Scientific Learning Final Project

June 2024

Team Members

- Sai Vijaya Bhargavi Akavarapu,
- Marta Celio,
- Bhanu Prakash Maruboyina

Part One: Traditional methods

1. We would like to derive the IMEX integration scheme of

$$\frac{\partial u}{\partial t} = \Sigma \Delta u - f(u)$$

with

$$f(u) = a(u - f_r)(u - f_t)(u - f_d)$$

If we define $u_k(x) = u(x, t_k)$, with t_k timesteps of distance δ_t from each other, then we can discretize the left-hand side as

$$\frac{\partial u}{\partial t} \approx \frac{u_{k+1} - u_k}{\delta_t}$$

As for the right-hand side, the discretization involves writing u as u_{k+1} when it appears as a derivative and as u_k when it appears inside f. So, the right-hand side becomes

$$\Sigma \Delta u_{k+1} - f(u_k)$$

Putting everything together, we get

$$\frac{u_{k+1} - u_k}{\delta_t} = \Sigma \Delta u_{k+1} - f(u_k)$$

Finally, rearranging the terms and multiplying both sides, we can write

$$\Sigma\delta_t\Delta u_{k+1} - u_{k+1} = \delta_t f(u_k) - u_k$$

2. Using the IMEX integration we just obtained, the weak formulation of the problem can be written as follows:

$$\int_\Omega \left(\Sigma\delta_t\Delta u_{k+1} - u_{k+1}\right) v = \int_\Omega \left(\delta_t f(u_k) - u_k\right) v$$

Using the property of the linearity of integrals, in combination with the divergence theorem, we can rewrite this as

$$\begin{aligned} -\Sigma\delta_t \int_\Omega \nabla u_{k+1}\nabla v - \int_\Omega u_{k+1}v &= \delta_t \int_\Omega f(u_k)v - \int_\Omega u_k v \\ \implies \Sigma\delta_t \int_\Omega \nabla u_{k+1}\nabla v + \int_\Omega u_{k+1}v &= -\delta_t \int_\Omega f(u_k)v + \int_\Omega u_k v \end{aligned}$$

Our goal is to find $u_{k+1} \in V$ such that this equation holds for any $v \in V$.

3. Let us discretize our problem in a space V_h of finite dimension. More specifically, for the purpose of this project, we'll divide the domain in polygonal (triangular) elements and take V_h such that every function in it is linear on each element, therefore being piece-wise linear on the full domain.

Our goal then becomes to find $u_{k+1,h} \in V_h$ such that

$$\Sigma\delta_t \int_\Omega \nabla u_{k+1,h}\nabla v_h + \int_\Omega u_{k+1,h}v_h = -\delta_t \int_\Omega f(u_{k,h})v_h + \int_\Omega u_{k,h}v_h$$

for all $v_h \in V_h$, with $u_{k,h}$ and $u_{k+1,h}$ being the approximations in V_h of u_k and u_{k+1}, respectively.

We can discretize this equation even further: since $u_{k,h}$ is a linear function, due to our definition of f, we know $f(u_{k,h})$ is a cubic function. By defining f_h as the approximation of $f(u_{k,h})$ in V_h, our goal then becomes to find $u_{k+1,h} \in V_h$ such that

$$\Sigma\delta_t \int_\Omega \nabla u_{k+1,h}\nabla v_h + \int_\Omega u_{k+1,h}v_h = -\delta_t \int_\Omega f_h v_h + \int_\Omega u_{k,h}v_h$$

for all $v_h \in V_h$.

Because V_h is a finite-dimensional space, a basis of functions $\{\varphi_i\}_i$ can be written for it. More specifically, writing $\{x_j\}_j$ as the set of vertices making up the elements of our domain, we choose $\{\varphi_i\}_i$ such that $\varphi_i(x_j) = \delta_{i,j}$

for every x_j. Since v_h can be written as a linear combination of those functions, thanks to the linearity of integrals, the problem is identical to finding $u_{k+1,h} \in V_h$ such that

$$\Sigma\delta_t \int_\Omega \nabla u_{k+1,h} \nabla\varphi_i + \int_\Omega u_{k+1,h}\varphi_i = -\delta_t \int_\Omega f_h\varphi_i + \int_\Omega u_{k,h}\varphi_i$$

for all φ_i. The same reasoning applies to f_h, $u_{k,h}$ and $u_{k+1,h}$: so, writing $f_h = \sum_j \hat{f}_{k,j}\varphi_j$, $u_{k,h} = \sum_j \hat{u}_{k,j}\varphi_j$ and $u_{k+1,h} = \sum_j \hat{u}_{k+1,j}\varphi_j$, we can write the problem as

$$\begin{aligned}\Sigma\delta_t \sum_j \hat{u}_{k+1,j} \int_\Omega \nabla\varphi_j \nabla\varphi_i + \sum_j \hat{u}_{k+1,j} \int_\Omega \varphi_j\varphi_i \\ = -\delta_t \sum_j \hat{f}_j \int_\Omega \varphi_j\varphi_i + \sum_j \hat{u}_{k,j} \int_\Omega \varphi_j\varphi_i\end{aligned}$$

