physical space in a similar manner.

1.1.4 Integration

In order to evaluate integrals in a similar way, the so-called Clenshaw–Curtis quadrature is used, which is derived in [5]. This is, for the integral over a smooth function f:

$$\int_{-1}^{1} f(x)dx = \sum_{j=0}^{N} w_j f(x_j), \tag{6}$$

where the weights are defined as:

$$w_j = \frac{2d_j}{N} \begin{cases} 1 - \sum_{k=1}^{(N-2)/2} \frac{2\cos(2kt_j)}{4k^2-1} - \frac{\cos(\pi j)}{N^2-1} & \text{for } N \text{ even}, \\ 1 - \sum_{k=1}^{(N-2)/2} \frac{2\cos(2kt_j)}{4k^2-1} & \text{for } N \text{ odd}, \end{cases}$$

see [2]. In 2DChebClass this is implemented as a vector, such that $(\text{Int})_j = w_j$. Again, this is done in computational space, so that we multiply Int pointwise with the transposed Jacobian transformation (5) to map onto a desired physical domain, so that

$$\left(\text{Int}_{\text{phys}}\right)_j = \text{Int}_j \tilde{J}_j.$$

The integration vector can then be applied to a vector of function values f_j.

1.1.5 Convolution

The final aspect to be considered is the convolution matrix, which is needed to compute convolution integrals. The convolution integral is defined as:

$$(n \star \chi)(\vec{y}) = \int \chi(\vec{y} - \vec{z}) n(\vec{z}) d\vec{z},$$

for a density n and a weight function χ. A convolution matrix Conv can be created as follows

$$\text{Conv}_{i,j} = \text{Int}_j \quad \chi(y_i - y_j)$$

where $\vec{y}$ is the vector of physical points in $[a, b]$. The convolution integral is now defined as matrix vector multiplication, by applying the matrix Conv to a density vector n. The convolution matrix can be applied to different densities n, which saves computational time.

1.2 The quadrilateral (2D)

In order to construct a quadrilateral in 2DChebClass, the set of vertices $\{X_i\} = \{(x_1^i, x_2^i)\}$, $i = 1, 2, 3, 4$ has to be provided, as well as the number of discretization points in each direction, N_1 and N_2, has to be specified.

1.2.1 Discretization points and domains

In order to extend the one dimensional considerations to two-dimensional grids, a so-called tensor product grid has to be defined. First, Chebyshev points x_1^j, for $j = 0, ..., N_1 - 1$, on the

x-axis and another set of Chebyshev points x_2^i, for $i = 0, ..., N_2 - 1$ on the y-axis are taken, both between $[-1, 1]$. Then the following two so called Kronecker vectors are defined:

$$\begin{aligned} \mathbf{x}_1^K &= (x_1^0, x_1^0, ..., x_1^0, x_1^1, x_1^1, ..., x_1^1, ..., x_1^n, x_1^n, ..., x_1^n)^T, \\ \mathbf{x}_2^K &= (x_2^0, x_2^1, ..., x_2^m, x_2^0, x_2^1 ..., x_2^m, ..., x_2^1, x_2^2, ..., x_2^m)^T, \end{aligned} \tag{7}$$

where $n = N_1 - 1$ and $m = N_2 - 1$. In $\mathbf{x}_1^K$, each x_1^j is repeated N_2 times, while $\mathbf{x}_2^K$, each sequence $x_2^0, x_2^1, ..., x_2^m$ is repeated N_1 times. The total length of each vector is $N_1 \times N_2$. These vectors are defined, so that the set $(\mathbf{x}_1^K, \mathbf{x}_2^K)$ is a full set of all Chebyshev points on the two-dimensional tensor grid in computational space. An equivalent set can be defined for the points on the physical domain. Note that the points are clustered around the boundary of the two-dimensional grid and sparse in the middle of the grid.

As in the one dimensional case we can map the points $(\mathbf{x}_1^K, \mathbf{x}_2^K)$ to an arbitrary quadrilateral shape discretized via $(\mathbf{y}_1^K, \mathbf{y}_2^K)$. We use the superscript k to indicate that $x^k \in \mathbf{x}^K$, for $k = 0, 1, ..., (N_1 - 1) \times (N_2 - 1)$. We first apply a linear map from $[-1, 1]^2$ to $[0, 1]^2$

$$x_1^k = \frac{x_1^k + 1}{2}, \quad x_2^k = \frac{x_2^k + 1}{2}.$$

Then the bilinear maps

$$\begin{aligned} y_1^k &= \alpha_1 + \alpha_2 x_1^k + \alpha_3 x_2^k + \alpha_4 x_1^k x_2^k, \\ y_2^k &= \beta_1 + \beta_2 x_1^k + \beta_3 x_2^k + \beta_4 x_1^k x_2^k, \end{aligned} \tag{8}$$

take the points on the computational domain to the ones on the physical domain. The α_i, β_i are determined by mapping the coordinates of the four corners of the computational domain in a cyclic order onto the corners of the quadrilateral. The Jacobian of this bilinear map is also saved in this step, as well as the Hessians with respect to both variables. However, most of the time we are given the coordinates of the quadrilateral in physical space, so that we have to map back to the computational domain. In order to do this, we again need to first find the values of the parameters α_i and β_i and then invert the bilinear map to solve for the set of points on the computational domain. We define a matrix B, with $A = BY$, such that $A_i = (\alpha_i, \beta_i)$, where $i = 1, 2, 3, 4$, based on the bilinear maps (8), as

$$B = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 \\ 1 & -1 & 1 & -1 \end{pmatrix}.$$

Once we know the values of the parameters, we can solve the inverse map

$$x_1^k = \frac{y_1^k - \alpha_1 - \alpha_3 x_2^k}{\alpha_2 + \alpha_4 x_2^k}$$

and the quadratic

$$
\begin{aligned}
&a\left(x_2^k\right)^2 + bx + c = 0,\\
&a = \alpha_4\beta_3 - \alpha_3\beta_4,\\
&b = \alpha_4\beta_1 - \alpha_1\beta - 4 + \alpha_2\beta_3 - \alpha_3\beta_2 + y_1^k\beta_4 - y_2^k\alpha_2,\\
&c = \alpha_2\beta_1 - \alpha_1\beta_2 + y_1^k\beta_2 - y_2^k\alpha_2.
\end{aligned}
$$

These give us the values for the x_1^k and x_2^k on $[0,1]^2$. In order to map to the computational domain $[-1,1]^2$, we apply a final linear map

$$x_1^k = 2x_1^k - 1 \quad x_2^k = 2x_2^k - 1.$$

1.2.2 Boundaries and Normals

We can define the boundary of the quadrilateral by defining vectors for each of the four sides of the quadrilateral as follows. We do this on the computational domain $[-1,1]^2$, since each of the sides of the computational domain is conformally mapped onto a corresponding side of the quadrilateral. It is then straightforward to define a boolean vector of size $N_1 \times N_2$, which contains ones where $x_1 == x_{1,min}$ and zeros everywhere else, to define the left boundary of the square. We can then do the same for the other sides. We combine these four vectors to one boundary vector of length $N_1 \times N_2$, which contains ones at each of the four faces and zeros everywhere else.

The normal vectors of the quadrilateral are found by considering the four corners of the shape. We have the four corners $\{Y_i\} = \{(y_1^i, y_2^i)\}$, starting from the left bottom corner and defined in clockwise order. We then have the following set of normals n_l, n_r, n_t, and n_b, for the left, right, top and bottom normal respectively. For the left normal we first define the vector

$$\vec{m}_l = \left(y_2^2 - y_2^1, y_1^2 - y_1^1\right) := \left(m_{l,1}, m_{l,2}\right).$$

We then define the normal to be

$$\vec{n}_l = \mathrm{sgn}(m_{l,1})\frac{\left(-m_{l,1}, m_{l,2}\right)}{\sqrt{m_{l,1}^2 + m_{l,2}^2}},$$

where the negative first component and $\mathrm{sgn}(m_{l,1})$ are taken so that we consider the outward normal. We furthermore want to work with the unit normal, which is the reason for the normalisation term. The other three normals are therefore defined as

$$
\begin{aligned}
\vec{n}_r &= \mathrm{sgn}(m_{r,1})\frac{\left(m_{r,1}, -m_{r,2}\right)}{||\vec{m}_r||_{l_2}},\\
\vec{n}_t &= \mathrm{sgn}(m_{t,2})\frac{\left(-m_{t,1}, m_{t,2}\right)}{||\vec{m}_t||_{l_2}},\\
\vec{n}_b &= \mathrm{sgn}(m_{b,2})\frac{\left(m_{b,1}, -m_{b,2}\right)}{||\vec{m}_b||_{l_2}}.
\end{aligned}
$$

In each of these terms, the definition of the m_1 is the difference in y_2 components, where the top coordinate is subtracted from the bottom coordinate and the m_2 refer to the left y_1 value being subtracted from the right y_1 value. Each of these normal vectors has values on its respective face of the quadrilateral and zeros everywhere else. We then combine these four vectors using the boundary boolean vector which we have defined in the above step. This means that the normals on the corners get summed together, since each of the two faces meeting at a corner has a normal defined at the corner. The normal at a corner is not uniquely defined. However, it is convenient to define it such that it is the average of the two normals from each shape. This is done by normalising the sum of the two normals, as demonstrated here for the corner normal between the left and top face of the quadrilateral

$$\vec{n}_c = \frac{(\vec{n}_l + \vec{n}_t)}{||\vec{n}_l + \vec{n}_t||_{l_2}}.$$

1.2.3 Interpolation

Interpolation in two dimensions is based on the idea of Kronecker tensor products. The interpolation matrices Interp_1 and Interp_2 are computed for each set of points $\{x_1^j\}$ on the x-axis and $\{x_2^i\}$ in the y direction, as explained in Section 1.1.2. The Kronecker product of these two matrices is taken resulting in the two-dimensional interpolation matrix. For an example with $i = j = 0, 1, 2$, this is the block matrix of size 9×9

$$\begin{aligned}\text{Interp} &= \text{Interp}_1 \otimes \text{Interp}_2 \\ &= \begin{pmatrix} \text{Interp}_1(0,0) \times \text{Interp}_2 & \text{Interp}_1(0,1) \times \text{Interp}_2 & \text{Interp}_1(0,2) \times \text{Interp}_2 \\ \text{Interp}_1(1,0) \times \text{Interp}_2 & \text{Interp}_1(1,1) \times \text{Interp}_2 & \text{Interp}_1(1,2) \times \text{Interp}_2 \\ \text{Interp}_1(2,0) \times \text{Interp}_2 & \text{Interp}_1(2,1) \times \text{Interp}_2 & \text{Interp}_1(2,2) \times \text{Interp}_2 \end{pmatrix}.\end{aligned}$$

1.2.4 Differentiation

A similar approach, using Kronecker vectors, can be used to find the Chebyshev differentiation matrices for two-dimensional problems as follows, compare to [1]. For a first derivative D in the x direction, a Kronecker product is taken of the one-dimensional Chebyshev differentiation matrix, which is computed as explained in Section 1.1.3, with the identity, as demonstrated

here with three points

$$D_{x_1} = I \otimes D = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} d_{11} & d_{12} & d_{13} \\ d_{21} & d_{22} & d_{23} \\ d_{31} & d_{32} & d_{33} \end{pmatrix}$$

$$= \begin{pmatrix} d_{11} & d_{12} & d_{13} & & & & & & \\ d_{21} & d_{22} & d_{23} & & & & & & \\ d_{31} & d_{32} & d_{33} & & & & & & \\ & & & d_{11} & d_{12} & d_{13} & & & \\ & & & d_{21} & d_{22} & d_{23} & & & \\ & & & d_{31} & d_{32} & d_{33} & & & \\ & & & & & & d_{11} & d_{12} & d_{13} \\ & & & & & & d_{21} & d_{22} & d_{23} \\ & & & & & & d_{31} & d_{32} & d_{33} \end{pmatrix},$$

where the block structure matches the repetition of each x_1^j in $\mathbf{x}_1^K$. The second derivative with respect to x_1, $D_{x_1x_1}$ can be found equivalently by computing $D_{x_1x_1} = I \otimes D_2$, where D_2 is the second order differentiation matrix in one dimension, as explained in Section 1.1.3. The derivative with respect to y is found by taking the Kronecker product the other way around

$$D_{x_2} = D \otimes I = \begin{pmatrix} d_{11} & d_{12} & d_{13} \\ d_{21} & d_{22} & d_{23} \\ d_{31} & d_{32} & d_{33} \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} d_{11} & & & d_{12} & & & d_{13} & & \\ & d_{11} & & & d_{12} & & & d_{13} & \\ & & d_{11} & & & d_{12} & & & d_{13} \\ d_{21} & & & d_{22} & & & d_{23} & & \\ & d_{21} & & & d_{22} & & & d_{23} & \\ & & d_{21} & & & d_{22} & & & d_{23} \\ d_{31} & & & d_{32} & & & d_{33} & & \\ & d_{31} & & & d_{32} & & & d_{33} & \\ & & d_{31} & & & d_{32} & & & d_{33} \end{pmatrix},$$

which now matches the repetition of each $x_2^0, ...x_2^m$ in $\mathbf{x}_2^K$. We have the following operators for the quadrilaterals in computational space

$$\begin{aligned} \text{Grad} &= \left(D_{x_1}, D_{x_2}\right), \\ \text{Div} &= \left(D_{x_1}, D_{x_2}\right)^\top, \\ \text{Lap} &= D_{x_1x_1} + D_{x_2x_2}. \end{aligned}$$

