# Dynamical Solver for Coulomb Explosions in a Magnetic Field

Tal Rubin

May 2020

## Abstract

A Coulomb explosion is a micro-plasma phenomenon in which an ionized cluster (or droplet) undergoes rapid expansion due to electrostatic repulsion between the ions. This is a single species plasma, with possible application for tabletop fusion neutron source. An interesting dynamic arises when such rapid expansion happens in a magnetic field. In this case, a rotation of the charge distribution occurs, and a "restoring force" appears, limiting the outer radius of the expansion in the direction perpendicular to the magnetic field.

In this project, I'll solve the dynamics of the Coulomb explosion of the single ion species. I'll use a fluid approximation, which is formulated as set of PDEs in time and 2 spatial dimensions.

## 1 Model Equations

The set of equations I am planning on solving is:

$$\nabla^2\varphi = -\frac{en}{\varepsilon}$$

$$\vec{E} = -\nabla\varphi$$

$$\frac{\partial n}{\partial t} + \nabla\cdot(\vec{v}n) = 0$$

$$mn\frac{\partial \vec{v}}{\partial t} = -mn\vec{v}\cdot\nabla\vec{v} - \nabla P + Zen\left(\vec{E} + \vec{v}\times\vec{B}\right)$$

With $P$ being a constant for now, $\vec{B} = B_0\hat{z} = Cosnt.$, with the intention of implementing the next Braginskii equation for the pressure later.

The boundary conditions are perfectly conducting walls, $\varphi = 0$, and no particle flux through them $\vec{v}\cdot\hat{n} = 0$ Initial conditions will be $\vec{v} = 0$ and $n = n_0\Theta\left(1 - \frac{x^2}{R_0^2} - \frac{y^2}{R_0^2}\right)$ with $\Theta$ being the Heaviside function.

# 2 FE Solver for Electric Potential

One of the required steps in solving an electrostatic plasma approximation is an electric field integrator. The purpose of this module is to produce an electric potential using boundary conditions and the space charge density in the domain. For this purpose, and because of the general usefulness of the tool, the finite element method was chosen, with quadratic elements involving eight nodes on the element boundary.

The first sub-module in this section is a mesher, used to define a grid of `N_col x N_row` elements, define the nodes and prepares the data for the solver.

## 2.1 Problem formulation

In a finite element solver, we use continuous Galerkin method to approximate the solution to a (partial) differential equation. The partial differential equation implemented in this code is the Poisson equation, written in the form

$$\begin{aligned} -\nabla\varepsilon(\vec{r})\nabla u = f(\vec{r}), &\qquad \vec{r}\epsilon\Omega \\ \sigma u + \beta\nabla u\cdot\hat{n} = q &\qquad \vec{r}\epsilon\partial\Omega \end{aligned}$$

with $\varepsilon$ generally being a second-order tensor.

For this project, $\varepsilon = 1$, a scalar, $f(\vec{r}) = \frac{Zen_i}{\varepsilon}$, with $Z = 1$ for hydrogen isotopes, $e$ being the elementary charge, and $\varepsilon$ beign the vacuum permittivity.

Here, we will solve Cartesian-coordinate domain, thus the $\nabla$ operators break down into $\nabla = \hat{x}\frac{\partial}{\partial x} + \hat{y}\frac{\partial}{\partial y}$

The Residual $R$ within the domain $\Omega$ is the difference between the approximate solution $\bar{u}$ to the exact solution, $u$.

$$R = -\left(\frac{\partial}{\partial x}\varepsilon_x\frac{\partial\bar{u}}{\partial x} + \frac{\partial}{\partial y}\varepsilon_y\frac{\partial\bar{u}}{\partial y}\right) - f$$

The approximate solution is assumed to be

$$\bar{u} = c_j N_j$$

with $c_j$ being scalar coefficients and $N_j$ orthonormal functions in the domain.

We will minimize the internal product of the residual with the basis functions $N_i$:

$$\langle R, N_i\rangle = \int_\Omega \left(-N_i\nabla\varepsilon(\vec{r})\nabla c_j N_j - fN_i\right)dxdy$$

$$\langle R, N_i\rangle = -\int_\Omega fN_i dxdy - \int_\Omega N_i\nabla\varepsilon(\vec{r})\nabla c_j N_j dxdy$$

Partially integrating the last integral:

$$\int_\Omega N_i\nabla\varepsilon(\vec{r})\nabla N_j dxdy = \int_{\partial\Omega}\left(\varepsilon(\vec{r})\nabla N_j\right)\cdot\hat{n}N_i ds - \int_\Omega \nabla N_i\cdot\left(\varepsilon(\vec{r})\nabla N_j\right)dxdy$$

