Explicit Jump Immersed Interface Method: Documentation for 2D Poisson Code

V. Rutka   A. Wiegmann

November 25, 2005

Abstract

The Explicit Jump Immersed Interface method is a powerful tool to solve elliptic pde with singular source terms, in complex domains, or with discontinuous coefficients. Examples include 2d Poisson problems, 2d and 3d linear elasticity and 2d Stokes to name a few. The power of the EJIIM lies in the fact that not grid generation is needed. EJIIM works on a uniform Cartesian grid and uses additionally some information about the boundary or interface location such as intersections with grid lines, normals and curvatures at these points, etc. On the other hand, the implementation of the EJIIM is rather involved. To make it easier to overcome the initial hurdle of using it, we provide matlab code and this documentation that contains detailed explanations of this code for two dimensional Poisson boundary value problems. We have tried to keep the notation as close to the sources as possible. For details of the method see [4, 5, 1]

Keywords: Immersed Interface Methods (IIM), source code, elliptic partial differential equations, finite differences, regular grid

1 Problem statement

To solve is the following Poisson boundary value problem

$$\begin{cases} \Delta u = f & \text{in } \Omega \subset \mathbb{R}^2 \\ u = u_D & \text{on } \partial\Omega_D \quad \text{(Dirichlet boundary)} \\ \partial_n u = u_N & \text{on } \partial\Omega_N \quad \text{(Neumann boundary)} \end{cases} \tag{1}$$

In the actual version of the code the following parameters are set:

- $f(x,y) = x^2 + y$ (set in function ASSEMBLE/set_rhs.m)

  Attention: if you are changing the form of the right hand side function f (stored in variable `F`), then you have to change manually also the *jump* in the right hand side, stored in variable `jf`.

- Ω can have two shapes: 'circle' or 'flower'.
  - 'Circle' is given by

    $$(x - x_c)^2 + (y - y_c)^2 \leq R^2$$

    with some parameters x_c, y_c and R.
  - The boundary of the domain 'flower' is given by

    $$\begin{cases} x &= 0.55(R + 0.15\sin(5\theta))\cos\theta + x_c \\ y &= 0.55(R + 0.15\sin(5\theta))\sin\theta + y_c \end{cases}, \quad \theta \in [0, 2\pi)\,.$$

  Parameters x_c, y_c and R are given in ASSEMBLE/problem_setup.m.

- $\partial\Omega$ is splitted into two distinct parts:

  $$\begin{aligned} \partial\Omega_D &:= \{(x,y) \in \partial\Omega \mid x < x_{level}\} \\ \partial\Omega_N &:= \partial\Omega \backslash \partial\Omega_D \end{aligned}$$

  Parameter x_{level} is set in ASSEMBLE/problem_setup.m.

  Points, where the boundary of Ω cuts the grid lines are called *intersection points*. In code they are always stored in structure variables `IX`. `IX_D` contains intersection points on $\partial\Omega_D$ and `IX_N` – the intersection points on $\partial\Omega_N$. Separation of these two sets of intersection points is done in ASSEMBLE/separate_boundaries.m.

- $u_D = x^2$ and $u_N = n_1 + n_2$. The Dirichlet boundary condition function u_D is stored in variable `uD`, the Neumann boundary condition is stored in `uN`. (Function ASSEMBLE/set_rhs.m)

2 EJIIM idea

2.1 Embedding into a rectangular domain

The original domain Ω is embedded in a rectangle $\Omega^* := (a_x, b_x) \times (a_y, b_y)$. Parameters a_x, b_x, a_y and b_y are given in ASSEMBLE/problem_setup.m.

After embedding, the original domain Ω becomes a *'−' phase* Ω^- and the rest is typically denoted by *'+' phase* $\Omega^+ := \Omega^* \backslash (\overline{\Omega^-})$. The boundary $\partial\Omega$ becomes an *interface* and boundary conditions turn into *jump conditions.* The extension is done by zero in Ω^+, thus, on $\partial\Omega^*$ homogeneous Dirichlet boundary conditions can be imposed.