We need to be map these to the physical domain. This is done by applying the Jacobian of the transformation between the physical and computational space to the differentiation matrix. The Jacobian is defined as

$$J = \begin{pmatrix} \frac{\partial \mathbf{y}_1^K}{\partial \mathbf{x}_1^K} & \frac{\partial \mathbf{y}_1^K}{\partial \mathbf{x}_2^K} \\ \frac{\partial \mathbf{y}_2^K}{\partial \mathbf{x}_1^K} & \frac{\partial \mathbf{y}_2^K}{\partial \mathbf{x}_2^K} \end{pmatrix},$$

however, each of these entries are vectors, so that this becomes a three dimensional array. We compute $\tilde{J} = \left(J^\top\right)^{-1}$ and get the gradient operator

$$\text{Grad} = \left(\text{diag}\left(\tilde{J}_{11}\right) D_{x_1} + \text{diag}\left(\tilde{J}_{12}\right) D_{x_2}, \text{diag}\left(\tilde{J}_{21}\right) D_{x_1} + \text{diag}\left(\tilde{J}_{22}\right) D_{x_2}\right).$$

The construction of the Laplacian follows in a similar manner, using the Hessian matrices of the bilinear map but is more complex in the computational implementation.

1.2.5 Integration

The two dimensional integration vector is found by considering the two integration vectors for the points in the x and the y axis directions. The Kronecker product of these are taken and then multiplied by the determinant of the Jacobian of the mapping to the physical domain, since the integration vectors are computed on the computational grid. This gives the two dimensional integration vector.

1.2.6 Convolution

The construction of the convolution matrix is identical to the one in one dimension, see Section 1.1.5, except that χ takes both $\mathbf{y}_1^K$ and $\mathbf{y}_2^K$ as inputs and the integration vector involved is the two dimensional integration vector. We have

$$\text{Conv}_{i,j} = \text{Int}_j \quad \chi(y_1^i - y_1^j, y_2^i - y_2^j).$$

If the function χ only takes one input, we consider the two given differences of points

$$d_1^{i,j} = y_1^i - y_1^j, \quad d_2^{i,j} = y_2^i - y_2^j,$$

and compute the euclidean distance between d_1 and d_2 to get the convolution matrix

$$\text{Conv}_{i,j} = \text{Int}_j \quad \chi(||d_1^{i,j}, d_2^{i,j}||_{l_2}).$$

1.3 The wedge (2D)

For a wedge, inner and outer radius, maximum and minimum angle as well as the origin of the wedge have to be given. Furthermore, the number of discretization points for each direction has to be given.

1.3.1 Discretization points and domains

A wedge is a section of an annulus, the boundary of which is defined in polar coordinates by an inner and outer radius r as well as a minimum and maximum angle θ_1 and θ_2. While θ_1 should be chosen to lie in $[0, 2\pi]$, the coordinate θ_2 is mapped into this principal domain by computing

$$\theta_2 = \theta_1 + d, \quad \text{where} \quad d = \theta_2 - \theta_1 \mod 2\pi.$$

The wedge can be conformally mapped to a computational domain $[-1, 1]^2$ (in polar coordinates ?!). For this the Kronecker vectors (7) for the computational points $\{x_1\}$ and $\{x_2\}$ are taken. The Kronecker points $\mathbf{x}_1^K$ and $\mathbf{x}_2^K$ can then be mapped to the physical domain in r and θ using the linear map (2) and then back to the computational domain using the map (3) from the one dimensional code. The derivatives of the map (2) for both variables are stored in the Jacobian matrix J.

1.3.2 Boundaries and normals

The boundaries of this shape are found in an identical manner to the quadrilateral case, since this is based on the computational grid. The definition of the normals in polar coordinates is straightforward (+ explain+). On the side with $\theta = \theta_2$ and on $r = r_2$ we have that $\vec{n} = (1, 0)$, $\vec{n} = (0, 1)$ and on the boundaries where $\theta = \theta_1$ and on $r = r_1$, we get that $\vec{n} = (-1, 0)$, $\vec{n} = (0, -1)$. Since these are added together to result in the full set of normals, the set of normals at the corners is $\{\vec{n}_c\} = \{(-1, -1), (-1, 1), (1, -1), (1, 1)\}$.

It can be useful to compute the normals in Cartesian coordinates, for example for plotting or for multishape applications. For this we have to consider the four faces of the shape, the inner and outer radii r_i, r_o and inner and outer angle θ_i and θ_o.
For the outer radius, we get $m_{r2,1} = \text{diag}(1)$ and $m_{r2,2} = \text{diag}(\theta)$. Equivalently, for the side where θ_2 is constant, we have $m_{2,1} = \text{diag}(1)$ and $m_{2,2} = \text{diag}(\theta) + \text{diag}(\pi/2)$. For the inner radius we get have $m_{r1,1} = \text{diag}(1)$ and $m_{r1,2} = \text{diag}(\theta)+\text{diag}(\pi)$. For the side where θ_1 is constant we have $m_{t1,1} = \text{diag}(1)$ and $m_{t1,2} = \text{diag}(\theta)+\text{diag}(3\pi/2)$. The first components of each normal, which correspond to the radial variable, $m_{r2,1}$, $m_{r1,1}$, $m_{t2,1}$, $m_{t1,1}$ are combined to n_1. Any points that do not lie on the boundary are deleted. For the second component of the normals, extra care has to be taken at the corners of the shape to fix the angles. The computation for each pair of angles is explained in Algorithm 1. Then all components $m_{r2,2}$, $m_{r1,2}$, $m_{t2,2}$, $m_{t1,2}$, with the update at the corners is stacked to give n_2 and any points that are not on the boundary are deleted. We combine n_1 and n_2, apply the standard transformation from polar to Cartesian coordinates to get the normal vectors for the wedge in Cartesian coordinates.

Algorithm 1: Determining angles of normals

Given two angles θ_1 and θ_2 at the corner of the wedge, we normalise to get a resulting angle θ_c. Set $\phi = \theta_2 - \theta_1$

if $|\phi| \leq \pi$ **then**
  $\theta_c = \theta_1 + \phi/2$
else if $\pi < \phi$ *and* $\phi < (3/2)\pi$ **then**
  $\theta_c = \theta_1 + (2\pi - \phi)/2$
else
  $\theta_c == \theta_1 - (2\pi - \phi)/2$
end
Set $\theta_c = \theta_c \mod 2\pi$.

1.3.3 Interpolation and differentiation

Since interpolation is done on the computational grid (++ check ++) this is equivalent to the discussion in Section 1.2.3.

In order to derive the differentiation matrices for the wedge, we start with the one-dimensional differentiation matrices described in Section 1.1.3. Since these are derived on the computational domain, we have to map them to the physical domain. This is done for each coordinate individually so that we have

$$D_r = J_1 D \otimes I \quad D_\theta = I \otimes J_2 D,$$

where D is the differentiation matrix for the one-dimensional domain and J_1 and J_2 are the corresponding Jacobians of the mapping from the computational to the physical domain. For each discretization point $j = 1, ..., N$ we have $J_1^j = \frac{dr^j}{dx_1^j}$ and $J_2^j = \frac{d\theta^j}{dx_2^j}$. However, this will result in differentiation matrices with respect to the coordinates r and θ. In order to compute the gradient, divergence and Laplacian we have to use the standard definition of the polar versions to derive these. For $r \neq 0$ we get

$$\begin{aligned}
\text{Grad} &= \left(D_r, \frac{1}{r} D_\theta \right), \\
\text{Div} &= \left(D_r + \frac{1}{r}, \frac{1}{r} D_\theta \right)^\top, \\
\text{Lap} &= D_{rr} + \frac{1}{r} D_r + \frac{1}{r^2} D_{\theta\theta}.
\end{aligned}$$

At $r = 0$, we need to treat these operators more carefully. We know that $\frac{1}{r} D_\theta = D_{r\theta}$, since $D_\theta = 0$ at $r = 0$, and so $\text{Grad} = (D_r, D_{r\theta})$. Similarly, at $r = 0$, we get that $\text{Div} = (2D_r, D_{r\theta})^\top$. Finally we have that $\text{Lap} = 2D_{rr} + \frac{1}{2} D_{\theta\theta} D_{rr}$ at the origin. (++ go back and think why ++)

1.3.4 Integration and convolution

The integration vector is found in the same way as discussed in Section 1.2.5. However, since this is an integral in polar coordinates, in a last step the integration vector has to be multiplied pointwise by r to satisfy the polar integration formula $\int\int f(r,\theta)rdrd\theta$.

The convolution matrix is computed in the same way as in Section 1.2.6, now using the polar version of the integration vector. Note that the convolution we are computing is

$$(n \star \tilde{\chi})(r,\theta) = \int\int \tilde{\chi}(r-r',\theta-\theta')n(r',\theta')dr'd\theta',$$

where $\tilde{\chi}$ is the polar version of the weight function χ.

1.4 The periodic box (2D)

As in the case of the quadrilateral, the required inputs for a periodic box are the number of points in each direction N_1 and N_2, as well as the coordinates of the vertices. However, it is enough to provide $x_{1,\min}$, $x_{2,\min}$, $x_{1,\max}$, $x_{2,\max}$, since we consider a box.

1.4.1 Domains, boundaries, normals

The periodic box has the first coordinate in in Cartesian space, discretized with Chebyshev points, the second coordinate is in Fourier space, using an equispaced discretization with stepsize $h = 2\pi/N$. This coordinate $\vec{x}_2$ is periodic. This means that while for the first coordinate $\vec{x}_1$ the linear maps (2) and (3) are still used, for the second coordinate we have the map

$$\vec{y} = a + (b-a)\vec{x},$$

from $[-1,1]$ to $[a,b]$ and the corresponding inverse map

$$\vec{x} = \frac{\vec{y}-a}{b-a}.$$

The Kronecker vectors are created in the same way as in the case for the wedge, see Section 1.3.1.
The boundary is defined by finding the boundaries of the computational domain, as described in 1.3.1. The normals are defined equivalently to the ones in Section 1.3.1, since the periodic box only needs the unit normals.