The residual on the domain boundary $\partial\Omega$ is the difference between the prescribed boundary conditions to the boundary condition the approximate solution satisfies:

$$r = \sigma c_j N_j + \beta \nabla c_j N_j \cdot \hat{n} - q$$

Integrating the boundary residual and adding the domain residual integral:

$$\begin{aligned}\langle R, N_i \rangle_{\Omega+\partial\Omega} = c_j \int_\Omega \nabla N_i \cdot (\varepsilon(\vec{r}) \nabla N_j)\, dxdy - c_j \int_{\partial\Omega} (\varepsilon(\vec{r}) \nabla N_j) \cdot \hat{n} N_i ds \\ + c_j \int_{\partial\Omega} \sigma N_j N_i ds + c_j \int_{\partial\Omega} (\beta \nabla N_j \cdot \hat{n})\, N_i ds - \int_\Omega f N_i dxdy - \int_{\partial\Omega} q N_i ds\end{aligned}$$

This formulation is called the weak formulation for the Galerkin method. An inspection shows that all but the two last integrals would produce matrices multiplying the coefficient vector, and the two last integrals would produce inhomogeneous terms.

The equation left to solve is:

$$\left(k_{ij}^{\varepsilon} + k_{ij}^{\beta-\varepsilon}\right) c_j = F_i$$

With the stiffness matrices values determined by the basis functions $N_j$.

In our case, we will choose $\beta = \varepsilon$, so $k_{ij}^{\beta-\varepsilon} = 0$.

Let us calculate the values $k_{ij}^{\varepsilon}$, assuming $\varepsilon = Const.$ and for the domain $x, y \epsilon [-1, 1] \otimes [-1, 1]$.

The traditional basis functions are Lagrange polynomials, each with the value 1 at its node, and 0 at all other nodes. This way, the value of the potential at any node is equal to the coefficient of the node.

The basis function for the domain are:

| | |
|---|---|
| $N_1$ | $1/4\xi(\xi-1)\eta(\eta-1)$ |
| $N_2$ | $1/4\xi(1+\xi)\eta(\eta-1)$ |
| $N_3$ | $1/4\xi(1+\xi)\eta(\eta+1)$ |
| $N_4$ | $1/4\xi(\xi-1)\eta(\eta+1)$ |
| $N_5$ | $1/2\eta(\eta-1)(1-\xi^2)$ |
| $N_6$ | $1/2\xi(1+\xi)(1-\eta^2)$ |
| $N_7$ | $1/2\eta(1+\eta)(1-\xi^2)$ |
| $N_8$ | $1/2\xi(\xi-1)(1-\eta^2)$ |
| $N_9$ | $(1-\xi^2)(1-\eta^2)$ |

And their gradients are:

| $\nabla N$ | $\hat{\xi}$ | $\hat{\eta}$ |
|---|---|---|
| $\nabla N_1$ | $1/4(2\xi-1)\eta(\eta-1)$ | $1/4\xi(\xi-1)(2\eta-1)$ |
| $\nabla N_2$ | $1/4(1+2\xi)\eta(\eta-1)$ | $1/4\xi(1+\xi)(2\eta-1)$ |
| $\nabla N_3$ | $1/4(1+2\xi)\eta(\eta+1)$ | $1/4\xi(1+\xi)(2\eta+1)$ |
| $\nabla N_4$ | $1/4(2\xi-1)\eta(\eta+1)$ | $1/4\xi(\xi-1)(2\eta+1)$ |
| $\nabla N_5$ | $-\xi\eta(\eta-1)$ | $1/2(2\eta-1)(1-\xi^2)$ |
| $\nabla N_6$ | $1/2(1+2\xi)(1-\eta^2)$ | $-\eta\xi(1+\xi)$ |
| $\nabla N_7$ | $-\xi\eta(1+\eta)$ | $1/2(1+2\eta)(1-\xi^2)$ |
| $\nabla N_8$ | $(\xi-1/2)(1-\eta^2)$ | $\xi\eta(1-\xi)$ |
| $\nabla N_9$ | $-2\xi(1-\eta^2)$ | $-2\eta(1-\xi^2)$ |