Attention: The embedding rectangle has to be selected in such a way that there is "enough" space between $\partial\Omega$ and $\partial\Omega^*$. "Enough" depends on the mesh width used in discretisation. In any case, there should be at least several layers of grid points between $\partial\Omega$ and $\partial\Omega^*$.

2.2 Discretisation in a rectangle

Over Ω^* a regular grid with mesh widthes h_x and h_y in x and y directions respectively is imposed. In code they are denoted by `hx` and `hy`. n_x and n_y, in code `nx` and `ny` (ASSEMBLE/problem_setup.m) are the number of grid points in x and y directions. Important is to note that number of grid points should be counted in Matlab convention, it is, starting from 1 and not from 0.

Due to the simplified geometry pre-processor which is used in the online version of the code, the parameters n_x and n_y have to be selected such that h_x and h_y are equal. This restriction can be removed without any further changes if the geometry pre-processor is replaced.

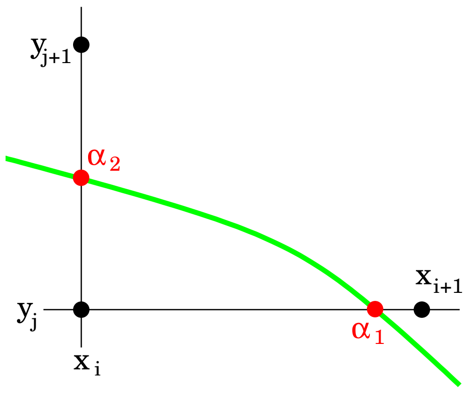
The Δ operator of (1) is discretised in Ω^* by *standard central finite differences*:

$$\Delta u \approx \frac{1}{h_x^2}\left(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}\right) + \frac{1}{h_y^2}\left(u_{i,j+1} - 2u_{i,j} + u_{i,j-1}\right) . \qquad (2)$$

The resulting standard finite difference matrix is denoted by $\mathbf{A}$, is stored in variable `A` and computed by function DIFFOP/sysmatrix_poisson.m.

2.3 Correction terms

The points where the 5-point stencil is cut by the interface are called *irregular points.* There, the approximation (2) has to be corrected by adding so called *correction terms.* They can be written for each of the appearing derivatives in (1) separately.



2.3.1 Correcting the second order differences

Consider a situation like in figure above and let $(x_{\alpha 1},\, y_{\alpha 1})$ be the coordinates of the intersection point α_1 and $(x_{\alpha 2},\, y_{\alpha 2})$: coordinates of the intersection point α_2. Let $h^+ := x_{i+1} - x_{\alpha 1}$, $h^- := x_i - x_{\alpha 1}$, $k^+ := y_{j+1} - y_{\alpha 2}$, $k^- := y_j - y_{\alpha 2}$. Then, the corrected differences of second order derivatives are

$$\begin{aligned}
\partial_{xx} u(x_i, y_j) &\approx \frac{1}{h_x^2}\left(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}\right) \\
&\quad - \frac{1}{h_x^2}\left([u]_{\alpha_1} + h^+[\partial_x u]_{\alpha_1} + \frac{1}{2}(h^+)^2[\partial_{xx} u]_{\alpha_1}\right) \qquad (3)\\
\partial_{xx} u(x_{i+1}, y_j) &\approx \frac{1}{h_x^2}\left(u_{i+2,j} - 2u_{i+1,j} + u_{i,j}\right) \\
&\quad + \frac{1}{h_x^2}\left([u]_{\alpha_1} + h^-[\partial_x u]_{\alpha_1} + \frac{1}{2}(h^-)^2[\partial_{xx} u]_{\alpha_1}\right) \qquad (4)
\end{aligned}$$