1.4.2 Interpolation

In order to construct the interpolation matrix for this problem, we need to consider the interpolation matrices for each coordinate and take the Kronecker product of them. The interpolation

matrix for $\vec{x}_1$ is constructed following the description for one-dimensional domains in Section 1.1.2, using barycentric Lagrange interpolation. However, the interpolation matrix for $\vec{x}_2$ is constructed using Discrete Fourier Transforms (DFT). The account below follows the work in [1]. The DFT formula is

$$\hat{v}_k = h \sum_{j=1}^{N} e^{-ikx_j} v_j, \quad k = -\frac{N}{2} + 1, ..., \frac{N}{2},$$

with stepsize h, and the inverse transform is

$$v_j = \frac{1}{2\pi} \sum_{k=-N/2+1}^{N/2} e^{ikx_j} \hat{v}_k, \quad j = 1, ..., N.$$

This inverse transform is well equipped to act as an interpolant for polar functions on periodic grids with period 2π, once a small adjustment is made. We need to define $\hat{v}_{-N/2} = \hat{v}_{N/2}$ and get the interpolant

$$p(x) = \frac{1}{4\pi} e^{-iNx/2} \hat{v}_{-N/2} + \frac{1}{4\pi} e^{iNx/2} \hat{v}_{N/2} + \frac{1}{2\pi} \sum_{k=-N/2+1}^{N/2-1} e^{ikx} \hat{v}_k, \tag{9}$$

where $x \in [-\pi/h, \pi/h]$, $h = 2\pi/N$. An equivalent definition of the interpolant is derived by considering the interpolant for the delta function, see [1] for a derivation. We get

$$p(x) = \sum_{m=1}^{N} v_m S_N(x - x_m),$$

where

$$S_N(x) = \frac{\sin(\pi x/h)}{(2\pi/h)\tan(x/2)} \tag{10}$$

The interpolation matrix for the periodic variable in the code is defined as

$$\text{Interp}_2 = \left[e^{2\pi i \vec{x}_2^I 0}, e^{2\pi i \vec{x}_2^I j_1}, \cos(2\pi \vec{x}_2^I M), e^{2\pi i \vec{x}_2^I j_2} \right],$$

where $j_1 = 1, ..., M-1$, $j_2 = M+1, ..., N$ and $M = (N+1)/2$ if N is odd and $M = N/2$ if N is even. Note that the cosine term results from combining the first two terms in (9). The set $\vec{x}_2^I$ is the set of points on which the second coordinate is interpolated on. The fast Fourier transform matrix is defined as

$$\text{FFTM} = I \otimes F, \quad \text{where} \quad \text{F}_{jk} = e^{-2\pi i/N jk},$$

where $j, k = 0, ..., N-1$. The interpolation matrix is then

$$\text{Interp} = \Re\left((\text{Interp}_1 \otimes \text{Interp}_2)\, \text{FFTM} \right).$$

(+++ match this section notation wise (indices, N etc.) once clear +++)

1.4.3 Differentiation

In order to derive the differentiation matrices for the periodic grid, we return to the periodic interpolant (10). This can be differentiated to give

$$S'_N(x_j) = \begin{cases} 0, & j \equiv 0 \mod N \\ \frac{1}{2}(-1)^j \cot(jh/2), & j \not\equiv 0 \mod N. \end{cases}$$

This gives one column of the differentiation matrix, which has Toeplitz structure and is created in the code as, see [1]

$$D_{x_2} = 2\pi \begin{pmatrix} 0 & -\frac{1}{2}\cot(h/2) \\ -\frac{1}{2}\cot(h/2) & \frac{1}{2}\cot(h) \\ \frac{1}{2}\cot(h) & -\frac{1}{2}\cot(3h/2) \\ -\frac{1}{2}\cot(3h/2) & \\ & \\ & \frac{1}{2}\cot(h/2) \\ \frac{1}{2}\cot(h/2) & 0 \end{pmatrix}.$$

The factor of 2π is there to map the matrix back onto the grid $[-1, 1]$. For the first coordinate x_1, the differentiation matrix is derived on the Chebyshev grid, as in the one-dimensional example. The two differentiation matrices are each mapped onto the physical domain, using Jacobian transformations, as described in Section 1.1.3. The appropriate Kronecker products of the resulting differentiation matrices in physical space are taken to create Grad, Div and Lap for the periodic box.

1.4.4 Integration and convolution

We use the Clenshaw-Curtis integration weight in the first coordinate, see Section 1.1.4. This is multiplied by the one-dimensional Jacobian to map these weights from the computational onto the physical domain. In the fourier domain however, the integration weights are just $1/N$. These are also multiplied by the corresponding one-dimensional Jacobian. The integration vector for the periodic box is then the Kronecker product of the two one-dimensional integration vectors.

The convolution matrix is constructed in the same way as in Section 1.2.6. However, in the quadrilateral case we computed the function χ at $\left(\vec{y}_1^{\,i} - \vec{y}_1^{\,j}, \vec{y}_2^{\,i} - \vec{y}_2^{\,i}\right)$. In the periodic box however, we need to consider the distances of $y_1^i - y_1^j$ and $y_2^i - y_2^j$, and compute them in two different ways. For the Chebyshev coordinate, nothing changes and we set $d_1^{i,j} = y_1^i - y_1^j$. Since the second coordinate is periodic, the distance between each pair of points in the periodic direction needs

to reflect this. We need to compute

$$d_2^{i,j} = c^{i,j} - \frac{L}{2}, \quad \text{where} \quad c^{i,j} = y_2^i - y_2^j + \frac{L}{2} \mod L,$$

and $L = y_2^{max} - y_2^{min}$. Then convolution matrix is

$$\text{Conv}_{i,j} = \text{Int}_j \ \ \chi(d_1^{i,j}, d_2^{i,j}),$$

where Int is the integration vector for the periodic box. If the function χ only takes one input, we take the euclidean distance between d_1 and d_2 and compute

$$\text{Conv}_{i,j} = \text{Int}_j \ \ \chi(||d_1^{i,j}, d_2^{i,j}||_{l_2}).$$

1.5 Solving the PDE

Regardless of the shape, the PDE can be solved as follows. The initial condition f_0 of the PDE is discretized using Chebyshev points, or in the case of the periodic box, also equispaced points. The PDE is discretized using the derived differentiation and convolution matrices to give a vector dfdt of size N in one dimension and $N_1 \times N_2$ in two dimensions, which is the discretized version of $\partial_t f$. The boundary conditions of the PDE are applied by replacing the boundary points of dfdt with an algebraic equation for the boundary condition, i.e. by computing

$$\text{dfdt(boundary)} = \text{BC equation}.$$

The mass matrix for this method is the identity matrix, denoted by M. At the boundary, we set M(boundary) = 0. Using an algebraic-differential equation solver, such as `ode15s` in Matlab, we solve M dfdt, so that on the boundary we solve BC equation = 0.

2 Convergence Stuff

++++++++++++++++ More on convergence stuff... see Boyd ++++++++++++++++++++++++++
The advantage of Spectral Methods is that, for smooth functions, the convergence is exponential, see [3]:

$$\text{Pseudospectral Error} \approxeq O\left[\left(\frac{1}{N}\right)^N\right].$$

A good overview on spectral methods is given in [1] and a more in depth discussion can be found in [3].

3 Implementation for multishape

In the previous section we have discussed the implementation of the methods for different single shapes. While 2DChebClass supports solutions to PDEs on various single shapes, the method is now extended to compute the solution to a PDE on a multishape. A multishape is a complex domain, which is not fully described by one of the single shapes introduced in the previous section. However, it is possible to discretize the multishape domain in such a way that each of the elements is either a quadrilateral or a wedge. While that does not include all possible complex shapes, it does describe most physically relevant domains. The philosophy of this multishape code is to use the existing code library [2], which is designed to efficiently and accurately solve PDEs on individual shapes, to do the same on a multishape with minimal additional effort for the user.

The solution of a PDE on such a multishape domain is achieved by employing the spectral element method (SEM), since the multishape is discretized into elements. This method is similar in spirit to the finite element method (FEM). FEM discretizes a domain into elements and computes the solution to a given PDE on each of those elements and matches the solution at the boundaries. Expansions of basis functions are used, which are low order polynomials, for interpolation on an equispaced grid. SEM follows the same philosophy but uses higher order basis functions such as Chebyshev or Lagrange polynomials and Chebyshev-Lobatto points on an interpolation grid on each element, as opposed to an equispaced grid, to avoid the Runge phenomenon. At the intersections between the elements, C^0 continuity is enforced, by imposing two matching conditions, usually the solution and the first derivative, see [3]. SEM was first introduced by Patera [6] using Chebyshev polynomials as basis functions and later adapted to Lagrange polynomials by Komatitsch and Vilotte [7], which is now the standard choice [3]. While this method is widely used to solve PDEs in their weak form, in this work the strong form of the PDE is considered, since this aligns best with the existing framework. Furthermore, instead of matching the first derivative of the solution at the intersection of two elements, the flux is matched.

3.1 Setting up the multishape

In order to set up a multishape, each of the discretized elements have to be specified. The information that has to be given for each element is the number of discretization points, N_1 in the x-direction, N_2 in the y-direction, whether the element is a quadrilateral or a wedge and whether to match the internal boundaries between elements. Note that, in order to create a functioning multishape, the number of points of the neighbouring faces of two elements have to match. For a quadrilateral, the four coordinates of the edges have do be given. For a wedge,

inner and outer radius, maximum and minimum angle as well as the origin of the wedge have to be given. As done throughout in the code, the main idea is to stack the vectors of the individual elements to result into a long vector of size $M \times 1$, where $M = \sum_i N_1^i N_2^i$, where i counts the number of elements in the multishape. An equivalent approach is taken for matrices. This stacking results in computations being done directly on the multishape, instead of individually on each element. (++ improve explanation ++)

Once this information is given, the vectors of computational points $\mathbf{x}_1^M$, $\mathbf{x}_2^M$ and physical points $\mathbf{y}_1^M$, $\mathbf{y}_2^M$ on the whole multishape can be set up. This is done as described in Algorithm 2. One important thing to notice is that the vectors $\mathbf{y}_1^M$, $\mathbf{y}_2^M$ are now defined in Cartesian coordinates, even if some of the underlying shapes are wedges, for which polar coordinates are the natural choice.

Algorithm 2: Multishape points

for *ishape in multishape* **do**

- Get computational points $\mathbf{x}_1^i$, $\mathbf{x}_2^i$ and physical points $\mathbf{y}_1^i$, $\mathbf{y}_2^i$.
- Add points to vector containing multishape points:

$$\mathbf{x}_1^M = [\mathbf{x}_1^M; \mathbf{x}_1^i], \quad \mathbf{x}_2^M = [\mathbf{x}_2^M; \mathbf{x}_2^i], \quad \mathbf{y}_1^M = [\mathbf{y}_1^M; \mathbf{y}_1^i], \quad \mathbf{y}_2^M = [\mathbf{y}_2^M; \mathbf{y}_2^i].$$

end

for *ishape in multishape* **do**

if *ishape is polar* **then**

- Convert polar $\mathbf{y}_1^i$, $\mathbf{y}_2^i$ to cartesian $\mathbf{y}_{1,\text{Cart}}^i$, $\mathbf{y}_{2,\text{Cart}}^i$.
- Replace in $\mathbf{y}_1^M$, $\mathbf{y}_2^M$:

$$\mathbf{y}_1^M(\text{ishape}) = \mathbf{y}_{1,\text{Cart}}^i, \quad \mathbf{y}_2^M(\text{ishape}) = \mathbf{y}_{2,\text{Cart}}^i.$$

end

end

3.2 Boundaries and intersections

In order to determine the points that lie on the boundary of a multishape, we first have to determine which faces of which shapes intersect. Once we have this information, we can take the boundaries of the individual shapes and substract the intersection boundaries from these to get the multishape boundary. The intersections between shapes are found by the code

automatically, as explained in Algorithm 3. We iterate through all pairs of shapes to check whether any of their faces intersect by comparing the points of each shape on these faces. One thing to note is that we also have to check whether the points on face i are equal to the flipped vector of points on face j. This is to account for the fact that there are different ways of constructing these points on each shape. Having found the intersections between the shapes, the boundary of the multishape is defined by a boolean vector of size M, containing ones at the boundary and zeros everywhere else, as in the case for single shapes. Algorithm 4 explains the steps.