### 2.1.1 Calculating $k_{ij}^{\varepsilon}$

The stiffness matrix elements $k_{0ij}^{\varepsilon} = \int_{\Omega} \nabla N_i \cdot \nabla N_j dxdy$ for a rectangular `2x2` grid would be:

$$k^{\varepsilon} = \begin{bmatrix} \frac{28}{45} & -\frac{1}{30} & -\frac{1}{45} & -\frac{1}{30} & -\frac{1}{5} & \frac{1}{9} & \frac{1}{9} & -\frac{1}{5} & -\frac{16}{45} \\ -\frac{1}{30} & \frac{28}{45} & -\frac{1}{30} & -\frac{1}{45} & -\frac{1}{5} & -\frac{1}{5} & \frac{1}{9} & \frac{1}{9} & -\frac{16}{45} \\ -\frac{1}{45} & -\frac{1}{30} & \frac{28}{45} & -\frac{1}{30} & \frac{1}{9} & -\frac{1}{5} & -\frac{1}{5} & \frac{1}{9} & -\frac{16}{45} \\ -\frac{1}{30} & -\frac{1}{45} & -\frac{1}{30} & \frac{28}{45} & \frac{1}{9} & \frac{1}{9} & -\frac{1}{5} & -\frac{1}{5} & -\frac{16}{45} \\ -\frac{1}{5} & -\frac{1}{5} & \frac{1}{9} & \frac{1}{9} & \frac{88}{45} & -\frac{16}{45} & 0 & -\frac{16}{45} & -\frac{16}{15} \\ \frac{1}{9} & -\frac{1}{5} & -\frac{1}{5} & \frac{1}{9} & -\frac{16}{45} & \frac{88}{45} & -\frac{16}{45} & 0 & -\frac{16}{15} \\ \frac{1}{9} & \frac{1}{9} & -\frac{1}{5} & -\frac{1}{5} & 0 & -\frac{16}{45} & \frac{88}{45} & -\frac{16}{45} & -\frac{16}{15} \\ -\frac{1}{5} & \frac{1}{9} & \frac{1}{9} & -\frac{1}{5} & -\frac{16}{45} & 0 & -\frac{16}{45} & \frac{88}{45} & -\frac{16}{15} \\ -\frac{16}{45} & -\frac{16}{45} & -\frac{16}{45} & -\frac{16}{45} & -\frac{16}{15} & -\frac{16}{15} & -\frac{16}{15} & -\frac{16}{15} & \frac{256}{45} \end{bmatrix}$$

But this is not the case for a deformed grid. For a general grid, a change of variables should be made, with $x, y = XY \cdot \vec{N}(\xi, \eta)$ where $\vec{N}$ is a column vector of the basis functions, and $XY$ is a `(2,9)` matrix composed of column vectors $\begin{bmatrix} \texttt{x} \\ \texttt{y} \end{bmatrix}$ of the vertices of the mapping.

Because the permittivity is taken to be a constant, the exact $k_{ij}^{\varepsilon}$ matrix elements are:

$$k_{ij}^{\varepsilon} = \int_{-1}^{1}\int_{-1}^{1} \left\{ \frac{\partial N_i}{\partial \xi}\frac{\partial N_j}{\partial \xi}\left[\left(\frac{\partial \xi}{\partial x}\right)^2 + \left(\frac{\partial \xi}{\partial y}\right)^2\right] + \frac{\partial N_i}{\partial \eta}\frac{\partial N_j}{\partial \eta}\left[\left(\frac{\partial \eta}{\partial x}\right)^2 + \left(\frac{\partial \eta}{\partial y}\right)^2\right]\right.$$
$$\left. + \left(\frac{\partial N_i}{\partial \xi}\frac{\partial N_j}{\partial \eta} + \frac{\partial N_i}{\partial \eta}\frac{\partial N_j}{\partial \xi}\right)\left(\frac{\partial \xi}{\partial x}\frac{\partial \eta}{\partial x} + \frac{\partial \xi}{\partial y}\frac{\partial \eta}{\partial y}\right)\right\} \frac{\partial N_n}{\partial \xi}\frac{\partial N_m}{\partial \eta}\left(X_n Y_m - Y_n X_m\right) d\xi d\eta$$

According to the inverse function theorem $\frac{\partial\xi}{\partial x} = \frac{1}{|J|}\frac{\partial y}{\partial\eta} = \frac{1}{|J|}Y_k\frac{\partial N_k}{\partial\eta}$, $\frac{\partial\xi}{\partial y} = -\frac{1}{|J|}\frac{\partial x}{\partial\eta} = -\frac{1}{|J|}X_k\frac{\partial N_k}{\partial\eta}$, $\frac{\partial\eta}{\partial x} = -\frac{1}{|J|}\frac{\partial y}{\partial\xi} = -\frac{1}{|J|}Y_k\frac{\partial N_k}{\partial\xi}$, and $\frac{\partial\eta}{\partial y} = \frac{1}{|J|}\frac{\partial x}{\partial\xi} = \frac{1}{|J|}X_k\frac{\partial N_k}{\partial\xi}$, so the matrix elements in terms of known functions are:

