Finite Element Method: Poisson Equation

Aleksandar Ivanov

April 14, 2021

1 Problem Statements

Implement and describe the properties of Finite Element Methods (FEM) in two dimensions using the example of the Poisson equation for the velocity of liquid flow through a pipe.

Problem 1

Calculate the volumetric flow rate for a semicircular pipe.

Problem 2

For a comparison with the SOR method calculate the volumetric flow through the birdhouse-shaped pipe cross-section from the 5th task.

2 Mathematical Setup

As discussed in the fifth task, we will be solving the dimensionless Poisson equation

$$\nabla^2 u = 1. \tag{1}$$

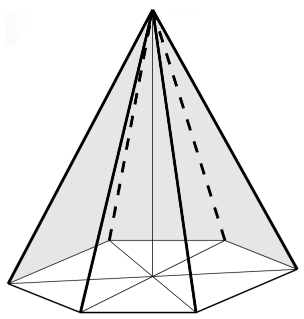

Figure 1: The form of the trial functions ϕ_n centered on the vertex (x_n, y_n).

Setting up the stage on which FEM will play its role consists of choosing an arbitrary (i.e. not limited to the x and y axes) mesh of N_N points called nodes, then connecting those points with lines so as to fill the domain of interest with N_T triangles. This gives us a triangulation of the domain. The idea of the method is then to expand every function that appears in our equation as a sum over trial functions ϕ_n

$$u = \sum_{i=1}^{N_N} c_i \phi_i, \tag{2}$$

where usually the trial functions are chosen to be the simplest possible functions — the piecewise linear functions. What these functions look like is best illustrated by plotting them and exactly this is shown in fig. 1.

Taking a geometrical stance, the Poisson equation from before can be rewritten as

$$\langle w, \nabla^2 u - 1 \rangle = 0 \qquad \forall w \in L^2(D), \tag{3}$$

where for the inner product we will choose

$$\langle f, g \rangle = \int_D f g \, \mathrm{d}S, \tag{4}$$

where f and g are real functions.

If we also expand the functions w, as a sum over the trial functions[1]

$$w = \sum_{i=1}^{N_N} d_i \phi_i, \tag{5}$$

then we can do the integration by hand, since our functions are so simple, and we have effectively

[1] In general, we can choose another set of functions called test functions to expand w into, but here we will not take that approach.

turned our initial PDE into a system of linear equations for the coefficients c_i

$$S\mathbf{c} = \mathbf{g}, \tag{6}$$

where $S \in \mathbb{R}^{N_N \times N_N}$ and $\mathbf{g} \in \mathbb{R}^{N_N}$ is constructed from the right-hand side of the PDE.

The complicated part of this method is then to construct the matrix S and the vector $\mathbf{g}$. We will use the following organization of the problem. Each node will be given a global index $i \in [1, N_N]$, which fixes a permutation of the rows/columns of the matrix S and vector $\mathbf{g}$. Furthermore, we will index the triangles with an index $t \in [1, N_T]$. Within each triangle the vertices will also have a local index $m \in [1, 2, 3]$, so that each triangle is defined by a relation between the local and global indices of the vertices that make it up.

To build the matrices we will iterate through the triangles and slowly add in all the interactions. For two points with local indices m and n in a given triangle t, the integration from the inner product gives the contribution

$$\begin{aligned} S^t_{mn} = \frac{1}{4A_t} \big[&(y_{m+1} - y_{m+2})(y_{n+1} - y_{n+2}) \\ &+ (x_{m+2} - x_{m+1})(x_{n+2} - x_{n+1}) \big], \end{aligned} \tag{7}$$

where A_t is the area of the triangle t and the index addition is taken cyclically i.e. modulo 3. The full matrix S is then constructed as

$$S_{ij} = \sum_{t,m,n} S^t_{mn}, \tag{8}$$

where the sum goes over those combinations of nodes and triangles that can give rise to the global index on the left, i.e. such t and m that give i and such t and n that give j.

Similarly, the vector $\mathbf{g}$ is constructed from the partial components gotten by integration

$$\begin{aligned} g^t_m = \frac{1}{6} \big[&(y_{m+2} - y_m)(x_{m+1} - x_m) \\ &- (x_{m+2} - x_m)(y_{m+1} - y_m) \big], \end{aligned} \tag{9}$$

by combining them as follows

$$g_i = \sum_{t,m} g^t_m, \tag{10}$$

where, again, the sum is of the same type as above.