Algorithm 3: Determining intersections between shapes in a multishape

```
for ishape in multishape do
    for jshape in multishape do
        for iface in ishape do
            for jface in jshape do
                if Pts_iface == Pts_jface then
                        Intersections(ishape,jshape).Pts = Pts_iface
                        Intersections(ishape,jshape).Face = iface
                        Intersections(ishape,jshape).Corners = Corners_iface
                        Intersections(ishape,jshape).Flip = False
                else if iface == flip(jface) then
                        Intersections(ishape,jshape).Pts = Pts_iface
                        Intersections(ishape,jshape).Face = iface
                        Intersections(ishape,jshape).Corners = Corners_iface
                        Intersections(ishape,jshape).Flip = True
                end
            end
        end
    end
end
```

Algorithm 4: Determining the boundary of a multishape

```
for ishape in multishape do
    for iface in ishape do
        for jface in ishape do
            if Intersections(ishape, jshape).Face = iface then
                Set IntersectionTest(iface) = True;
            end
        end
        if IntersectionTest(iface) = False then
            Set Boundary(iface) = True;
        end
    end
end
```

Constructing the normal vector for the multishape is quite complex and explained in Algorithm 5. One thing to be mindful of is the change from polar to Cartesian coordinates and back. The final normal vector contains polar values when corresponding to a wedge element and Cartesian coordinates for quadrilateral elements of the multishape. Extra care has to be taken at the corners of the boundary where two shapes intersect. There the normals are averaged as explained in Algorithm 5. However, when the discretization of the multishape is more complex, this may sometimes not be sufficient, and the resulting outward normal is not sensible. For this reason it is possible to override any of the normals manually as a user. The effect is demonstrated in the validation tests in Section +++ ref validation tests +++.

Algorithm 5: Determining the outward normals of a multishape

for *ishape in multishape* **do**

- Get normal vector inormal for ishape.
- Set normalVec(ishape) = inormal.

if *ishape is polar* **then**

- Get Cartesian normal vector inormalCart for ishape.
- Set normalCart(ishape) = inormalCart.

else

- Set normalCart(ishape) = inormal.

end

- Delete entries that do not lie on the boundary of the multishape.

end

Fixing normals at corners. Find entries in normalCart that share same points (up to a tolerance). Store points in duplicates.

for *dup in duplicates* **do**

if *override is True* **then**
Normal vector n at dup is specified by the user.
else
Average and normalize normals in normalCart(dup) to get n.
end

end

Set normalCart(dup)= n.

for *ishape in multishape* **do**

if *ishape is polar* **then**
Convert normals from normalCart(ishape) to polar coordinates and assign to normal(ishape).
else
Set normal(ishape) = normalCart(ishape).
end

end

3.3 Interpolation, differentiation, integration and convolution

The interpolation matrix for the multishape is constructed by computing the individual interpolation matrices on each shape and stacking them together in a blockdiagonal matrix. The gradient, divergence and Laplacian operators for the multishape are constructed in an equivalent way. The integration vector is constructed by simply stacking the integration vectors for each shape. Each of these constructions is demonstrated in Algorithm 6.

Algorithm 6: Constructing the interpolation matrix, gradient, divergence and Laplacian as well as the integration vector

for *ishape in multishape* **do**

- Get $\text{Interp}_{\text{ishape}}$, $\text{Grad}_{\text{ishape}}$, $\text{Div}_{\text{ishape}}$, $\text{Lap}_{\text{ishape}}$, $\text{Int}_{\text{ishape}}$.
- Set
  Interp = blkdiag(Interp, $\text{Interp}_{\text{ishape}}$),
  Grad = blkdiag(Interp, $\text{Grad}_{\text{ishape}}$),
  Div = blkdiag(Interp, $\text{Div}_{\text{ishape}}$),
  Lap = blkdiag(Interp, $\text{Lap}_{\text{ishape}}$),
  Int = (Int, $\text{Int}_{\text{ishape}}$).

end

While computing the standard interpolation matrix is straightforward, computing the interpolation matrix for a set of physical points in multishape is a bit more complex. Given a set of points $\mathbf{y}_1^I$ and $\mathbf{y}_2^I$ that we want to interpolate onto, we need to first determine which of these points are in which shape, since the interpolation is done shapewise as explained in Algorithm

6. Algorithm 7 illustrates this. (+++ check algorithm after meeting +++)

Algorithm 7: Constructing the interpolation matrix for interpolating onto points in physical space

for *ishape in multishape* **do**

Get $\mathbf{y}_1^I$ and $\mathbf{y}_2^I$ to interpolate onto;

for *ishape in multishape* **do**

- Get computational points x_1, x_2 for ishape.
- Get $x_{1,\min} = \min(x_1) - 10^{-10}$, $x_{1,\max} = \max(x_1) + 10^{-10}$, $x_{2,\min} = \min(x_2) - 10^{-10}$, $x_{2,\max} = \max(x_2) + 10^{-10}$.

if *ishape is polar* **then**

Get polar version of $\mathbf{y}_1^I$ and $\mathbf{y}_2^I$.

end

- Transform $\mathbf{y}_1^I$ and $\mathbf{y}_2^I$ to $\mathbf{x}_1^I$ and $\mathbf{x}_2^I$ in computational space, using the linear mapping associated to the shape.
- Define iMask = $(x_1 >= x_{1,min})$ & $(x_1 <= x_{1,max})$ & $(x_2 >= x_{2,min})$ & $(x_2 <= x_{2,max})$ & doneMask.
- Create new vectors that only contain the points of the shape: $\mathbf{x}_1^I$(iMask), $\mathbf{x}_2^I$(iMask).

for *i in length(*$\mathbf{x}_1^I$*(ishape))* **do**

- Compute interpolation matrix pointwise for $x_1(i) \in \mathbf{x}_1^I$(ishape), $x_2(i) \in \mathbf{x}_1^I$(ishape).
- Set $\text{Interp}_{\text{ishape}}$ = [$\text{Interp}_{\text{ishape}}$, InterpolationMatrix$(x_1(i), x_2(i))$].

end

- Stack interpolation matrices in a blockdiagonal: Interp = blkdiag(Interp, $\text{Interp}_{\text{ishape}}$).
- Set doneMask(iMask) = True.

end

- Set Interp(doneMask) = [].
- Set $\mathbf{y}_1^I = \mathbf{y}_1^I$(doneMask), $\mathbf{y}_2^I = \mathbf{y}_2^I$(doneMask).

end

The convolution matrix cannot be taken from the individual shapes, since convolution is a global operation. We compute it in the exact same way as for a single quadrilateral, see Section 1.2.6, now using the multishape points $\mathbf{y}_1^M$ and $\mathbf{y}_2^M$ and the integration vector that was constructed for the multishape.

3.4 Boundary matching and solving the PDE

As discussed above, the code automatically identifies the intersection boundaries between two shapes when setting up the multishape. Once the intersections between the neighboring shapes are identified, user-defined boundary conditions can be applied. There are currently two options, although the addition of further boundary conditions is straightforward. In general, both the solution to the PDE and the flux are matched at these intersection boundaries to create a coherent solution over the whole shape. Alternatively, hard walls between two shapes can be simulated easily, by applying a no-flux boundary condition at that intersection boundary. On boundaries which are on the outside of the multishape, the boundary conditions of the PDE, such as no-flux and Dirichlet conditions, can be applied in the same way as for single shapes. The application of the boundary conditions and solution of the PDE follows the method for single shapes and is illustrated in Algorithm 8. Note that the mass matrix for a collocation method is diagonal. The boundary and matching conditions are applied by setting the relevant entries in the mass matrix to zero.

Algorithm 8: Applying boundary and intersection conditions, solving the PDE.

- Get discretized PDE dfdt.
- Get boundary condition equation BCeq and intersection boundary condition Intereq.
- Set the mass matrix M = diag(ones).
- Set M(boundary) = 0, M(intersection) = 0.
- Set dfdt(boundary) = BCeq.
- Set dfdt(intersection)= Intereq.
- Compute M dfdt using `ode15s`.

4 Validation Tests Multishape

(Note: all the things from this section are in PDECO/MultiShapeTesting in Matlab. The following files are particularly relevant: PlottingNormalComparison, TablesExactSolutions, TableForwardExamples, TablesDiffIntetc and MultiShapeAssorted. The other files are called inside these wrappers. Plotting is currently commented out but burried in there somewhere...)

4.1 Testing Differentiation, Interpolation, Convolution and Integration

TablesDiffIntetc() in Matlab

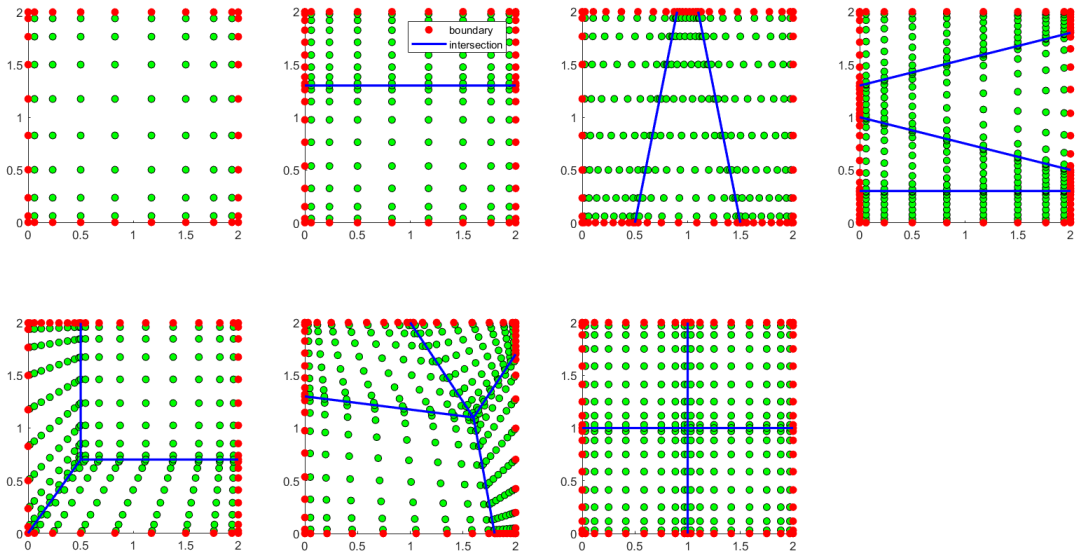

Figure 1: Different discretizations of the box (a - g).

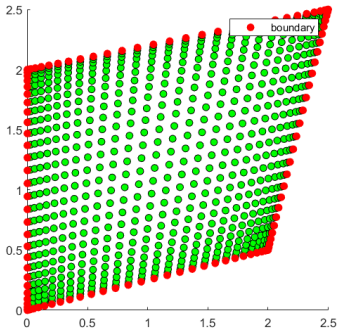

Figure 2: Quadrilateral domain (q).

We investigate the accuracy of differentiation, interpolation, integration and convolution comparing different discretizations of the box and a wedge. The different discretizations can be seen in Figures 1, 2 and 3. The operations are validated using the exact solution

$$\begin{aligned} \rho &= \exp(\alpha_1 t + \beta_1 y_1 + \beta_2 y_2), \\ \nabla\rho &= [\beta_1\rho, \beta_2\rho], \\ \nabla^2\rho &= \nabla\cdot\nabla\rho = (\beta_1^2 + \beta_2^2)\rho, \end{aligned} \tag{11}$$

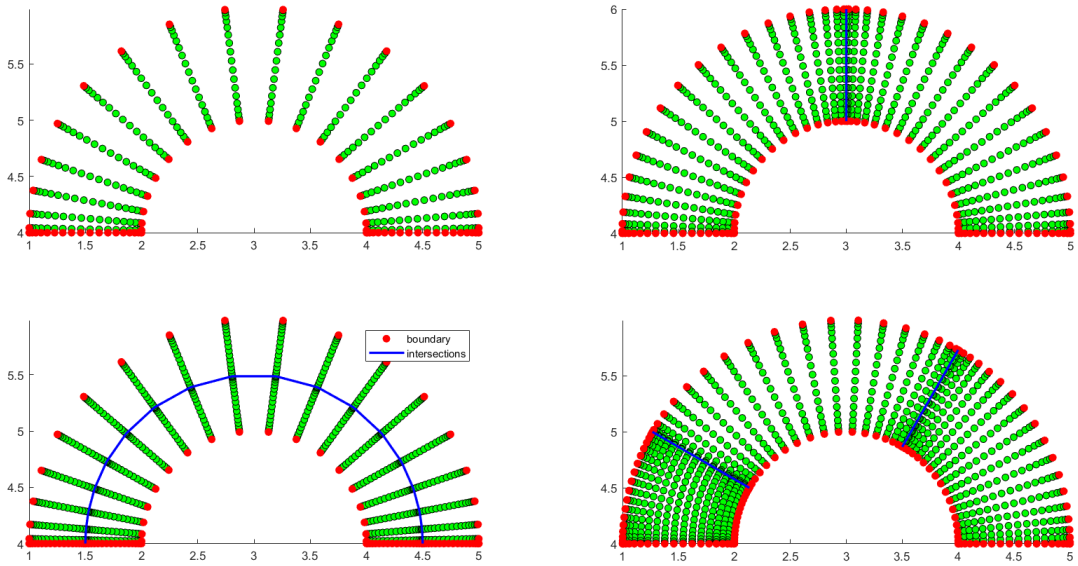

Figure 3: Different discretizations of the wedge (h - k).

where $\beta_1 = 0.1$, $\beta_2 = 0.1$, $\alpha_1 = -0.5$.