$$k^{\varepsilon}_{ij} = \int\limits_{-1}^{1}\int\limits_{-1}^{1}\Bigg\{\frac{\partial N_i}{\partial\xi}\frac{\partial N_j}{\partial\xi}\frac{\partial N_n}{\partial\eta}\frac{\partial N_m}{\partial\eta} + \frac{\partial N_i}{\partial\eta}\frac{\partial N_j}{\partial\eta}\frac{\partial N_n}{\partial\xi}\frac{\partial N_m}{\partial\xi} - \left(\frac{\partial N_i}{\partial\xi}\frac{\partial N_j}{\partial\eta} + \frac{\partial N_i}{\partial\eta}\frac{\partial N_j}{\partial\xi}\right)\frac{\partial N_n}{\partial\xi}\frac{\partial N_m}{\partial\eta}\Bigg\} \frac{Y_nY_m + X_nX_m}{\frac{\partial N_a}{\partial\xi}\frac{\partial N_b}{\partial\eta}\left(X_aY_b - Y_aX_b\right)} d\xi d\eta$$

Because there is no guarantee the division of the polynomial in the curly brackets by the polynomial in the denominator leave no remainder, evaluating this integral should be done numerically.

The polynomial in the curly brackets is of degree 6 in both $\xi$ and $\eta$, and the polynomial in the denominator is of degree 3 in both $\xi$ and $\eta$. In the best case scenario, the resulting integrand would be a 3rd degree polynomial, but it could also have an additive term that is an inverse of a 3rd degree polynomial.

Gauss–Legendre quadrature using `n` points can exactly find the integral of a polynomial of order $2n-1$ in one dimension. For 2 dimensional polynomials with remainder, `n=6` is chosen, for increased accuracy in evaluating the remainder, with the option of using `n=5` instead.

$$\int\limits_{-1}^{1}\int\limits_{-1}^{1} f(\xi,\eta)d\xi d\eta \approx \sum_i\sum_j w_iw_jf(\xi_i,\eta_j)$$

with $w_i$ and $x_i$ from Table 25.4 in Abramowitz and Stegun (1964)

6th order quadrature

| $x_i$ | $w_i$ |
|---|---|
| $\pm 0.238619186083197$ | 0.467913934572691 |
| $\pm 0.661209386466265$ | 0.360761573048139 |
| $\pm 0.932469514203152$ | 0.171324492379170 |

5th order quadrature

| $x_i$ | $w_i$ |
|---|---|
| 0 | 128/225 |
| $\pm\frac{1}{3}\sqrt{5-2\sqrt{\frac{10}{7}}}$ | $\frac{322+13\sqrt{70}}{900}$ |
| $\pm\frac{1}{3}\sqrt{5+2\sqrt{\frac{10}{7}}}$ | $\frac{322-13\sqrt{70}}{900}$ |

### 2.1.2 Calculating $F_i$

Similarly to $k^{\varepsilon}_{ij}$, the force vector elements $F_i$ are the inner product of the basis functions $N_i$ and the charge density divided by the permittivity.

$$F_i = \frac{e}{\varepsilon_0} \int_\Omega n N_i dx dy$$

The charge density will be represented by quadratic basis functions as well

$$n = d_j M_j$$

turning the integral into

$$F_i = \frac{e}{\varepsilon_0} \int_\Omega d_j N_i M_j \left| \frac{\partial(x,y)}{\partial(\xi,\eta)} \right| d\xi d\eta$$

$$F_i = \frac{e}{\varepsilon_0} \int_{-1}^{1} \int_{-1}^{1} N_i M_j \frac{\partial N_n}{\partial \xi} \frac{\partial N_m}{\partial \eta} d\xi d\eta \left(X_n Y_m - Y_n X_m\right) d_j = F_{ij} d_j$$

We are left with solving the linear system of equations

$$k_{ij} c_j = F_{ij} d_j$$

with $d_j$ defining the (predetermined) charge distribution.

For now, the FE solver is decoupled from the DG solver, so the charge density is being represented in the FE scheme using the same basis functions $M_k = N_k$, with a separate projection operator to broadcast the Legendre Polynomials into the Lagrange-interpolation polynomials.

## 2.2 Convergence test

We have calculated the $L_2$ norm of the error between the FE solution and a (truncated) analytic solution to the Poisson equation.

$$\begin{aligned} L_2 &= \sqrt{\int_\Omega (u - \bar{u})^2 dx dy} \\ &= \sqrt{\sum_k \int_{-1}^{1} \int_{-1}^{1} \Big( u\big(x_k(\xi,\eta), y_k(\xi,\eta)\big) - \bar{u}_k(\xi,\eta) \Big)^2 \left| \frac{\partial(x,y)}{\partial(\xi,\eta)} \right| d\xi d\eta} \end{aligned}$$

with the sum over k being the sum over elements. These integrals are evaluated as a 6th order Gauss quadrature.