The boundary condition in our case are of Dirichlet type, which are easy to implement for FEM. All we need to do to take this kind of condition into account is to just ignore the contributions to the above sums that come from points on the boundary. [2] It should be made clear that this is not the same as completely ignoring the points on the boundary since they still play a role in generating the triangulation.

Finally, to calculate the volumetric flow rate and the Poiseuille coefficient all we need to do is integrate the velocity field u and in terms of the coefficients c_i this becomes

$$\Phi = \int_D u \, \mathrm{d}S = \sum_{t=1}^{N_T} \frac{A_t}{3} \sum_{m=1}^{3} u_{i(t,m)} \tag{11}$$

$$C = 8\pi \frac{\Phi}{A^2}, \tag{12}$$

where $i(t, m)$ is the global index i assigned to the local index m of triangles t and A is the total area of the domain.

3 Numerical Setup

An important part for the success of FEM is the choice of mesh used. Because of the widespread use of FEM, there exist many sophisticated generators of meshes that are optimized for good results. As an introduction, here we will not use meshes that are very optimized.

We will however, always add a set of points that parametrize the boundary, since one of the main problems that FEM easily fixes is an easy implementation of the boundary conditions, which is a challenge for methods like the Finite Differences Method.

A basic type of mesh that we can try is a uniform xy mesh with the extra added boundary. This can then tell us about the importance of the boundary since the only difference between this mesh and the Finite Differences Method is the better implementation of the boundary.

To automatically generate meshes that are a bit more advanced, i.e. that have the minimal number of very acute triangles while still having a larger density of vertices near the boundary of the region compared to the center, we will use the following procedure.

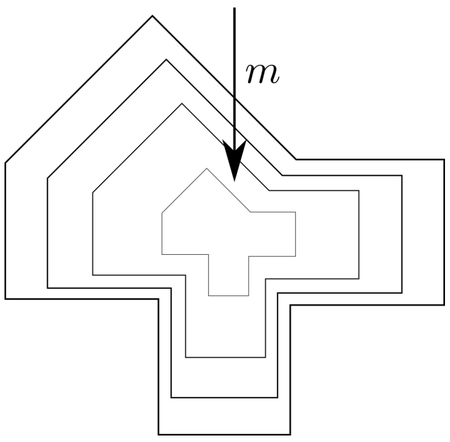

Figure 2: The scaling down described in the procedure and the direction of growing m.

1. We parametrize the boundary of our region. For the example of a circle this would be

$$(x = \cos(s), y = \sin(s)) \in \partial D, \quad s \in [0, 2\pi). \tag{13}$$

2. Uniformly sampling the parameter s we place n points along the boundary, thus describing it.

3. We describe another curve that has the shape of the boundary, but this time scaled by a factor $f \in (0, 1]$, which is taken from a density distribution ρ, to bias it close to the boundary, making the density of points there larger. The number of points on the curve is suitably reduced by a factor of f. This new curve's initial point for the parametrization is shifted by half a step so that we make the triangles as isosceles as possible, giving us the best chance of equilateral triangles. Figure 2 shows an example for the birdhouse cross-section.

4. We repeat the previous step m times until we have filled our domain D.

This procedure is not always perfect, for one thing it requires a tuning of the parameters m and n, and even then it can't always avoid sharp triangles, but it is an automatic way to generate a relatively good mesh. When working on domains with sharp edges, we need to be careful in our choice of n so that we don't aggressively cut off the aforementioned sharp edges,

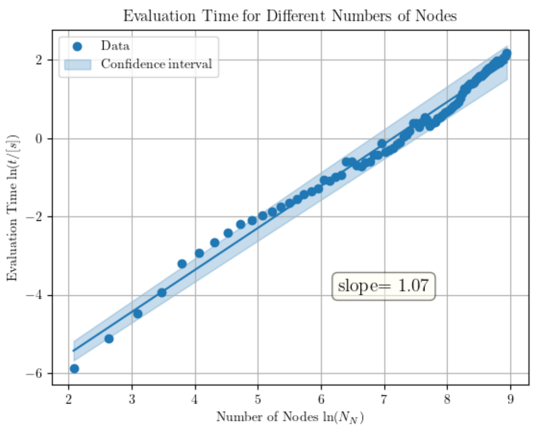

Figure 3: The evaluation time of the method as a function of the number of nodes. The number of triangles has a similar behavior since it is related by a factor.