for derivatives in x-direction and similarly for the y-derivatives

$$\begin{aligned}
\partial_{yy} u(x_i, y_j) &\approx \frac{1}{h_y^2}\left(u_{i,j+1} - 2u_{i,j} + u_{i,j-1}\right) \\
&\quad - \frac{1}{h_y^2}\left([u]_{\alpha_2} + k^+[\partial_y u]_{\alpha_2} + \frac{1}{2}(k^+)^2[\partial_{yy} u]_{\alpha_2}\right) \qquad (5)\\
\partial_{yy} u(x_i, y_{j+1}) &\approx \frac{1}{h_y^2}\left(u_{i,j+2} - 2u_{i,j+1} + u_{i,j}\right) \\
&\quad + \frac{1}{h_y^2}\left([u]_{\alpha_2} + k^-[\partial_y u]_{\alpha_2} + \frac{1}{2}(k^-)^2[\partial_{yy} u]_{\alpha_2}\right) \qquad (6)
\end{aligned}$$

Note, we have introduced *new variables* into a system. At each intersection point we have 6 new unknowns: $[u]$, $[\partial_x u]$, $[\partial_y u]$, $[\partial_{xx} u]$, $[\partial_{yy} u]$ and $[\partial_{xy}]$, where in the case of Poisson equation some of them are redundant.

In general, jump in the mixed derivative is not used in expressions (3–6) and jumps in x-derivatives are not used at y-intersections, as well as jumps in y-derivatives are not used in x-intersections.

All jumps are stored in one supervector J (see ASSEMBLE/run_ejiim.m, variable `J`) with ordering

$$J = ([u]_{\alpha_1}, [\partial_x u]_{\alpha_1}, [\partial_y u]_{\alpha_1}, [\partial_{xx} u]_{\alpha_1}, [\partial_{yy} u]_{\alpha_1}, [\partial_{xy} u]_{\alpha_1}, [u]_{\alpha_2}, \ldots, [\partial_{xy} u]_{\alpha_l})$$

where l is the number of intersection points $\alpha_1, \alpha_2, \ldots, \alpha_l$. Coefficients of J in the correction terms (3–6) are stored in matrix $\mathbf{\Psi}$, in code denoted by `P` (see ASSEMBLE/run_ejiim.m).

With this, approximation of the differential operator Δ in domain Ω^* can be written as

$$\mathbf{A}U + \mathbf{\Psi}J = F \tag{7}$$

2.3.2 Details of constructing the P matrix

Matrix $\mathbf{\Psi}$, in code stored in variable `P` is constructed of two parts

$$\mathbf{\Psi} = (\mathbf{\Psi}_D \ , \ \mathbf{\Psi}_N) \ ,$$

where $\mathbf{\Psi}_D$ corresponds to intersection points belonging to $\partial\Omega_D$ and $\mathbf{\Psi}_N$ – to intersection points along $\partial\Omega_N$.

Each of $\mathbf{\Psi}_D$ and $\mathbf{\Psi}_D$ is constructed as sum

$$\begin{aligned} \mathbf{\Psi}_D &= \mathbf{\Psi}_D^{xx} + \mathbf{\Psi}_D^{yy} \\ \mathbf{\Psi}_N &= \mathbf{\Psi}_N^{xx} + \mathbf{\Psi}_N^{yy} \end{aligned}$$

where $\mathbf{\Psi}^{xx}$ corrects the approximation of ∂_{xx} derivative according to (3,4) and $\mathbf{\Psi}^{yy}$ corrects the approximation of ∂_{yy} derivative according to (5,6). See ASSEMBLE/run_ejiim.m.

Note: Matrix Ψ depends only on the differential operator and geometry. No dependence on boundary conditions!

Construction of the matrix $\mathbf{\Psi}$ is done in function DIFFOP/corrections.m. The interface information is stored in the structure variable `IX`, see Section 3 for details.

Constructing of $\mathbf{\Psi}^{xx}$ matrix

Idea: run a loop over all intersection points and check, which grid points are affected.

We know that ∂_{xx} derivative is affected only by x-type intersections, so the matrix $\mathbf{\Psi}^{xx}$ has zeros in the right block, corresponding to y-type intersection points.