The reference problem is Poisson equation on a `(-1,1)x(-1,1)` square. With boundary condition of zero potential on the four boundaries, and uniform unit forcing term.

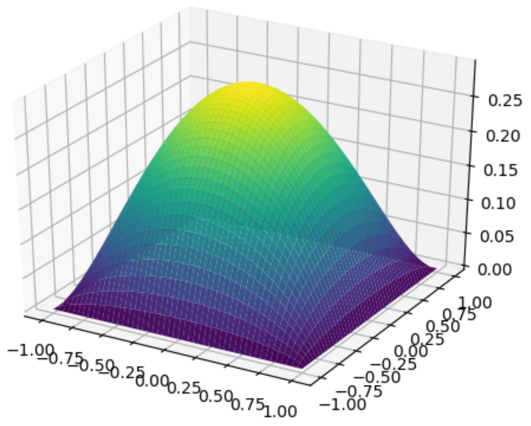

Figure 1: Analytic solution, sum over odd k -s up to 99

$$\Delta u = 1$$

The solution for $u$ is:

$$u(x,y) = \frac{1-x^2}{2} - \frac{16}{\pi^3} \sum_{odd \quad k} \frac{\sin\left(k\pi \frac{1+x}{2}\right)}{k^3 \sinh(k\pi)} \left(\sinh\left(k\pi \frac{1+y}{2}\right) + \sinh\left(k\pi \frac{1-y}{2}\right)\right)$$

And is plotted in Figure 1.

The error norm, calculated for various number of divisions of the domain is plotted in Figure 2. The error is falling the fastest when the number of row elements is equal to the number of column elements (square elements), due to the symmetry of the solution.

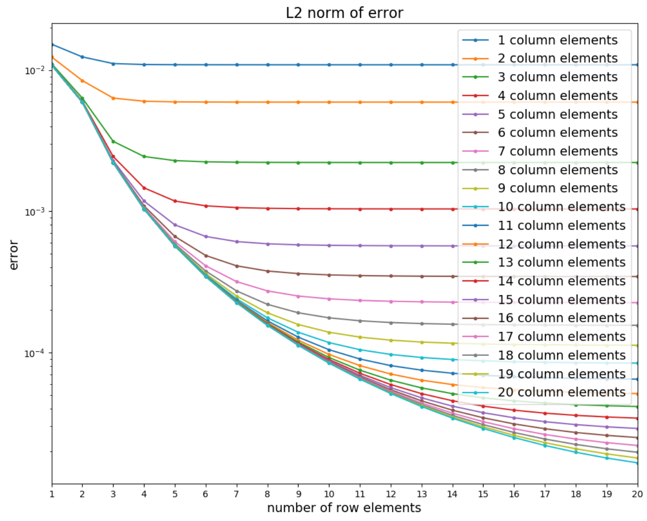

Figure 2: L2 norm of the error

.

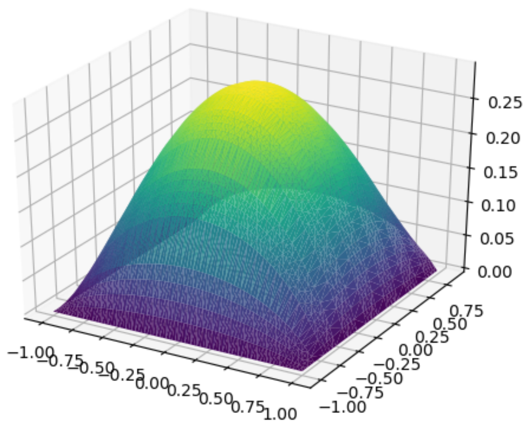

Figure 3: Solution with division of the domain into 7x20 elements.

## 3 DG Solver for density

The equation we are intending to solve is

$$\frac{\partial n}{\partial t} + \nabla(\vec{v}n) = 0$$

where, as in the previous section the density is a weighted sum of basis functions $n = d(t)_j M_j$, but in the case of discontinuous Galerkin, each function is defined only in a single element.

We remember that the potential solution $c_j$ corresponds to the value of the solution at each node, and nodes 1-8 are shared between elements.