For the triangulation of these points, once they have been generated, we use the built-in Delaunay triangulation under `scipy.spatial.Delaunay`.

In a similar fashion, the generated linear system of N_N equations is solved with the built-in method `numpy.linalg.solve`.

4 Results

Firstly, we want to test how fast out implementation of the Finite Element Method is. We will do this as a function of the number of vertices of our mesh. We can make this choice because the number of triangles is, asymptotically, just a factor times the number of nodes.

To perform the test we will use the square domain with a roughly uniform distribution of points. We expect that the speed of the implementation itself doesn't depend on the distribution of the mesh too much since the brunt of the work comes from solving the linear system of equations. Figure 3 shows the calculated behavior along with a power law fit. We see that the evaluation time grows approximately linearly with the number of nodes. At first one might find this surprising; It is true that the creating of the $N_N \times N_N$ matrix is linear in the number of triangles and thus in the number of nodes, but solving the system is generically an

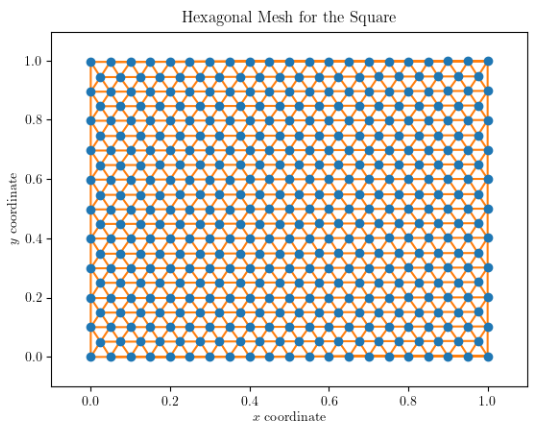

Figure 4: Uniform hexagonal mesh for the square domain.

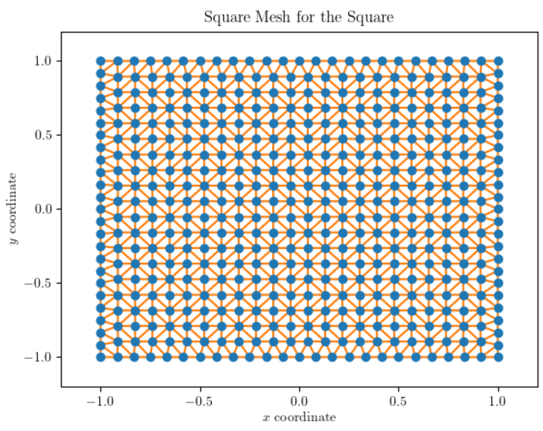

Figure 5: Uniform square mesh for the square domain.

$\mathcal{O}(N_N^3)$ procedure. But one should remember that at the core of the algorithm one vertex interacts with only a handful of neighbors so that with a proper indexing of the vertices one can bring the matrix into a few-diagonal form, which we know is solved in $\mathcal{O}(N_N)$. We did not do this explicitly, but a reindexing of the vertices corresponds to nothing more than some permutation of the rows[2] of the matrix. Some algorithms that solve systems of equations do such permutations to increase stability and speed, so the linear growth is not completely unexpected.