In a loop over all x-intersections:

1. Determine the coordinates of the anchor point, call it $(x_i,\ y_j)$.
2. Check if the interface lies left or right from the anchor point.
   - If interface lies LEFT from the anchor, use (4) at point (x_i, y_j) and (3) at (x_{i-1}, y_j).
   - If interface lies RIGHT from the anchor, use (3) at the point (x_i, y_j) and (4) at (x_{i+1}, y_j)

Construction of $\mathbf{\Psi}^{yy}$ is done completely analogously, only now zeros are in the left block of $\mathbf{\Psi}^{yy}$ and loop has to be done over all intersections.

In the actual version of the code, the EJIIM system is solved by Matlab '\'-operator. For larger problems, especially three dimensional, the Schur-complement approach together with some fast solver for inverting the operator $\mathbf{A}$ is highly suggested [3, 1, 2].

2.4 Additional equations for jumps

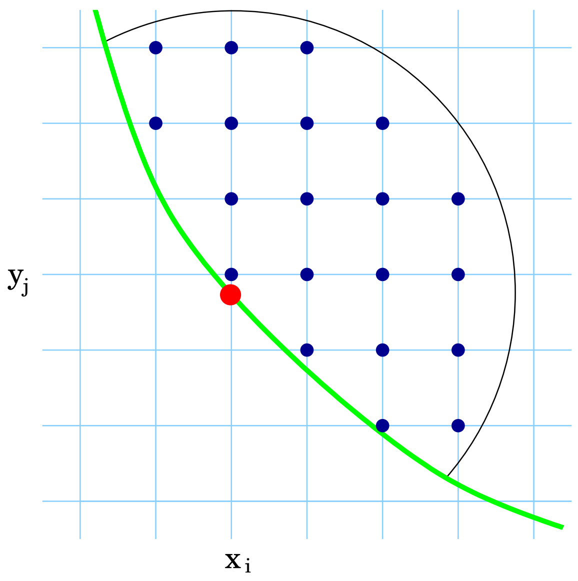
At first, jumps are expressed in terms of known quantities, like boundary conditions, and one sided derivatives. Then, for jumps at arbitrary intersection point we can write equations

- Dirichlet b.c.

$$\begin{array}{rcrcl} & & [u] & = & -u_D \\ \partial_x u^- & + & [\partial_x u] & = & 0 \\ \partial_y u^- & + & [\partial_y u] & = & 0 \\ \partial_{xx} u^- & + & [\partial_{xx} u] & = & 0 \\ \partial_{yy} u^- & + & [\partial_{yy} u] & = & 0 \\ \partial_{xy} u^- & + & [\partial_{xy} u] & = & 0 \end{array} \tag{8}$$

- Neumann b.c.

$$\begin{array}{rcrcl} u^- & + & [u] & = & 0 \\ t_1^2 \partial_x u^- + t_1 t_2 \partial_y u^- & + & [\partial_x u] & = & -n_1 u_N \\ t_1 t_2 \partial_x u^- + t_2^2 \partial_y u^- & + & [\partial_y u] & = & -n_2 u_N \\ \partial_{xx} u^- & + & [\partial_{xx} u] & = & 0 \\ \partial_{yy} u^- & + & [\partial_{yy} u] & = & 0 \\ \partial_{xy} u^- & + & [\partial_{xy} u] & = & 0 \end{array} \tag{9}$$



Note that the one sided derivatives depend on the *unknown* solution. Our goal is to approximate them using a least squares fit of a quadratic polynomial.

The first step is to select the stencil for least squares interpolation. We follow the methodology from [3]. It is done by selecting some radius $k \in \mathbb{N}$ and then taking all grid points of Ω^- inside circle with radius $k\sqrt{h_x h_y}$. (Line 73 in DIF-FOP/d_matrix.m.) With l we denote the stencil cardinality.