In the case of discontinuous Galerkin, each element has 9 nodes, each with its own unique charge distribution value.

$$R = \frac{\mathrm{d}d(t)_j}{\mathrm{d}t} M_j + d(t)_j \nabla \cdot (\vec{v} M_j)$$

$$\langle R, M_i \rangle = \int_\Omega \left( \frac{\mathrm{d}d(t)_j}{\mathrm{d}t} M_i M_j + d(t)_j M_i \nabla \cdot (\vec{v} M_j) \right) dxdy$$

$$\langle R, M_i \rangle = \frac{\mathrm{d}d(t)_j}{\mathrm{d}t} Mass_{ij} - d(t)_j \int_\Omega \vec{v} M_j \cdot \nabla N_i dxdy + d(t)_j \int_{\partial\Omega} M_j M_i \vec{v} \cdot \hat{n} ds$$

Using a 3rd order RK time derivative, the continuity equation becomes:

$$d_j^{(1)} = \left[ 1 + \Delta t Mass_{ij}^{-1} \left( \int_\Omega M_j \vec{v} \cdot \nabla M_i dxdy - \int_{\partial\Omega} M_j M_i \vec{v} \cdot \hat{n} ds \right) \right] d_j^t$$

$$d_j^{(2)} = \frac{3}{4} d_j^t + \frac{1}{4} \left[ 1 + \Delta t Mass_{ij}^{-1} \left( \int_\Omega M_j \vec{v} \cdot \nabla M_i dxdy - \int_{\partial\Omega} M_j M_i \vec{v} \cdot \hat{n} ds \right) \right] d_j^{(1)}$$

$$d_j^{t+1} = \frac{1}{3} d_j^t + \frac{2}{3} d_j^{(2)} \left[ 1 + \Delta t Mass_{ij}^{-1} \left( \int_\Omega M_j \vec{v} \cdot \nabla M_i dxdy - \int_{\partial\Omega} M_j M_i \vec{v} \cdot \hat{n} ds \right) \right]$$

We will use the same basis function form for $\vec{v}$:

$$\vec{v} = (v_{xk}\hat{x} + v_{yk}\hat{y}) M_k$$

### 3.1 Calculating $\Gamma_{ij}^\Omega$

The matrix $\Gamma_{ij}^\Omega = \int_\Omega M_j \vec{v} \cdot \nabla M_i dxdy$ is straightforward to calculate:

$$\Gamma_{ij}^\Omega = \int_\Omega M_j M_m (v_{xm}\hat{x} + v_{ym}\hat{y}) \cdot$$
$$\left[ \left( \frac{\partial M_i}{\partial \xi} \frac{\partial \xi}{\partial x} + \frac{\partial M_i}{\partial \eta} \frac{\partial \eta}{\partial x} \right) \hat{x} + \left( \frac{\partial M_i}{\partial \xi} \frac{\partial \xi}{\partial y} + \frac{\partial M_i}{\partial \eta} \frac{\partial \eta}{\partial y} \right) \hat{y} \right] \left| \frac{\partial(x,y)}{\partial(\xi,\eta)} \right| d\xi d\eta$$
$$= \int_{-1}^{1} \int_{-1}^{1} M_j M_m \left( \frac{\partial M_i}{\partial \xi} \frac{\partial N_n}{\partial \eta} - \frac{\partial M_i}{\partial \eta} \frac{\partial N_n}{\partial \xi} \right) d\xi d\eta (Y_n v_{xm} - X_n v_{ym})$$

And it is written in the form such that $\bar{\Gamma}^{\Omega}_{ijnk}$ is calculated independently and is multiplied by the node position coordinates and the velocity magnitudes.

## 3.2 Calculating $\Gamma^{\partial\Omega}_{ij}$

The term $\Gamma^{\partial\Omega}_{ij} = \int_{\partial\Omega} M_j M_i \vec{v}\cdot\hat{n} ds$ however, describes interaction between adjacent elements. The normal vectors $\hat{n}$ at each element edge are:

$$\hat{n}_0 = \hat{n}\big|_{\eta=-1} = \frac{\frac{\partial y}{\partial\xi}\hat{x} - \frac{\partial x}{\partial\xi}\hat{y}}{\sqrt{\left(\frac{\partial x}{\partial\xi}\right)^2 + \left(\frac{\partial y}{\partial\xi}\right)^2}}$$

$$\hat{n}_1 = \hat{n}\big|_{\xi=1} = \frac{\frac{\partial y}{\partial\eta}\hat{x} - \frac{\partial x}{\partial\eta}\hat{y}}{\sqrt{\left(\frac{\partial x}{\partial\eta}\right)^2 + \left(\frac{\partial y}{\partial\eta}\right)^2}}$$

$$\hat{n}_2 = \hat{n}\big|_{\eta=1} = \frac{-\frac{\partial y}{\partial\xi}\hat{x} + \frac{\partial x}{\partial\xi}\hat{y}}{\sqrt{\left(\frac{\partial x}{\partial\xi}\right)^2 + \left(\frac{\partial y}{\partial\xi}\right)^2}}$$

$$\hat{n}_3 = \hat{n}\big|_{\xi=-1} = \frac{-\frac{\partial y}{\partial\eta}\hat{x} + \frac{\partial x}{\partial\eta}\hat{y}}{\sqrt{\left(\frac{\partial x}{\partial\eta}\right)^2 + \left(\frac{\partial y}{\partial\eta}\right)^2}}$$