To examine the effects of the mesh, we will look at the following two tests. First we will look at the square domain and compare a uniform hexagonal mesh and a uniform square mesh with added boundary. These are shown in fig. 4 and fig. 5, respectively. The calculated Poiseuille coefficients are then

$$\begin{aligned} C_{\text{hex}} &= 0.87860 && (N_N = 431) \\ C_{\text{sq}} &= 0.87728 && (N_N = 492) \\ C_{\text{real}} &= 0.88327. \end{aligned} \tag{14}$$

The real coefficient above can be calculated analytically, by standard basis expansions in terms of sines for the Laplacian and is given by the expression

$$C = \frac{2\pi}{3} - \frac{2^7}{\pi^4} \sum_{n=1}^{\infty} \frac{1}{(2n-1)^5} \frac{\cosh((2n-1)\pi) - 1}{\sinh((2n-1)\pi)}. \tag{15}$$

The main difficulty in comparing the above two results is that we can't really get the number of points to be exactly the same in both cases, however even with fewer nodes we see that the hexagonal grid does a better job. This is a sign of the

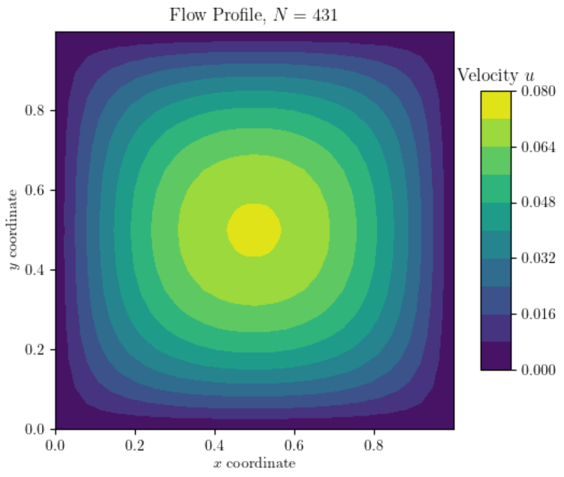

Figure 6: Flow profile for the square cross-section with $N_N = 431$ nodes using a hexagonal mesh.

[2]The matrix is, of course, symmetric so a permutation of the columns has to be accompanied by the corresponding permutation of the rows, in such a way as to keep the matrix symmetric

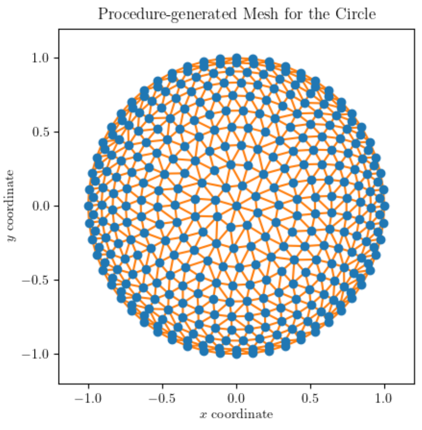

Figure 7: Mesh generated by the procedure above for the circular domain.

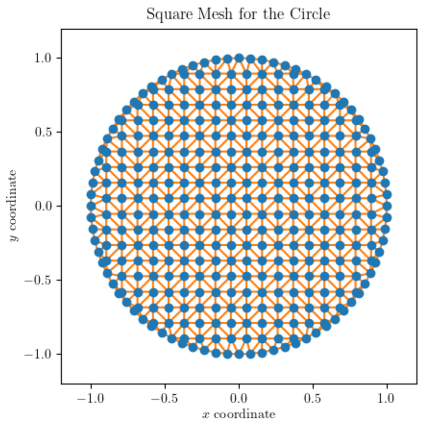

Figure 8: Uniform square mesh for the circular domain.

fact that triangles with angles close to 60° are ideal for the triangulation. Another thing to notice is that even with a relatively small number of points (~ 450), whose density isn't even chosen in a complicated way, we already have the coefficient to an error of 0.01. This is further corroborated by the flow profile shown in fig. 6, which looks as expected for the square domain.

The second test we will do is to compare the uniform square mesh with the mesh generated by the procedure described before using the density function

$$\rho(x) = 1 - (1 - x)^{1.5}, \tag{16}$$

which has a slight bias close to the boundary. The meshes are shown in figs. 7 and 8.

The calculated Poiseuille coefficients are

$$\begin{aligned} C_{\text{uniform}} &= 0.99733 \qquad (N_N = 359) \\ C_{\text{procedure}} &= 0.99716 \qquad (N_N = 356) \\ C_{\text{real}} &= 1.00000. \end{aligned} \tag{17}$$

We see that the non-uniform mesh is possibly slightly better than the uniform one but definitely not by a lot. This is probably because of the sharper triangles near the boundary of the non-uniform mesh pointing to the fact that the procedure above could be improved by tweaking how the number of points decreases from level to level, namely by increasing the density of points along the boundary and close to it.