We test each of the operators Grad, Div and Lap by computing the error between the numerical and the exact solution

$$\mathcal{E}_{\text{Abs}}(f) = \max \max |f_{num} - f_{ex}| \tag{12}$$

$$\mathcal{E}_{\text{Rel}}(f) = \frac{\mathcal{E}_{\text{Abs}}(f)}{\max \max |f_{ex}|}. \tag{13}$$

The Div operator is tested by taking the divergence of the exact solution for $\nabla\rho$ and comparing it to the exact solution for $\nabla^2\rho$. Note that the exact $\nabla\rho$ has to be transformed into polar coordinates first. The solutions can be seen in Tables 1, 2 and 3.

We investigate the functionality of the interpolation matrix by interpolating from the different multishapes onto a uniform grid. This grid is created by setting up a uniform rectangular grid which fully contains the multishape. Then we loop through the shapes and interpolate the test function ρ (11) onto the uniform points that lie inside each shape. In the end we discard the points that lie outside the multishape. This causes the final solution on the uniform grid to vary in size, depending on the discretization of the multishape. We test the functionality by comparing the interpolated function values to ρ evaluated on the uniform grid points, using (12). The results can be seen in Table 4.

In a second test we interpolate a function ρ, (11), computed with $N = 50$ on shapes (a) and (h) respectively onto a uniform grid, as described above. Then we compute (11) on the multishapes (b), (f), and (g) with $N = 10$, $N = 20$ and $N = 30$. We interpolate this onto the same uniform

grid as shape (a) using the physical interpolation function. The values of ρ coming from (a) can be compared with those from (b), (f), and (g) on the uniform grid, using (a) as a reference solution. Similarly, the solution on shape (h) serves as reference solution for multishapes (j) and (k). The errors, computed using (12) can be seen in Table 5.
Finally, we investigate what happens when we consider interpolation to a uniform grid with $N_1 \neq N_2$ for the wedge, when compared to the exact solution on the uniform grid. This is interesting, because when the solution is interpolated onto a uniform grid (in Cartesian coordinates), it is to be expected that more points are needed in the angular direction than in the radial direction, since in Cartesian coordinates points are more densely clustered in radial direction, as can be seen in Figure 3. This is indeed what happens, as can be seen in Table 6. We get considerably better results for small N_1 than for small N_2 and vice versa.
We consider a similar approach for testing the convolution matrix. We compute the convolution matrix with $N = 50$ on shape (a) (or (h) respectively) and apply it to the function ρ in equation (11). We then interpolate this onto the other shapes and compute the error (12) with the convolution of ρ on that shape. We use the value of the convolution on a single shape with $N = 50$ as the reference value for the multishape convolution. The results are displayed in Table 7.
For the integration vector, we compute the integral of ρ on shape (a) and compare this to the integral of ρ on multishapes (b), (f) and (g). We then compare the integral of ρ on the single shape (h) to the one on discretizations (j) and (k). The errors computed with (12) can be seen in Table 8, where again the value of the integral on the single shape is the reference for the multishape integration.

4.2 Exact Tests

The solution to known testproblems with Dirichlet and no-flux boundary conditions are considered in this section. The errors are calculated as an l_2 error in space and an l_∞ error in time.

4.2.1 Exact Tests - Dirichlet Conditions

(MS_TestJonnaADExactDisectBoxes and MS_TestJonnaADExactDisectBoxesPart2
and MS_TestJonnaADExactDisectWedges) Several examples are run, using exact solutions, to validate the multishape code. This is done using an exact solution to the advection diffusion equation on an infinite domain, so that Dirichlet boundary conditions can be applied, by matching the value of the exact solution on the boundary of the multishape. The exact solution is

	$N = 10$		$N = 20$		$N = 30$	
	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$
a	3.6637×10^{-15}	4.0491×10^{-14}	4.5158×10^{-14}	4.9908×10^{-13}	1.0987×10^{-13}	1.2143×10^{-12}
b	2.5202×10^{-14}	2.7853×10^{-13}	1.3006×10^{-13}	1.4374×10^{-12}	3.9264×10^{-13}	4.3394×10^{-12}
c	3.5680×10^{-14}	3.9432×10^{-13}	2.7724×10^{-13}	3.0639×10^{-12}	1.0356×10^{-12}	1.1445×10^{-11}
d	7.3080×10^{-14}	8.0766×10^{-13}	6.0429×10^{-13}	6.6785×10^{-12}	1.2559×10^{-12}	1.3880×10^{-11}
e	3.1912×10^{-14}	3.5268×10^{-13}	2.0508×10^{-13}	2.2665×10^{-12}	6.5671×10^{-13}	7.2578×10^{-12}
f	5.7149×10^{-14}	6.3159×10^{-13}	4.4410×10^{-13}	4.9081×10^{-12}	1.7717×10^{-12}	1.9580×10^{-11}
g	1.3947×10^{-14}	1.5414×10^{-13}	9.5771×10^{-14}	1.0584×10^{-12}	2.6459×10^{-13}	2.9242×10^{-12}
h	3.1143×10^{-5}	1.3588×10^{-4}	1.3522×10^{-11}	5.9203×10^{-11}	4.0515×10^{-13}	1.7705×10^{-12}
i	1.3490×10^{-7}	5.9567×10^{-7}	2.4433×10^{-13}	1.0689×10^{-12}	4.5003×10^{-13}	1.9658×10^{-12}
j	3.1143×10^{-5}	1.3588×10^{-4}	1.3522×10^{-11}	5.9203×10^{-11}	9.8901×10^{-13}	4.3219×10^{-12}
k	9.8299×10^{-8}	4.2888×10^{-7}	3.1769×10^{-13}	1.3866×10^{-12}	8.6467×10^{-13}	3.7732×10^{-12}

Table 1: Table Grad

	$N = 10$		$N = 20$		$N = 30$	
	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$
a	8.3267×10^{-16}	4.6012×10^{-14}	4.8798×10^{-15}	2.6965×10^{-13}	1.2278×10^{-14}	6.7849×10^{-13}
b	2.9594×10^{-15}	1.6353×10^{-13}	1.3121×10^{-14}	7.2507×10^{-13}	4.0837×10^{-14}	2.2566×10^{-12}
c	8.3614×10^{-15}	4.6204×10^{-13}	2.8682×10^{-14}	1.5849×10^{-12}	1.0649×10^{-13}	5.8845×10^{-12}
d	7.9138×10^{-15}	4.3731×10^{-13}	4.2664×10^{-14}	2.3575×10^{-12}	9.6187×10^{-14}	5.3152×10^{-12}
e	3.6846×10^{-15}	2.0360×10^{-13}	2.1975×10^{-14}	1.2143×10^{-12}	6.9682×10^{-14}	3.8505×10^{-12}
f	9.7717×10^{-15}	5.3997×10^{-13}	6.4991×10^{-14}	3.5913×10^{-12}	1.3360×10^{-13}	7.3824×10^{-12}
g	3.0878×10^{-15}	1.7063×10^{-13}	1.0623×10^{-14}	5.8704×10^{-13}	3.3662×10^{-14}	1.8601×10^{-12}
h	5.2272×10^{-5}	1.6127×10^{-3}	3.4844×10^{-11}	1.0758×10^{-9}	5.3096×10^{-14}	1.6387×10^{-12}
i	9.2692×10^{-8}	2.8672×10^{-6}	3.2897×10^{-14}	1.0155×10^{-12}	6.4497×10^{-14}	1.9903×10^{-12}
j	5.2272×10^{-5}	1.6127×10^{-3}	3.4852×10^{-11}	1.0761×10^{-9}	1.3768×10^{-13}	4.2490×10^{-12}
k	8.4307×10^{-8}	2.6010×10^{-6}	4.5634×10^{-14}	1.4080×10^{-12}	9.0605×10^{-14}	2.7954×10^{-12}

Table 2: Table Div

	$N = 10$		$N = 20$		$N = 30$	
	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$
a	1.7055×10^{-13}	9.4244×10^{-12}	4.7951×10^{-12}	2.6497×10^{-10}	2.5029×10^{-11}	1.3831×10^{-9}
b	1.3937×10^{-12}	7.7015×10^{-11}	4.4301×10^{-11}	2.4480×10^{-9}	1.2869×10^{-10}	7.1114×10^{-9}
c	7.7744×10^{-12}	4.2960×10^{-10}	3.8083×10^{-10}	2.1044×10^{-8}	3.8902×10^{-9}	2.1497×10^{-7}
d	1.4964×10^{-11}	8.2688×10^{-10}	5.5707×10^{-10}	3.0783×10^{-8}	1.6726×10^{-9}	9.2426×10^{-8}
e	4.2647×10^{-12}	2.3566×10^{-10}	1.0580×10^{-10}	5.8462×10^{-9}	1.0035×10^{-9}	5.5450×10^{-8}
f	1.4029×10^{-11}	7.7521×10^{-10}	5.8611×10^{-10}	3.2388×10^{-8}	3.0143×10^{-9}	1.6657×10^{-7}
g	9.4664×10^{-13}	5.2310×10^{-11}	3.5819×10^{-11}	1.9793×10^{-9}	1.4403×10^{-10}	7.9586×10^{-9}
h	5.3665×10^{-4}	1.6556×10^{-2}	1.0457×10^{-9}	3.2287×10^{-8}	1.7551×10^{-10}	5.4167×10^{-9}
i	4.6482×10^{-6}	1.4378×10^{-4}	5.0420×10^{-11}	1.5565×10^{-9}	2.3952×10^{-10}	7.3915×10^{-9}
j	5.3665×10^{-4}	1.6556×10^{-2}	1.0223×10^{-9}	3.1565×10^{-8}	7.2370×10^{-10}	2.2335×10^{-8}
k	3.3996×10^{-6}	1.0488×10^{-4}	1.0291×10^{-10}	3.1753×10^{-9}	4.0443×10^{-10}	1.2478×10^{-8}

Table 3: Table Lap

	$N = 10$		$N = 20$		$N = 30$	
	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$
a	9.9920×10^{-16}	1.1043×10^{-15}	1.1102×10^{-15}	1.2270×10^{-15}	1.9984×10^{-15}	2.2086×10^{-15}
b	8.8818×10^{-16}	9.8159×10^{-16}	1.5543×10^{-15}	1.7178×10^{-15}	1.8874×10^{-15}	2.0859×10^{-15}
c	8.8818×10^{-16}	9.8159×10^{-16}	1.6653×10^{-15}	1.8405×10^{-15}	2.2204×10^{-15}	2.4540×10^{-15}
d	8.8818×10^{-16}	9.8159×10^{-16}	1.9984×10^{-15}	2.2086×10^{-15}	2.6645×10^{-15}	2.9448×10^{-15}
e	8.8818×10^{-16}	9.8159×10^{-16}	1.4433×10^{-15}	1.5951×10^{-15}	2.6645×10^{-15}	2.9448×10^{-15}
f	1.2212×10^{-15}	1.3497×10^{-15}	1.6653×10^{-15}	1.8405×10^{-15}	2.8866×10^{-15}	3.1902×10^{-15}
g	8.8818×10^{-16}	9.8159×10^{-16}	1.8874×10^{-15}	2.0859×10^{-15}	2.5535×10^{-15}	2.8221×10^{-15}
h	4.6655×10^{-6}	2.8922×10^{-6}	7.9314×10^{-13}	4.9000×10^{-13}	2.6645×10^{-15}	1.6450×10^{-15}
i	1.1935×10^{-8}	7.3650×10^{-9}	3.3307×10^{-15}	2.0553×10^{-15}	3.1086×10^{-15}	1.9183×10^{-15}
j	4.6655×10^{-6}	2.8922×10^{-6}	7.9226×10^{-13}	4.8945×10^{-13}	4.2188×10^{-15}	2.6047×10^{-15}
k	7.7730×10^{-9}	4.8017×10^{-9}	2.2204×10^{-15}	1.3705×10^{-15}	4.4409×10^{-15}	2.7405×10^{-15}