$$\Gamma^{\partial\Omega}_{ij} = \int_{\partial\Omega} M_i M_j \vec{v}\cdot\hat{n} ds = \Gamma^{\partial 1}_{ij} + \Gamma^{\partial 2}_{ij} + \Gamma^{\partial 3}_{ij} + \Gamma^{\partial 4}_{ij}$$

$$\left[\int_{-1}^{1} M_i M_j M_m \frac{\partial N_n}{\partial\xi}\bigg|_{\eta=-1} d\xi + \int_{-1}^{1} M_i M_j M_m \frac{\partial N_n}{\partial\eta}\bigg|_{\xi=1} d\eta\right.$$
$$\left. - \int_{-1}^{1} M_i M_j M_m \frac{\partial N_n}{\partial\xi}\bigg|_{\eta=1} d\xi - \int_{-1}^{1} M_i M_j M_m \frac{\partial N_n}{\partial\eta}\bigg|_{\xi=-1} d\eta\right] (Y_n v_{xm} - X_n v_{ym})$$

## 3.3 Lax–Friedrichs Flux

In order to the fluxes described above are"centered" fluxes. In order to get a more accurate "upwind" fluxes, we will introduce a numerical dissipation.

The dissipation is

$$\Gamma^{\partial\Omega}_{D} = \frac{1}{2}\left(\alpha^1(n(a) - n(b)\right)$$
$$\alpha = |\overline{\vec{v}\cdot\hat{n}}|$$

$$\alpha^1 = \left| \int_{-1}^{1} M_m \left.\frac{\partial N_n}{\partial \xi}\right|_{\eta=-1} d\xi \, (Y_n v_{xm} - X_n v_{ym}) \right|$$

$$\alpha^2 = \left| \int_{-1}^{1} M_m \left.\frac{\partial N_n}{\partial \eta}\right|_{\xi=1} d\eta \, (Y_n v_{xm} - X_n v_{ym}) \right|$$

$$\alpha^3 = \left| \int_{-1}^{1} M_m \left.\frac{\partial N_n}{\partial \xi}\right|_{\eta=1} d\xi \, (Y_n v_{xm} - X_n v_{ym}) \right|$$

$$\alpha^4 = \left| \int_{-1}^{1} M_m \left.\frac{\partial N_n}{\partial \eta}\right|_{\xi=-1} d\eta \, (Y_n v_{xm} - X_n v_{ym}) \right|$$

****

# 4 Code Structure

## 4.1 Pre-Processor Module

The pre-processor module handles meshing and matrix assembly for the finite-element solver. It also partly assembles the geometry-dependent matrices for the discontinuous Galerkin solver.

Many methods in this module have a parameter `DGorder`. This parameter determines the number of the DG basis functions used in the DG scheme.

### 4.1.1 CheckDetJ(XY,fineness=10)

Before meshing begins, the pre-processor checks the geometry provided for singular values of the determinant. This method receives the geometry definition intended for the `NodeArrange` method, and checks a `fineness x fineness` grid of points within for non-positive $|J|$.

### 4.1.2 NodeArrange(N_row,N_col,XY,NodeBeginNum,ElemBeginNum)

This method is a mesher - it arranged nodes within a "macro-element". A "macro-element" is defined by the `(2,9)` matrix `XY`, with parametrization $x, y =$ `XY`$\cdot\vec{N}(\xi, \eta)$. The "macro-element" is divided into `N_row` rows and `N_col` columns in $\xi$ and $\eta$. As a consequence of this, positioning nodes 5, 7, and 9 to the left of the "macro-element" center would result in a refined mesh in the region to the left of these nodes, and a coarser mesh in the region to the right.

The method returns a list of nodes. Each node is represented by a column, with structure:

$$\begin{bmatrix} \texttt{\# node} \\ \texttt{node x coordinate} \\ \texttt{node y coordinate} \end{bmatrix}$$

and a list of elements, with each element represented by a row, with structure:

```
[# element, # node_1, ..., # node_9]
```

The node numbering is presented in Figure 4.

Additionally, the method returns a connectivity matrix, identifying the element numbers adjacent to each face:

```
[# ele face_1,# ele face_2, # ele face_3, # ele face_4]
```

where `face_1` is the face containing node 5, `face_2` is the face containing node 6, `face_3` is the face containing node 7, and `face_4` is the face containing node 8.