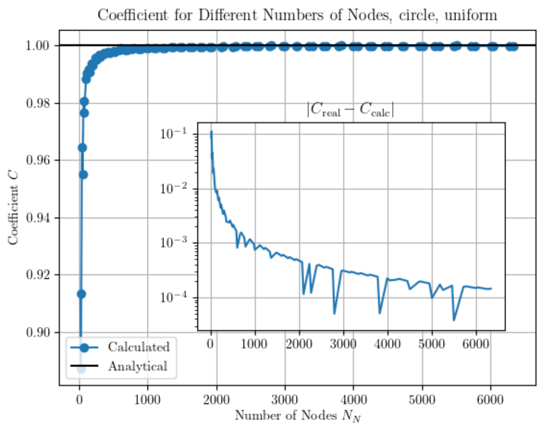

Figure 9: The calculated coefficient C for a circle as a function of the number of nodes. The inset plot shows how the difference goes to zero in a logarithmic scale.

Again, we notice that even our error compared to the real result is quite small even with ~ 350 points.

To precisely see how many points we need to get a good accuracy on the coefficient we calculate the coefficient with different numbers of nodes on a uniform mesh like the one in fig. 8 on a circular domain and see how fast it converges. The resulting graph is shown in fig. 9 We see that we quickly converge to good accuracy, but from the logarithmic scale

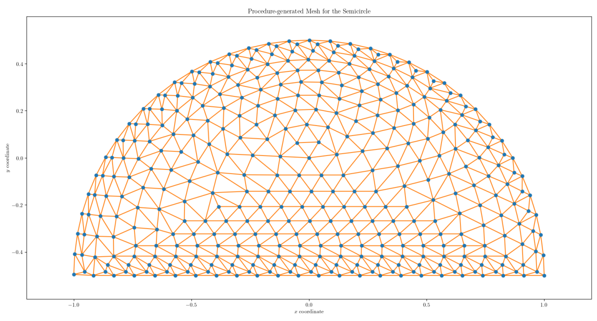

Figure 10: Procedure-generated mesh for the semicircular cross-section with $n = 59$ and $m = 10$.

we also see that the convergence is not exponential. We also see some dips, which are due to the fact that when generating the mesh some meshes get more sharp triangles along the boundary than others, depending on how the points match up.

Continuing on to the semicircular pipe we generate two meshes one uniform on by the procedure above and compute the flow profile and Poiseuille coefficient. For the non-uniform mesh we will use $N_N = 4551$ points similarly distributed as shown in fig. 10, while for the uniform mesh we will use $N_N = 4557$ points. The flow profiles calculated with these look almost the same and fig. 11 shows the profile from the non-uniform mesh. The calculated Poiseuille coefficients are

$$\begin{aligned} C_{\text{uniform}} &= 0.75745 \qquad (N_N = 4557) \\ C_{\text{procedure}} &= 0.75724 \qquad (N_N = 4551) \\ C_{\text{Galerkin}} &= 0.75772, \end{aligned} \tag{18}$$

where the final value is calculated in [1] using a Galerkin method with a tailored choice of basis. We see that the uniform mesh does slightly better and that our error from the real value is around $2 \cdot 10^{-4}$ in that case. A better choice of mesh with even less sharp triangles would lead to even better accuracy.

The outstanding fact is that the automatization of the mesh generation that avoids sharp triangles for higher and higher numbers of nodes is in a way a bigger bottleneck than solving the system of linear equations. We can choose the number of nodes to be in the ten thousands and still have the computation finish in a couple of seconds. If combined with an advanced mesh generation method that knows how to distribute points in the domain makes this method very powerful, and that's why it is so ubiquitous.