Let's now recall that, by definition, each basis function φ_j is such that $\varphi_j(x_j) = 1$ and $\varphi_j(x_l) = 0$ for every $l \neq j$. This means that the j-th coefficient of every function in V_h is exactly equal to the evaluation of that function in x_j. Indeed, taking $u_{k,h}$ as an example,

$$u_{k,h}(x_j) = \sum_l \hat{u}_{k,l}\varphi_l(x_j) = \sum_l \hat{u}_{k,l}\delta_{l,j} = \hat{u}_{k,j}$$

The same goes for $u_{k+1,h}$ and $f(u_k)$. In the latter case, this allows us to rewrite its coefficients like so:

$$\hat{f}_j = f(u_{k,h}(x_j)) = f(\hat{u}_{k,j})$$

Substituting these coefficients in the equation yields us this:

$$\begin{aligned}\Sigma\delta_t \sum_j \hat{u}_{k+1,j} \int_\Omega \nabla\varphi_j \nabla\varphi_i + \sum_j \hat{u}_{k+1,j} \int_\Omega \varphi_j\varphi_i \\ = -\delta_t \sum_j f(\hat{u}_{k,j}) \int_\Omega \varphi_j\varphi_i + \sum_j \hat{u}_{k,j} \int_\Omega \varphi_j\varphi_i\end{aligned}$$

Finally, by defining $\hat{\mathbf{u}}_{k+1}$ and $\hat{\mathbf{u}}_k$ as the array of the coefficients $\hat{u}_{k+1,j}$ and $\hat{u}_{k,j}$, respectively, this equation can be written as

$$\delta_t A(\Sigma)\hat{\mathbf{u}}_{k+1} + M\hat{\mathbf{u}}_{k+1} = -\delta_t M f(\hat{\mathbf{u}}_k) + M\hat{\mathbf{u}}_k,$$
$$[A(\Sigma)]_{i,j} = \Sigma \int_\Omega \nabla\varphi_j \nabla\varphi_i \ ; \ [M]_{i,j} = \int_\Omega \varphi_j \varphi_i$$

4. A and M as matrices are defined in the files `assembleDiffusion.m` and `assembleReaction.m`, respectively. Their definitions follow the theory we have seen in in-class sessions.

   For the change of the definition of A, an extra parameter `S` was added in input to the function, representing the different diffusivity value in each element. `S` is intended to be an array of length equal to the number of elements: if a single number is passed instead, it is converted into an array of the right length with all elements equal to that number.

   We already know from the theory that we can split the definition of A over each element, by repeatedly defining a local matrix A_{loc} for each element and adding it to the respective submatrix of A. This is why we define Σ as an array `S` over the elements: if we know the value of Σ over each element, we can simply multiply each local matrix by the local numerical value of Σ. The code already provides flags for each element to distinguish the healthy domains from the diseased domains, in preparation for this exact procedure.

5. The code for the solution of the linear system is written in the file `fem1.m`. Aside from this MATLAB file, this project used `assembleDiffusion.m` and `assembleReaction.m` as explained earlier, as well as the other functions and classes provided in the appendix.

   At the top of the file, the code holds all important constants that can be tweaked if need be, such as the number of timesteps, the total time and the constants in the definition of f. Right after that, the array of values of Σ is set, by simply creating an array all made of equal elements and then changing the array at the indices corresponding to non-healthy elements, by checking the element flags in the mesh.

   A full matrix is initialized to store the approximated value of u at each time step, with the initial conditions being set in its first row. The matrices A and M are also initialized through their respective functions, then summed together to obtain the left-hand side matrix.

   Then, a for cycle starts: starting at time step 2, taking the solution at each previous time step (with the initial conditions corresponding to time step 1), the right-hand side for each time step is calculated and. The solution is found by solving its respective linear system, then stored in the matrix created for the purpose.

   The cycle also contains extra code in order to generate a plot and a video output for the time visualization.

6. Additional code was added in order to check if the left-hand side matrix is an M-matrix, if the potential ever exceeds 1 and what is the exact value of the activation time. The M-matrix check actually exploits a property of M-matrices: a matrix with non-positive off-diagonal elements is an M-matrix if and only if its LU decomposition results in two matrices with positive diagonals. Therefore, it's the LU decomposition of the matrix that gets checked.