Table 4: Table Interp

	$N = 10$		$N = 20$		$N = 30$	
	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$
b	2.3315×10^{-15}	2.5767×10^{-15}	2.3315×10^{-15}	2.5767×10^{-15}	3.2196×10^{-15}	3.5583×10^{-15}
c	2.4425×10^{-15}	2.6994×10^{-15}	2.8866×10^{-15}	3.1902×10^{-15}	2.5535×10^{-15}	2.8221×10^{-15}
d	2.3315×10^{-15}	2.5767×10^{-15}	3.2196×10^{-15}	3.5583×10^{-15}	3.3307×10^{-15}	3.6810×10^{-15}
e	2.3315×10^{-15}	2.5767×10^{-15}	2.6645×10^{-15}	2.9448×10^{-15}	2.8866×10^{-15}	3.1902×10^{-15}
f	2.2204×10^{-15}	2.4540×10^{-15}	2.7756×10^{-15}	3.0675×10^{-15}	3.2196×10^{-15}	3.5583×10^{-15}
g	2.6645×10^{-15}	2.9448×10^{-15}	2.9976×10^{-15}	3.3129×10^{-15}	3.6637×10^{-15}	4.0491×10^{-15}
i	1.1886×10^{-8}	7.3362×10^{-9}	5.1070×10^{-15}	3.1520×10^{-15}	6.4393×10^{-15}	3.9742×10^{-15}
j	4.8940×10^{-6}	3.0205×10^{-6}	8.1379×10^{-13}	5.0226×10^{-13}	4.6629×10^{-15}	2.8779×10^{-15}
k	7.8923×10^{-9}	4.8710×10^{-9}	5.1070×10^{-15}	3.1520×10^{-15}	5.1070×10^{-15}	3.1520×10^{-15}

Table 5: Table Interp Compare

	$N_1 = 20$	$N_1 = 20$	$N_1 = 5$	$N_1 = 10$	$N_1 = 30$	$N_1 = 20$
	$N_2 = 10$	$N_2 = 15$	$N_2 = 20$	$N_2 = 20$	$N_2 = 20$	$N_2 = 30$
h	2.8922×10^{-6}	5.1117×10^{-10}	1.5795×10^{-9}	4.8973×10^{-13}	4.9000×10^{-13}	1.5080×10^{-15}
i	7.3650×10^{-9}	1.6031×10^{-14}	1.5807×10^{-9}	1.2332×10^{-15}	1.9183×10^{-15}	2.4663×10^{-15}
j	2.8922×10^{-6}	5.1117×10^{-10}	5.1835×10^{-11}	4.8959×10^{-13}	4.8986×10^{-13}	1.9192×10^{-15}
k	4.8017×10^{-9}	7.6215×10^{-14}	1.5808×10^{-9}	1.2334×10^{-15}	2.4668×10^{-15}	2.0554×10^{-15}

Table 6: Table Comparing interpolation errors on wedges, when $N_1 \neq N_2$.

	$N = 10$		$N = 20$		$N = 30$	
	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$
b	9.1924×10^{-8}	5.5907×10^{-8}	7.5495×10^{-15}	4.5500×10^{-15}	8.2157×10^{-15}	4.9463×10^{-15}
c	9.2322×10^{-8}	5.5989×10^{-8}	8.8818×10^{-15}	5.3498×10^{-15}	1.3323×10^{-14}	8.0197×10^{-15}
d	9.2351×10^{-8}	5.6018×10^{-8}	8.8818×10^{-15}	5.3490×10^{-15}	1.7097×10^{-14}	1.0293×10^{-14}
e	1.8382×10^{-8}	1.1094×10^{-8}	9.1038×10^{-15}	5.4844×10^{-15}	1.3989×10^{-14}	8.4207×10^{-15}
f	1.4109×10^{-8}	8.5341×10^{-9}	8.2157×10^{-15}	4.9497×10^{-15}	1.9984×10^{-14}	1.2033×10^{-14}
g	1.9228×10^{-10}	1.1579×10^{-10}	9.7700×10^{-15}	5.8815×10^{-15}	1.5543×10^{-14}	9.3564×10^{-15}
i	3.2840×10^{-6}	1.3349×10^{-6}	1.7055×10^{-12}	6.9138×10^{-13}	1.5987×10^{-14}	6.4744×10^{-15}
j	1.6310×10^{-3}	6.6333×10^{-4}	5.0919×10^{-8}	2.0616×10^{-8}	8.9782×10^{-12}	3.6342×10^{-12}
k	2.7196×10^{-6}	1.1056×10^{-6}	1.7668×10^{-12}	7.1588×10^{-13}	2.9310×10^{-14}	1.1870×10^{-14}

Table 7: Table Conv

	$N = 10$		$N = 20$		$N = 30$	
	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$
b	$0.0000 \times 10^{+00}$	$0.0000 \times 10^{+00}$	4.4409×10^{-16}	1.4937×10^{-16}	8.8818×10^{-16}	2.9873×10^{-16}
c	$0.0000 \times 10^{+00}$	$0.0000 \times 10^{+00}$	4.4409×10^{-16}	1.4937×10^{-16}	1.3323×10^{-15}	4.4810×10^{-16}
d	$0.0000 \times 10^{+00}$	$0.0000 \times 10^{+00}$	4.4409×10^{-16}	1.4937×10^{-16}	8.8818×10^{-16}	2.9873×10^{-16}
e	$0.0000 \times 10^{+00}$	$0.0000 \times 10^{+00}$	8.8818×10^{-16}	2.9873×10^{-16}	1.3323×10^{-15}	4.4810×10^{-16}
f	$0.0000 \times 10^{+00}$	$0.0000 \times 10^{+00}$	$0.0000 \times 10^{+00}$	$0.0000 \times 10^{+00}$	8.8818×10^{-16}	2.9873×10^{-16}
g	$0.0000 \times 10^{+00}$	$0.0000 \times 10^{+00}$	4.4409×10^{-16}	1.4937×10^{-16}	8.8818×10^{-16}	2.9873×10^{-16}
i	3.9603×10^{-11}	6.1859×10^{-12}	8.8818×10^{-16}	1.3873×10^{-16}	3.5527×10^{-15}	5.5493×10^{-16}
j	2.4085×10^{-8}	3.7621×10^{-9}	1.7764×10^{-15}	2.7746×10^{-16}	2.6645×10^{-15}	4.1620×10^{-16}
k	2.7334×10^{-11}	4.2695×10^{-12}	1.7764×10^{-15}	2.7746×10^{-16}	1.7764×10^{-15}	2.7746×10^{-16}

Table 8: Table Int

[8]

$$\rho = \exp(\alpha t + \beta_1 y_1 + \beta_2 y_2) \tag{14}$$
$$\mathbf{v} = \left(\beta_1 - \frac{\alpha}{2\beta_1} + p_1 \exp(-\beta_1 y_1), \beta_2 - \frac{\alpha}{2\beta_2} + p_2 \exp(-\beta_2 y_2)\right),$$

where $\beta_1 = 0.1$, $\beta_2 = 0.1$, $\alpha = -0.5$, $p_1 = -1$ and $p_2 = 1$. We compare the exact solution on a box of dimensions $[0, 2] \times [0, 2]$ with different discretizations of the box using multishape, signified by letters (a) to (g), see Figure 1. Each of the shapes are discretized with $N = 10$, $N = 20$ and $N = 30$ points in each spatial direction, which means that the dissected box has more points in total than the original box. The ODE solver tolerances are 10^{-9}. The solution can be seen in Figure 4. The question is whether the results of the PDE on the box and the different discretizations of the box have a similar error when compared to the exact solution. The absolute and relative errors, $\mathcal{E}_{Abs}$ and $\mathcal{E}_{Rel}$, are measured in an l_2 norm in space and an l_∞ norm in time and are displayed in Table 9. The errors for the non-rectangular quadrilateral, which is shown in Figure 2, are displayed on the row with the letter (q).

The same test can be done for a wedge. Here, a single wedge and discretized versions are considered, denoted by (h) to (k), see Figure 3. Table 9 also shows the errors measured against the exact solution for different discretizations of the wedge, for $N = 10$, $N = 20$ and $N = 30$. The solution can be seen in Figure 5.

Next the advection diffusion equation is solved on a multishape which is composed of four quadrilaterals, see Figure 6. The same is done for a second example involving a wedge, see Figure 7. The errors, as compared to the exact solution (14), are displayed in Table 10. There, the first multishape is denoted ms1 and the second ms2.

	$N = 10$		$N = 20$		$N = 30$	
	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$
a	2.5869×10^{-7}	1.0894×10^{-9}	2.2063×10^{-7}	1.1235×10^{-9}	2.1913×10^{-7}	1.1159×10^{-9}
b	3.4991×10^{-7}	1.1308×10^{-9}	3.2073×10^{-7}	1.1377×10^{-9}	3.1877×10^{-7}	1.1307×10^{-9}
c	4.1386×10^{-7}	1.1596×10^{-9}	3.9107×10^{-7}	1.1500×10^{-9}	3.8858×10^{-7}	1.1426×10^{-9}
d	4.2622×10^{-7}	1.0268×10^{-9}	3.9019×10^{-7}	1.0026×10^{-9}	3.8768×10^{-7}	9.9616×10^{-10}
e	4.4595×10^{-7}	1.0704×10^{-9}	3.3852×10^{-7}	9.8259×10^{-10}	3.3718×10^{-7}	1.0085×10^{-9}
f	4.1985×10^{-7}	1.0275×10^{-9}	4.2547×10^{-7}	1.0412×10^{-9}	4.2291×10^{-7}	1.0350×10^{-9}
g	5.0161×10^{-7}	1.1254×10^{-9}	4.4470×10^{-7}	1.1323×10^{-9}	4.3978×10^{-7}	1.1198×10^{-9}
q	2.4145×10^{-7}	1.0389×10^{-9}	2.2844×10^{-7}	1.1160×10^{-9}	2.2505×10^{-7}	1.0995×10^{-9}
h	1.8507×10^{-3}	4.4410×10^{-6}	3.1141×10^{-7}	7.4763×10^{-10}	2.7613×10^{-7}	7.5178×10^{-10}
i	3.4678×10^{-6}	6.5183×10^{-9}	3.9525×10^{-7}	7.5983×10^{-10}	3.8485×10^{-7}	7.4376×10^{-10}
j	2.6220×10^{-3}	4.4489×10^{-6}	4.2335×10^{-7}	7.4943×10^{-10}	3.8922×10^{-7}	7.5064×10^{-10}
k	2.6814×10^{-6}	4.0877×10^{-9}	4.4144×10^{-7}	7.0896×10^{-10}	4.3211×10^{-7}	6.9682×10^{-10}

Table 9: Table Dirichlet Exact Solution 1

	$N = 10$		$N = 20$		$N = 30$	
	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$
ms1	5.9588×10^{-7}	1.3030×10^{-9}	6.1246×10^{-7}	1.3259×10^{-9}	6.0034×10^{-7}	1.2997×10^{-9}
ms2	1.6829×10^{-3}	3.6591×10^{-6}	3.7239×10^{-7}	8.9375×10^{-10}	3.7589×10^{-7}	8.9532×10^{-10}

Table 10: Table Dirichlet Exact Solution 1 on two multishapes

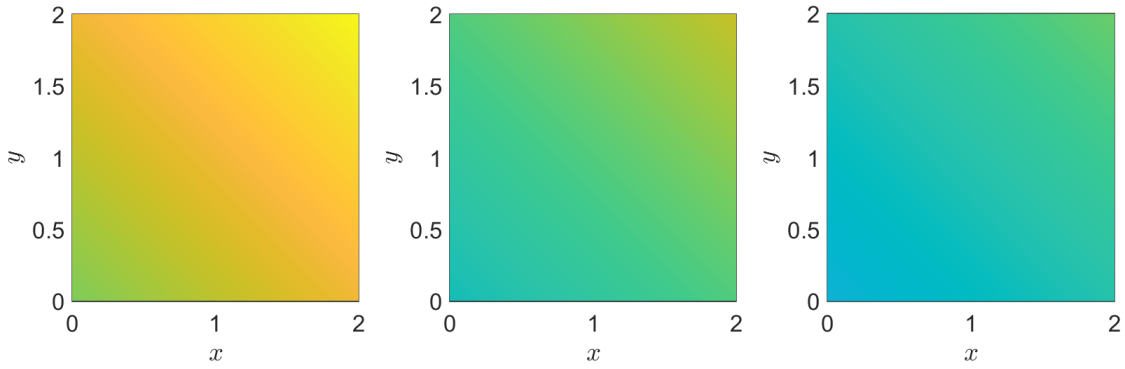

Figure 4: Exact solution on the box.