For each stencil point we try to fit a quadratic polynomial:

$$u(x_i, y_j) \approx p(x_i, y_j) := p_1 + p_2 h_i + p_3 k_j + p_4 h_i^2 + p_5 k_j^2 + p_6 h_i k_j \,,$$

where $h_i := x_i - x_\alpha$ and $k_j := y_j - y_\alpha$. With this a weighted least squares problem is obtained:

$$\sum_{(x_i,y_j)\in \text{ stencil}} w_{i,j}^2 \left(p(x_i, y_j) - u_{i,j}\right)^2 \xrightarrow[P]{} \min$$

where $P := (p_1, p_2, p_3, p_4, p_5, p_6)^\top$. In the code the following weights are used:

$$w_{i,j} = \left(1 + d(x_i, y_j)/h_x\right)^{-1} \quad , \quad d(x_i, y_j) := \sqrt{h_i^2 + k_j^2}\,.$$

For a given grid function U, coefficients of the polynomial are given by

$$P = (M^\top W^2 M)^{-1}(M^\top W^2 R U)\,,$$

where

$$W := \mathrm{diag}(w_{i_1,j_1}, w_{i_2,j_2}, \ldots, w_{i_l,j_l})$$

and

$$M := \begin{pmatrix} \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & h_i & k_j & h_i^2 & k_j^2 & h_i k_j \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

(in code this is the matrix `MM` in DIFFOP/d_matrix.m.)

Matrix R is the restriction operator, RU restricts the vector U defined at all grid points to its values only at the stencil points. R is possible to write explicitely, however, we have found that this costs enormous computational time and because of this reason in code the matrix W^2R is directly computed (matrix `W2R` in DIFFOP/d_matrix.m).

Note that

$$\begin{array}{rcl} u(x_\alpha, y_\alpha) & \approx & p_1 \\ \partial_x u(x_\alpha, y_\alpha) & \approx & p_2 \\ \partial_y u(x_\alpha, y_\alpha) & \approx & p_3 \\ \partial_{xx} u(x_\alpha, y_\alpha) & \approx & 2p_4 \\ \partial_{yy} u(x_\alpha, y_\alpha) & \approx & 2p_5 \\ \partial_{xy} u(x_\alpha, y_\alpha) & \approx & p_6 \end{array}$$

and this allows us to write the corresponding one-sided derivatives as

$$(u^-, \partial_x u^-, \partial_y u^-, \partial_{xx} u^-, \partial_{yy} u^-, \partial_{xy} u^-)^\top = BU\,,$$

where

$$B := S(M\top W^2 M)^{-1}(M^\top W^2 R)\,.$$

In DIFFOP/d_matrix.m to matrix B corresponds the variable `B`.

Using (8) we obtain in the case of Dirichlet boundary condition

$$\begin{array}{ccccccc} 0 \cdot B(1,:) & U & + & [u] & = & -u_D \\ 1 \cdot B(2,:) & U & + & [\partial_x u] & = & 0 \\ 1 \cdot B(3,:) & U & + & [\partial_y u] & = & 0 \\ 1 \cdot B(4,:) & U & + & [\partial_{xx} u] & = & 0 \\ 1 \cdot B(5,:) & U & + & [\partial_{yy} u] & = & 0 \\ \underbrace{1 \cdot B(6,:)}_{\text{These form matrix } \mathbf{D}} & U & + & [\partial_{xy} u] & = & \underbrace{0}_{\text{These form vector } \tilde{F}} \end{array}$$

Using (9) we get for Neumann boundary

$$\begin{array}{cccccc}
1 \cdot B(1,:) & U & + & [u] & = & 0 \\
(t_1^2 B(2,:) + t_1 t_2 B(3,:)) & U & + & [\partial_x u] & = & -n_1 u_N \\
(t_1 t_2 B(2,:) + t_2^2 B(3,:)) & U & + & [\partial_y u] & = & -n_2 u_N \\
1 \cdot B(4,:) & U & + & [\partial_{xx} u] & = & 0 \\
1 \cdot B(5,:) & U & + & [\partial_{yy} u] & = & 0 \\
\underbrace{1 \cdot B(6,:)}_{\text{These form matrix } \mathbf{D}} & U & + & [\partial_{xy} u] & = & \underbrace{0}_{\text{These form vector } \tilde{F}}
\end{array}$$

In code this corresponds to lines 95—115 in DIFFOP/d_matrix.m.