We repeat the procedure foe the birdhouse-shaped cross-section. A mesh generated by the procedure above is shown in fig. 12. It has a small number of points but calculations with it give $C = 0.576$. We notice that the mesh seems to have very sharp triangles outside of the domain, but this is just an artifact of the triangulation; it doesn't know what part is in the domain and what part is outside of it when it gets the points. These triangles won't contribute to the result in any way since all their vertices lie on the boundary and are

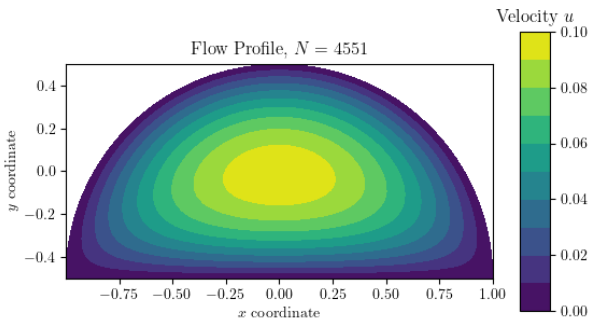

Figure 11: Flow profile computed with the procedure-generated mesh with $N_N = 4551$ points.

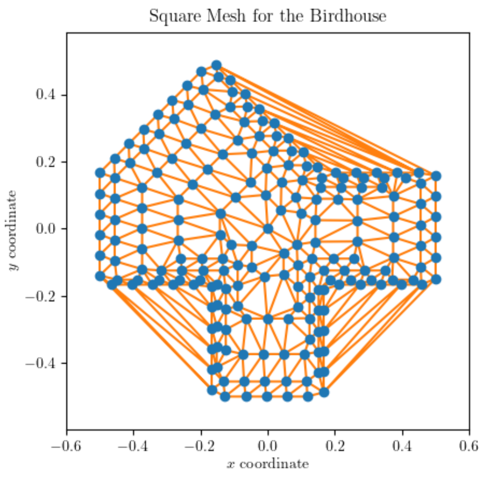

Figure 12: Procedure-generated mesh for the birdhouse-shaped cross-section with $n = 59$ and $m = 5$.

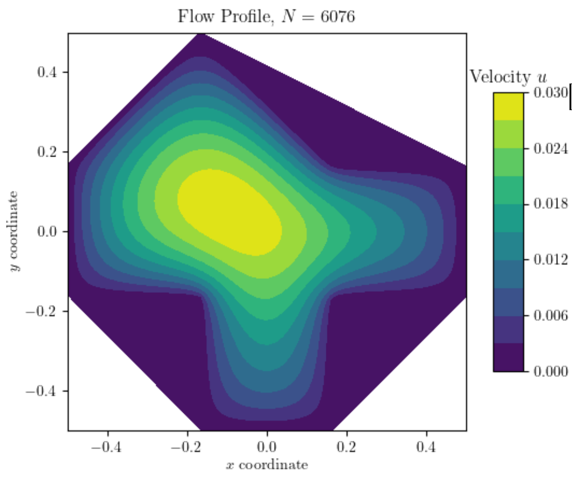

Figure 13: Flow profile for the birdhouse-shaped cross-section with $N_N = 6076$ nodes.

consequently ignored because of the Dirichlet condition. Similar things will happen for any domain with concave regions of boundary. Increasing the number of nodes to $N_N = 6076$ ($n = 110$, $m = 30$) gives a value of

$$\begin{aligned} C_{\text{procedure}} &= 0.5892 \\ C_{\text{SOR}} &= 0.593, \end{aligned} \tag{19}$$

where the SOR value was calculated in task 5. Calculations with the uniform mesh and different numbers of nodes of the order of 10^4 give results more in line with the FEM result above ranging between 0.587 and 0.589. This leads us to believe that the SOR result from last time is the one that is overestimating the coefficient and that the true value is closer to 0.589 with a conservative error estimate of around 0.002. The flow profile calculated in this way is shown in fig. 13. It also shows distorts the domain just because the triangulation introduces the redundant triangles outside of it, but otherwise looks similar to the flow profile calculated using the SOR method in task 5.

References

[1] Aleksandar Ivanov. *Galerkin Methods for PDEs*. [Last accessed: 14.04.2021]. URL: `https://github.com/ackiivanov/mfp-projects/blob/main/11.Galerkin%20Methods%20for%20Solving%20PDEs/gal.pdf`.

[2] Alkan Kabakçıoğlu. *Boundary Conditions for FEM*. [Last accessed: 14.04.2021]. URL: `http://home.ku.edu.tr/~akabakcioglu/teaching/math506/fem.pdf`.