4.2.2 Exact Tests - Dirichlet Conditions, Solution 2

We solve an exact Dirichlet problem which does not have an exponential form but a quadratic form. In order to do this, we add on a source term f to the advection-diffusion equation. We define the exact solution as

$$\rho = t y_1^2 y_2^2,$$
$$f = y_1^2 y_2^2 - 2y_1^2 t - 2y_2^2 t + 2y_1 y_2^2 t,$$

and with a velocity field of strength one acting in the y_1 direction. We run this on the box and the wedge discretizations (a) to (g) and the quadrilateral (q). The results are displayed in Table 11.

4.2.3 Exact Tests - No-Flux Conditions

We consider an exact solution on a box with no-flux boundary conditions for the advection diffusion equation. The solution is

$$\rho = 2 + \exp(-(\mu_1^2 + \mu_2^2)t)\cos(\mu_1 y_1)\cos(\mu_2 y_2),$$

where $\mu_1 = n\pi/L_1$, $\mu_2 = n\pi/L_2$ and $n = 2$, $L_1 = d - c$, $L_2 = b - a$ for a domain $[a, b] \times [c, d]$. We use the discretizations of the box displayed in Figure 1 with $N = 10$, $N = 20$ and $N = 30$. The exact solution for this problem can be seen in Figure 8. In Table 12 we can see the errors

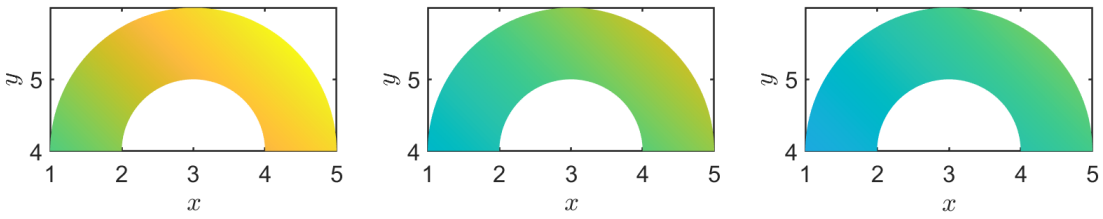

Figure 5: Exact solution on the wedge.

made on the different discretizations. We can observe that the errors in multishapes (c), (d) and (f) are considerably larger than the errors for the other shapes. This is because of the complex discretization chosen in these multishapes. Due to this discretization, the standard averaging of the two normals at an intersection corner does not result in reasonable normals. This becomes an issue when computing no-flux boundary conditions, since this requires the normal information. The code library offers the option of overriding individual normals. We redefine the normals at the intersection corner to be the outward normals of the box. This improves the errors greatly as can be seen in Table 13. The change in normals for these three multishapes is shown in Figure 9.

4.3 Forward Problems on multishapes

We first consider a forward problem on the different discretizations of the box, see Figure 1. We compare the results of the discretized boxes with the result for box (a) with large N. We choose the initial condition for ρ to be

$$\rho_0 = \exp(-2((y_1 - 0.7)^2 + (y_2 - 0.2)^2)),$$

and impose a constant flow of strength 0.8 acting upward. We choose $N = 10$, $N = 20$ and $N = 30$ as before. The errors are displayed in Table **??** and the result can be seen in Figure 10.

	$N = 10$		$N = 20$		$N = 30$	
	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$
a	2.2747×10^{-13}	3.9799×10^{-16}	3.9207×10^{-13}	6.3023×10^{-16}	4.7291×10^{-13}	7.5978×10^{-16}
b	1.1317×10^{-12}	1.1097×10^{-15}	6.1443×10^{-13}	5.9848×10^{-16}	2.5646×10^{-12}	2.5682×10^{-15}
c	7.0072×10^{-13}	7.0033×10^{-16}	1.4877×10^{-12}	1.5185×10^{-15}	2.0691×10^{-12}	2.4102×10^{-15}
d	6.8357×10^{-13}	5.1274×10^{-16}	1.4899×10^{-12}	2.9801×10^{-15}	2.6521×10^{-12}	2.0667×10^{-15}
e	2.2548×10^{-12}	6.7534×10^{-15}	5.5358×10^{-13}	6.3464×10^{-16}	8.8186×10^{-13}	2.3179×10^{-15}
f	8.1603×10^{-13}	1.5216×10^{-15}	1.4019×10^{-12}	8.7680×10^{-16}	3.3732×10^{-12}	1.8904×10^{-15}
g	4.9611×10^{-13}	1.9913×10^{-15}	5.0195×10^{-12}	8.5065×10^{-15}	1.4727×10^{-12}	1.1481×10^{-15}
q	6.0563×10^{-13}	1.7871×10^{-15}	9.9468×10^{-13}	4.3977×10^{-15}	1.1245×10^{-12}	3.2386×10^{-15}
h	$9.2590 \times 10^{+00}$	1.5820×10^{-4}	3.6221×10^{-6}	6.1886×10^{-11}	5.9042×10^{-11}	1.0457×10^{-15}
i	2.6170×10^{-2}	3.1608×10^{-7}	5.9610×10^{-11}	7.4688×10^{-16}	8.2340×10^{-11}	1.0439×10^{-15}
j	$1.3082 \times 10^{+01}$	1.5808×10^{-4}	5.1279×10^{-6}	6.1963×10^{-11}	2.1720×10^{-10}	2.7449×10^{-15}
k	1.9298×10^{-2}	2.0145×10^{-7}	6.8410×10^{-11}	7.4568×10^{-16}	9.9529×10^{-11}	1.0842×10^{-15}

Table 11: Table Dirichlet Exact Solution 2

	$N = 10$		$N = 20$		$N = 30$	
	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$
a	1.0458×10^{-2}	2.5230×10^{-5}	2.8350×10^{-7}	6.8623×10^{-10}	2.6819×10^{-7}	6.4918×10^{-10}
b	1.0178×10^{-2}	1.7387×10^{-5}	3.5488×10^{-7}	6.0816×10^{-10}	3.5538×10^{-7}	6.0901×10^{-10}
c	7.1323×10^{-2}	9.8994×10^{-5}	7.0449×10^{-3}	9.7842×10^{-6}	2.6254×10^{-3}	3.6463×10^{-6}
d	8.4906×10^{-2}	1.0217×10^{-4}	8.2570×10^{-3}	9.9356×10^{-6}	3.0338×10^{-3}	3.6506×10^{-6}
e	5.6960×10^{-3}	8.1264×10^{-6}	4.3060×10^{-7}	6.0669×10^{-10}	4.3142×10^{-7}	6.0784×10^{-10}
f	1.2934×10^{-1}	1.5474×10^{-4}	1.3465×10^{-2}	1.6099×10^{-5}	4.0715×10^{-3}	4.8753×10^{-6}
g	8.7462×10^{-6}	1.0753×10^{-8}	5.1501×10^{-7}	6.2332×10^{-10}	5.1469×10^{-7}	6.2294×10^{-10}

Table 12: Table No-Flux Exact Solution

	$N = 10$		$N = 20$		$N = 30$	
	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$	$\mathcal{E}_{Abs}$	$\mathcal{E}_{Rel}$
c	1.8668×10^{-2}	2.5911×10^{-5}	4.5961×10^{-7}	6.4023×10^{-10}	4.5881×10^{-7}	6.3911×10^{-10}
d	2.4277×10^{-2}	2.9194×10^{-5}	5.2874×10^{-7}	6.3810×10^{-10}	5.2815×10^{-7}	6.3740×10^{-10}
f	2.1887×10^{-3}	2.6634×10^{-6}	5.3703×10^{-7}	6.3964×10^{-10}	5.3681×10^{-7}	6.3938×10^{-10}

Table 13: Table No-Flux Exact Solution with Corrected Normal Vectors at Intersections

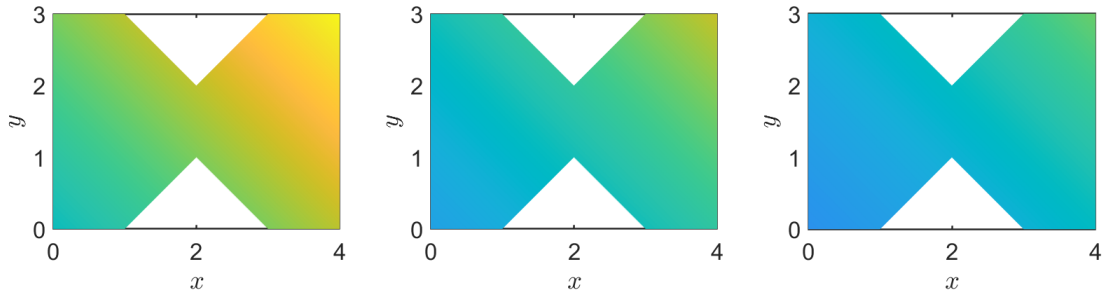

Figure 6: Example 1 multishape

We follow the same idea, but now consider the wedge discretizations, see Figure 3. We choose the initial condition for ρ to be

$$\rho_0 = \exp(-2((y_1 - 1.5)^2 + (y_2 - 4.5)^2)),$$

and impose a constant flow of strength 3 acting from left to right, along the angular direction. We choose $N = 20$ and $N = 30$. The errors are displayed in Table **??** and the result can be seen in Figure 11.

Finally, two more complex multishape examples are considered. The first of these examples is solving an advection diffusion problem on a multishape consisting of two quadrilaterals and two wedges, with constant velocity of strength ten. The initial condition for this problem is:

$$\rho_0 = \exp(-2(y_1 - 0.5)^2 - 2(y_2 + 1)^2).$$

The result, evaluated for $N = 20$ on each shape, can be seen in Figure 12.

In a second example, the velocity is of strength 5 and the initial condition is:

$$\rho_0 = \exp(-2(y_1 - 0.5)^2 - 2(y_2 - 1.5)^2).$$

The result, which is computed on a multishape made up of four quadrilaterals into a channel, can be seen in Figure 13.

In Table **??** the solution to these two examples is evaluated with $N = 50$ and compared to the solution evaluated with $N = 10$, $N = 20$ and $N = 30$, using the same error measure as before.

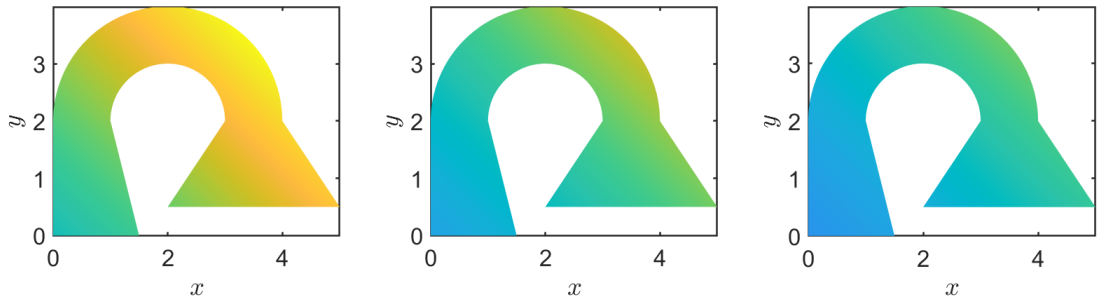

Figure 7: Example 2 multishape

References

[1] Lloyd N. Trefethen. *Spectral Methods in Matlab*. SIAM, 2000.