Thus, the system for jumps can be written as

$$\mathbf{D}U + J = \tilde{F}\,. \tag{10}$$

In code, the matrix $\mathbf{D}$ (stored in variable `D`) and the right hand side vector $\tilde{F}$ (stored in variable `Ft`) consist each of two parts, corresponding to Dirichlet and Neumann intersection points:

$$\mathbf{D} = \begin{pmatrix} \mathbf{D}_D \\ \mathbf{D}_D \end{pmatrix} \quad , \quad \tilde{F} = \begin{pmatrix} \tilde{F}_D \\ \tilde{F}_N \end{pmatrix} .$$

(ASSEMBLE/run_ejiim.m.)

Note: matrix $\mathbf{D}$ and vector $\tilde{F}$ depend on boundary conditions!

2.5 Solving the resulting system

Using the relations (7) and (10) the (augmented) EJIIM system is written

$$\begin{pmatrix} \mathbf{A} & \mathbf{\Psi} \\ \mathbf{D} & \mathbf{I} \end{pmatrix} \begin{pmatrix} U \\ J \end{pmatrix} = \begin{pmatrix} F \\ \tilde{F} \end{pmatrix}$$

where $\mathbf{I}$ is the identity operator.

In code (see ASSEMBLE/run_ejiim.m) the following notations are used:

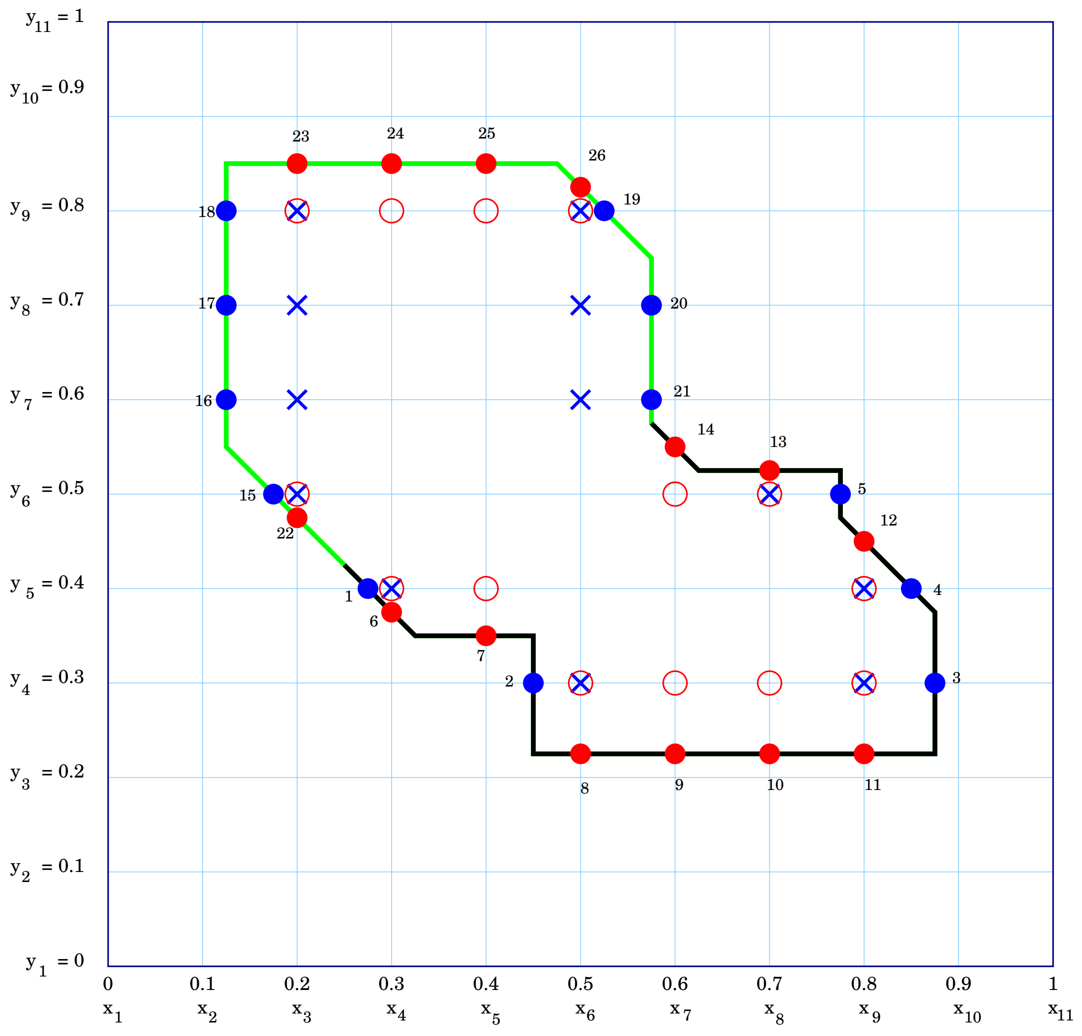
$$\texttt{EJIIM\_MATR} := \begin{pmatrix} \mathbf{A} & \mathbf{\Psi} \\ \mathbf{D} & \mathbf{I} \end{pmatrix} \quad , \quad \texttt{EJIIM\_RHS} := \begin{pmatrix} F \\ \tilde{F} \end{pmatrix} \quad , \quad \texttt{SOL} := \begin{pmatrix} U \\ J \end{pmatrix}$$