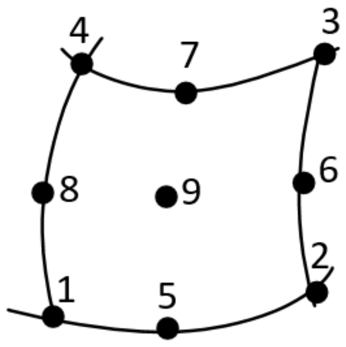

Figure 4: Node numbering within each element

### 4.1.3 assembleGlobalMats(GlobalElementMatrix,NodeList,Qorder)

This method assembles the matrices $k_{ij}$ and $F_{ij}$ as described above. It first calculate the matrix for each element, and then adds its values to the global matrices. By default, it calculates 6th order Gaussian quadrature for the elements of $k_{ij}$, but if Qorder=5, it would calculate a 5th order quadrature instead.

It returns `GlobalStiffMat, GlobalForceMat`.

### 4.1.4 assembleElementMats(GlobalElementMatrix,NodeList,DGorder)

This method assembles geometry dependent part the DG matrices. Since in DG the flux operator depends on both the geometry and the velocity, the final assembly of the flux matrix is performed in the Solver.

In DG each element is treated independently, and the boundary fluxes are connecting the elements, the method returns arrays of matrices.

The mass matrix is block diagonal, so in order to save calculation time in the time-stepping loop of the solver, each block is inverted independently.

This method returns:

`EleFluxMatx, EleFluxMaty` arrays of flux matrices within each element. The total flux within each element would be `EleFluxMatx`$v_x$ + `EleFluxMaty`$v_y$ with $v_x$ and $v_y$ being vectors of coefficients of the DG expansion of the velocity.

`InvMassMat` the inverse of the mass matrix.

`b_mast_1x, b_mast_1y, ..., b_mast_4x, b_mast_4y` arrays of flux matrices out of the boundaries of each element. The total flux out of boundary 1 would be `b_mast_1x`$v_x$ + `b_mast_1y`$v_y$ with $v_x$ and $v_y$ being vectors of coefficients of the DG expansion of the velocity.

`LF_mast_1x, LF_mast_1y, ..., LF_mast_4x, LF_mast_4y` arrays of the Lax-Friedrichs flux matrices out of the boundaries of each element. The total Lax-Friedrichs flux out of boundary 1 would be `LF_mast_1x`$v_x$ + `LF_mast_1y`$v_y$ with $v_x$ and $v_y$ being vectors of coefficients of the DG expansion of the velocity.

#### 4.1.5 initMasterFluxMat(DGorder), initMasterForcingMatrix(), initMasterMassMatrix(DGorder), initMasterBoundMatrix(DGorder), initMasterLaxFriedrichs(DGorder)

These methods calculate the master flux forcing or mass matrices. The master matrices are 2 or 4 dimensional matrices, that after multiplication by the geometry parameters and / or the velocity parameters become the operators required for the FE or DG solvers.

### 4.2 polynomial2d Module, poly Class

The poly class handles 2D polynomial operations. Polynomials are represented as 2D arrays, with `coefficients[i,j]` representing the coefficient of $y^i x^j$, for the polynomial $p(x, y) = \sum_{i,j}$ `coefficients[i,j]`$y^i x^j$. Methods in this class:

#### 4.2.1 __init__(self, coefficients)

Initiates a poly instance, with list, list of lists, or numpy array as the coefficients.

#### 4.2.2 __repr__(self), tex(self), __str__(self)

Returns the canonical string representation of a polynomial in python or tex formats.

#### 4.2.3 __call__(self, x=None,y=None)

Returns the evaluated polynomial as either a 1D polynomial in x or y, if given only a y or x value, or returns the evaluated polynomial as a scalar if given both x and y.

#### 4.2.4 degreex(self), degreey(self)

Returns the degree +1 of the x or y dimension of the polynomial.

#### 4.2.5 degree(self)

Returns the degree +1 of the polynomial.

#### 4.2.6 __add__(self,other), __radd__(self,other), __sub__(self,other)

Performs polynomial addition between two polynomials or between a polynomial and a scalar. Scalars are casted into a 0th degree polynomial, and then added. Subtraction is performed as multiplication by -1 followed by addition.

#### 4.2.7 __mul__(self,other), __rmul__(self,other)

Performs multiplication of polynomials or polynomial and a scalar. Returns a polynomial of degree in x and y being the sums of the degrees in x and y of the constituent polynomials.

#### 4.2.8 dx(self), dy(self)

Performs a derivative operation on a polynomial in x or y. Returns a polynomial with a reduced degree in x or y.

#### 4.2.9 intx(self,ax=None,bx=None), inty(self,ay=None,by=None)

Performs integration in x or y, and returns the integrated polynomial with the constant of integration = 0 - if not given ax, or ay , or the integrated polynomial evaluated at ax or ay - if not given bx or by, or the integral between ax and bx or ay and by.

#### 4.2.10 scalar(self)

Returns the scalar value of a 0th degree polynomial.