[2] Andreas Nold, Benjamin D. Goddard, Peter Yatsyshin, Nikos Savva, and Serafim Kalliadasis. Pseudospectral methods for density functional theory in bounded and unbounded domains. *CoRR*, abs/1701.06182, 2017.

[3] John P. Boyd. *Chebyshev and Fourier Spectral Methods*. Dover Publications, Inc, 2000.

[4] Jean-Paul Berrut and Lloyd N. Trefethen. Barycentric Lagrange interpolation. *SIAM Review*, 46(3):501–517, 2004.

[5] Charles W. Clenshaw and A. R. Curtis. A method for numerical integration on an automatic computer. *Numerische Mathematik*, 2(1):197–205, 1960.

[6] Anthony T Patera. A spectral element method for fluid dynamics: Laminar flow in a channel expansion. *Journal of Computational Physics*, 54(3):468–488, 1984.

[7] Dimitri Komatitsch and Jean-Pierre Vilotte. The spectral element method: an efficient tool to simulate the seismic response of 2d and 3d geological structures. *Bulletin of the Seismological Society of America*, 88:368–392, 04 1998.

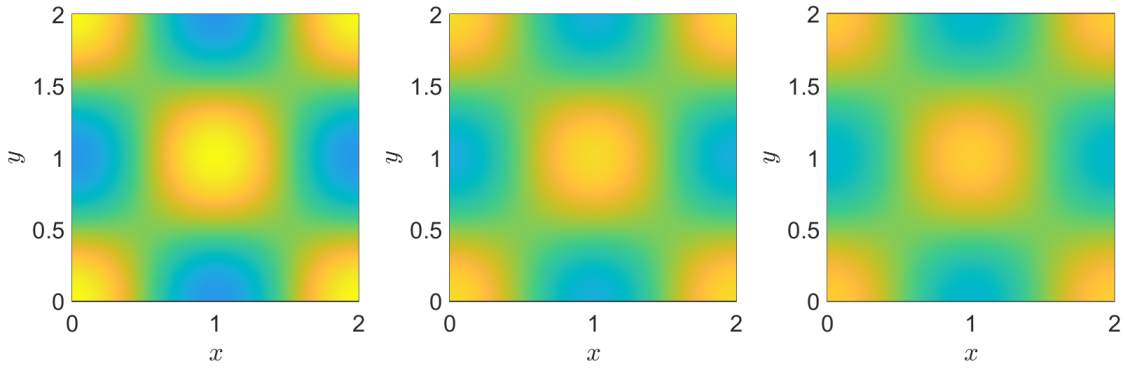

Figure 8: Exact solution on the box with no-flux boundary conditions.

[8] G D Hutomo, J Kusuma, A Ribal, A G Mahie, and N Aris. Numerical solution of 2-d advection-diffusion equation with variable coefficient using du-fort frankel method. *Journal of Physics: Conference Series*, 1180:012009, feb 2019.

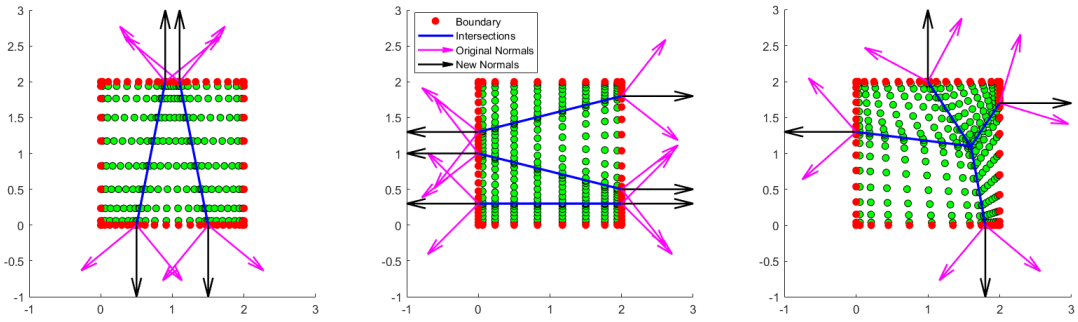

Figure 9: Original, non-sensible normals are compared to new, customized normals.

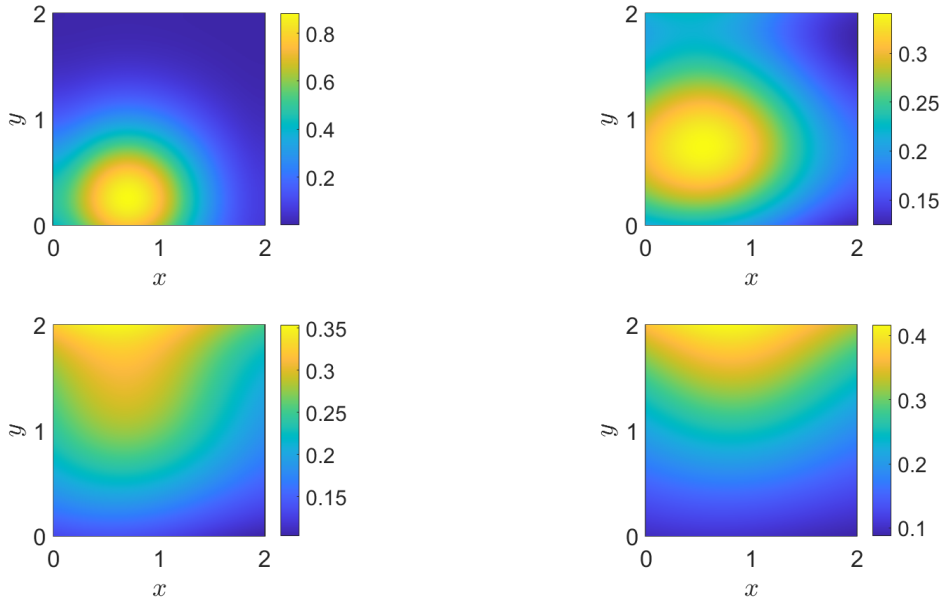

Figure 10: Forward Example on box discretizations

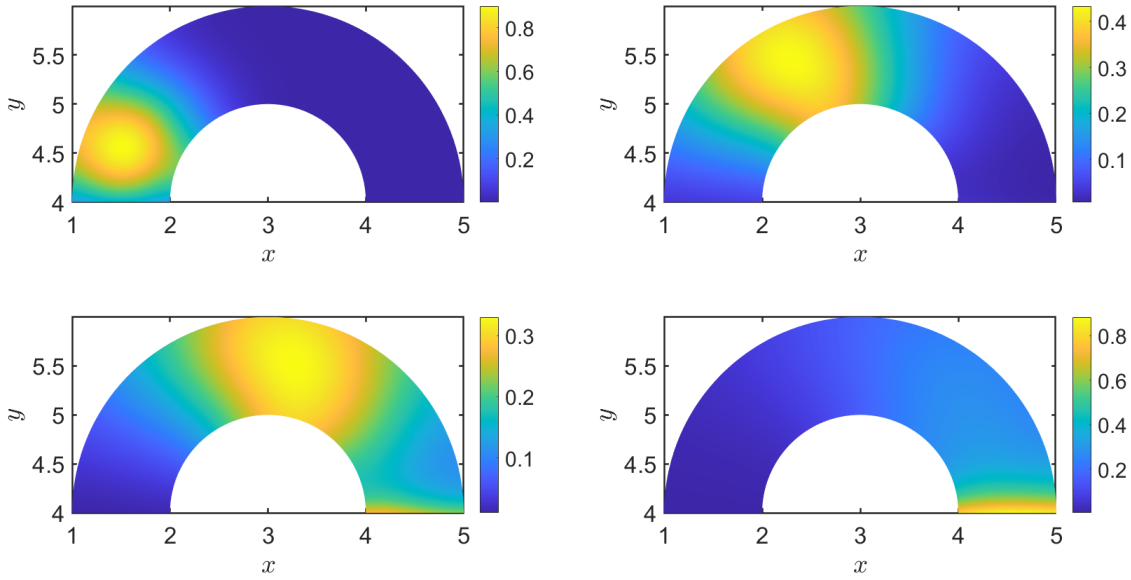

Figure 11: Forward Example on wedge discretizations

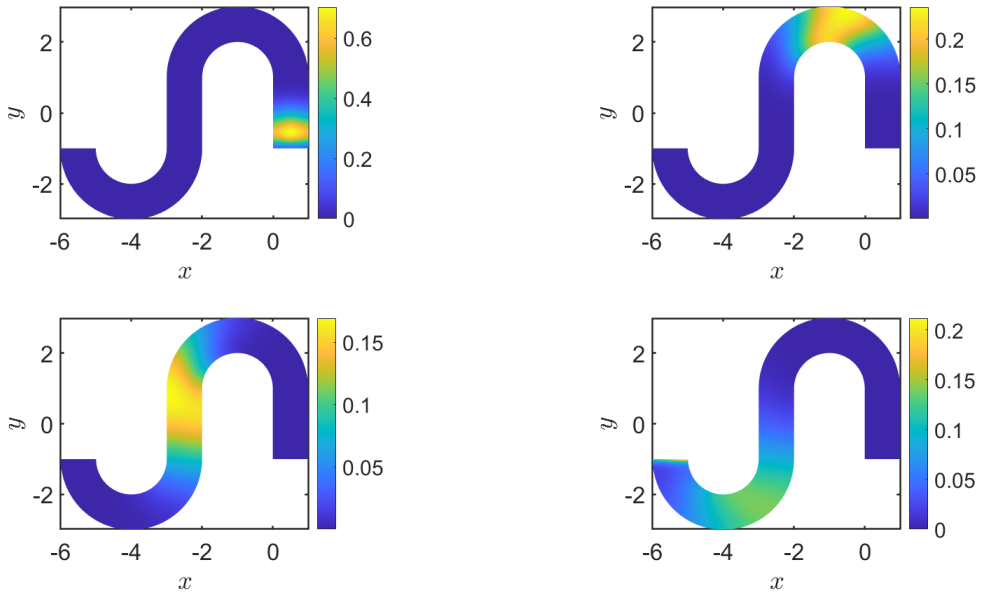

Figure 12: Forward Problem 1, different colour scale for each plot to highlight particle mass location

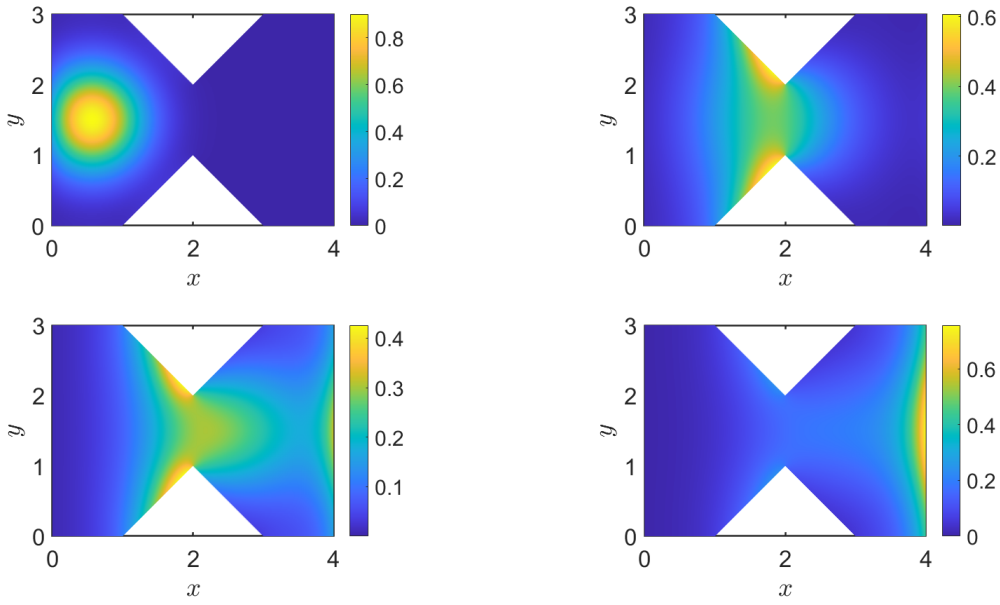

Figure 13: Forward Problem 2, different colour scale for each plot to highlight particle mass location