2.6 Visualisation of the results

Visualisation is done in ASSEMBLE/visualise_solution.m.

3 Example of interface structure

If you are going to replace the built-in geometry pre-processor by your own one, it has to provide the interface structure `IX`. Probably the best way to explain how it has to look like is a concrete example.



Light blue lines indicate the grid lines and the domain Ω^* is in this case a unit square with $n_x = n_y = 11$ and $h_x = h_y = 0.1$. On the black marked part of the boundary, Dirichlet boundary condition is given, the green one is the Neumann boundary.

Blue filled circles mark x-type intersection points, red filled circles are intersection points of y-type. Blue crosses mark the anchor points for x-intersections, red empty circles mark the anchor points for y-intersections.

Interface structure has following fields:

- `coord`: coordinates of the intersection points
- `type`: has values 'x' or 'y' corresponding to x and y-type intersections
- `anch`: indices of the anchor point. To a given intersection point (x_α, y_α) the anchor is defined as follows: It is a point $(x_i, y_j) \in \Omega^-$ such that
  - **for x-intersections:** $y_j = y_\alpha$ and $|x_i - x_\alpha| < h_h$
  - **for y-intersections:** $x_i = x_\alpha$ and $|y_j - y_\alpha| < h_y$
- `n`: components of the normalised normal vector at each of the intersection points.
- `t`: components of the tangential vector at each of the intersection points. The pair (n, t) has to form a right hand system. In two dimensions, $t = (-n_2, n_1)$.

Ordering of the intersection points
Currently, the code is constructed in such way that `IX` has to consist of two parts, `IX_D` (contains points along the Dirichlet boundary) and `IX_N` (contains the intersection points along the Neumann boundary). It is,

$$\texttt{IX} = \begin{pmatrix} \texttt{IX\_D} \\ \texttt{IX\_N} \end{pmatrix}$$

Each of structures `IX_D` and `IX_N` has to be ordered in such way that x-type intersections come first and then come y-intersections.