   The checks actually have a leniency based on a certain threshold, equal to $2.2204 \cdot 10^{-16}$ (the floating-point relative accuracy) for the M-matrix check and 10^{-6} for the activation time check. The exceeded potential check also has a more lenient variation with a threshold of 10^{-10}: if the potential exceeds by some value, but less than this value, it will be reported in the table as "Barely".

   Here are the results:

dt	mesh	**activation time**	**is M-matrix?**	**potential exceeds?**
0.1	128	25.2ms	No	Yes
0.1	256	25.3ms	No	Yes
0.025	128	22.5ms	No	Barely
0.025	256	22.6ms	No	Barely

Table 1: Results for various dt and mesh configurations for $\Sigma_d = 10\Sigma_h$

dt	mesh	**activation time**	**is M-matrix?**	**potential exceeds?**
0.1	128	30.7ms	No	Yes
0.1	256	>35ms	No	Yes
0.025	128	27.5ms	No	Barely
0.025	256	35ms	No	Barely

Table 2: Results for various dt and mesh configurations for $\Sigma_d = \Sigma_h$

dt	mesh	**activation time**	**is M-matrix?**	**potential exceeds?**
0.1	128	34.4ms	No	Yes
0.1	256	>35ms	No	Yes
0.025	128	30.8ms	No	Yes
0.025	256	31.5ms	No	Yes

Table 3: Results for various dt and mesh configurations for $\Sigma_d = 0.1\Sigma_h$

   The activation time gets higher as the diffusivity gets lower. In all cases, the left-hand side matrix is not an M-matrix, and the potential exceeds the $[0, 1]$ interval. Finer meshes limit the amount by which the potential exceeds the interval, however.

7. Changing the matrix to its diagonal lumped counterpart makes it a lot

more likely to pass the M-matrix check. However, that still isn't guaranteed, since some of the rows of the matrix might have negative sum. Even when this change is made, the potential tends to exceed the $[0, 1]$ interval all the same.

Part Two: Physics informed neural networks (PINNs)

1. All the code for this part is in the file `pinns2.ipynb`.

   The loss function can simply be taken as the average of the squares of the residuals. Bringing everything in our equation to one side, we would like to have

   $$-\frac{\partial u}{\partial t} + \Sigma \Delta u - f(u) = 0$$

   and therefore this new left-hand side constitutes our residual, which we would like to minimize, for our predicted solution u_{NN}. Since u_{NN} is a tensor, calculating the derivatives can be done directly if u_{NN} is set to require a gradient. We can find the Laplacian of u_{NN} by summing the second derivatives of u_{NN} with respect to x and y respectively. The calculation of the residual is written down explicitly in the function `compute_pde_residual`, which gets called by `eval_loss_pde` in order to find the loss.

2. A dataset can be generated for training by sampling the points on the domain through a random uniform distribution. More specifically, a high number of points is sampled in each of the circles corresponding to the unhealthy parts of the domain, by sampling the angle and the length from the radius; then, a number of points is randomly sampled over the domain. The time coordinate is also sampled this way, being essentially coded as a third coordinate.

   After constructing the set of collocation points, the array of values of Σ is computed: this time each value corresponds to each collocation point. First, a check is performed to select all points contained in either of the three circles; then, the elements of Σ are set to either Σ_h or Σ_d depending on the results of the check. The check is performed by imposing the coordinates of the points of the domain inside the location of each circle. The array of values of Σ is separate from the matrix of collocation points coordinates.

3. On $t = 0$, the solution can be written as

   $$u(x, y, 0) = \chi_{\{x \leq 0.1 \wedge y \geq 0.9\}}$$

   Therefore, the initial condition is simply taken as

   $$g(x, y, t) = \chi_{\{x \leq 0.1 \wedge y \geq 0.9\}}$$

In order to impose hard constraints on the initial equations of $u_{\text{NN}}(x, y, t)$, after finding $\tilde{u}_{\text{NN}}$ through the nodes of the neural network, we must find u_{NN} by taking

$$u_{\text{NN}} = g + l\tilde{u}_{\text{NN}}$$

with l such that $l(x, y, 0) = 0$; $l(x, y, t) > 0$ for $t \neq 0$.

In this case, we simply take

$$l(x, y, t) = \frac{t}{T_f}$$

with T_f being the maximum time over the time interval. Doing so applies the initial conditions exactly at $t = 0$.

4. The model was trained using a standard Adam optimizer with a learning rate of 0.01.

5. The solution found through PINNs has a very different appearance to the one found through FEM, essentially increasing as time goes on. The FEM solution is closer to the expected solution for this problem, meaning that our PINN has not managed to properly learn the solution.

The video files showcasing the solutions are included in the .zip file under the names `solution_FEM.avi` and `solution_PINNs.mp4`.