In our example this would look as follows:

Nr (IX)	Nr (IX_D)	Nr (IX_N)	coord	type	anch	n	t
1.	1.	–	0.275, 0.400	‘x’	4, 5	(-1, -1)/$\sqrt{2}$	(1, -1)/$\sqrt{2}$
2.	2.	–	0.450, 0.300	‘x’	6, 4	(-1, 0)	(0, -1)
3.	3.	–	0.875, 0.300	‘x’	9, 4	(1, 0)	(0, 1)
4.	4.	–	0.850, 0.400	‘x’	9, 5	(1, 1)/$\sqrt{2}$	(-1, 1)/$\sqrt{2}$
5.	5.	–	0.775, 0.500	‘x’	8, 6	(1, 0)	(0, 1)
6.	6.	–	0.300, 0.375	‘y’	4, 5	(-1, -1)/$\sqrt{2}$	(1, -1)/$\sqrt{2}$
7.	7.	–	0.400, 0.350	‘y’	5, 5	(0, -1)	(1, 0)
8.	8.	–	0.500, 0.225	‘y’	6, 4	(0, -1)	(1, 0)
9.	9.	–	0.600, 0.225	‘y’	7, 4	(0, -1)	(1, 0)
10.	10.	–	0.700, 0.225	‘y’	8, 4	(0, -1)	(1, 0)
11.	11.	–	0.800, 0.225	‘y’	9, 4	(0, -1)	(1, 0)
12.	12.	–	0.800, 0.450	‘y’	9, 5	(1, 1)/$\sqrt{2}$	(-1, 1)/$\sqrt{2}$
13.	13.	–	0.700, 0.525	‘y’	8, 6	(0, 1)	(-1, 0)
14.	14.	–	0.600, 0.550	‘y’	7, 6	(1, 1)/$\sqrt{2}$	(-1, 1)/$\sqrt{2}$
15.	–	1.	0.175, 0.500	‘x’	3, 6	(-1, -1)/$\sqrt{2}$	(1, -1)/$\sqrt{2}$
16.	–	2.	0.125, 0.600	‘x’	3, 7	(0, -1)	(1, 0)
17.	–	3.	0.125, 0.700	‘x’	3, 8	(0, -1)	(1, 0)
18.	–	4.	0.125, 0.800	‘x’	3, 9	(0, -1)	(1, 0)
19.	–	5.	0.525, 0.800	‘x’	6, 9	(1, 1)/$\sqrt{2}$	(-1, 1)/$\sqrt{2}$
20.	–	6.	0.575, 0.700	‘x’	6, 8	(1, 0)	(0, 1)
21.	–	7.	0.575, 0.600	‘x’	6, 7	(1, 0)	(0, 1)
22.	–	8.	0.200, 0.475	‘y’	3, 6	(-1, -1)/$\sqrt{2}$	(1, -1)/$\sqrt{2}$
23.	–	9.	0.200, 0.850	‘y’	3, 9	(0, 1)	(-1, 0)
24.	–	10.	0.300, 0.850	‘y’	4, 9	(0, 1)	(-1, 0)
25.	–	11.	0.400, 0.850	‘y’	5, 9	(0, 1)	(-1, 0)
26.	–	12.	0.500, 0.825	‘y’	6, 9	(1, 1)/$\sqrt{2}$	(-1, 1)/$\sqrt{2}$

References

[1] V. Rutka. *Immersed Interface Methods for Elliptic Boundary Value Problems.* PhD thesis, TU Kaiserslautern, 2005.

[2] V. Rutka, A. Wiegmann, and H. Andrä. EJIIM for Calculation of Effective Elastic Moduli in 3D Linear Elasticity. In preparation.

[3] J. A. Sethian and A. Wiegmann. Structural Boundary Design via Level Set and Explicit Jump Immersed Interface Methods. *J. Comput. Phys.*, 163(2):489—528, 2000.

[4] A. Wiegmann. *The Explicit–Jump Immersed Interface Method and Interface Problems for Differential Equations.* PhD thesis, University of Washington, 1998.

[5] A. Wiegmann and K. P. Bube. The Explicit-Jump Immersed Interface Method: Finite Difference Methods for PDEs with Piecewise Smooth Solutions. *SIAM J. Numer. Anal.*, 37(3):827—862, 2000.