# Group Final Project: Model for heat flow based on the Laplace equation

## 1 Instructions

This project is designed to assess your proficiency in algebra as well as your skill in Python implementation. Ensure that you provide Python code along with explanations for each line of code, using the syntax # to clarify your code's functionality. When you copy your code into Gradescope, please ensure that you also include the results generated by your code.

## 2 Collaborative Data Visualization Project

In this project, you will work together as a team to create an interactive data visualization using Python. This project aims to improve your teamwork, data analysis, and data visualization skills.

1. Collaboratively analyze the problem to find insights or trends. Each team member should work on a specific aspect of analysis. For example, one member can focus on descriptive linear algebra, another on data visualization, and so on.

2. Use a version control system (e.g., Colab: https://www.datarlabs.com/post/learn-to-use-google-colaboratory) to manage your project's code. Maintain a shared code repository where all team members can collaborate.

3. Utilize Piazza for regular updates and discussions. Schedule regular team meetings to discuss progress, challenges, and next steps.

4. Prepare a final presentation of your project (**This will be done during final exam week**). Each team member should participate in the presentation. You shall discuss the data, analysis, and the significance of your findings.

5. **Individual Evaluation:** At the end of the project, each team member should provide a self-assessment that outlines their contributions and experiences during the project.

6. **Peer Evaluation:** Perform peer evaluations to assess the contributions of your team members. Provide feedback on their collaboration, communication, and work quality.

7. **Final Report Submission:** Submit your project report, code, and presentation materials as a team. Include your individual and peer evaluations.

8. **Grading Criteria:**

   Collaborative Effort: 10%

   Analysis: 25%

   Data Visualization: 20%

   Python function implementation: 25%

   Project Documentation: 5%

   Communication and Collaboration: 10%

   Presentation: 5%

   **Note:** The assessment is based on your individual contributions as well as your ability to work effectively as part of a team.

9. In *Step #3 of the project, one half of the group implements Step 3a, while the other implements Step 3b. During the final presentation of the project, you will present and discuss your findings regarding the rapid convergence of these two iterative solvers, making a comparison.*

# 3 Steady Heat Flow

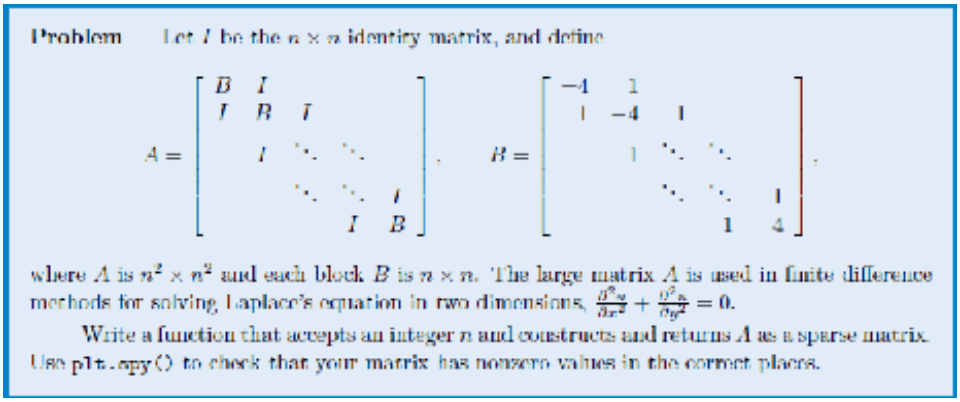
Laplace's equation is an important partial differential equation that arises often in both pure and applied mathematics. Laplace's equation can be used to model heat flow. Consider a square metal plate $\Omega$ where **the top and bottom borders are fixed at 100 degrees Celsius and the left and right sides are fixed at 0 Celsius**. Given these boundary conditions, we want to describe how heat diffuses through the rest of the plate. The solution to Laplace's equation describes the plate when it is in a steady state, meaning that the heat at a given part of the plate no longer changes with time. In two dimensions, the equation has the following form:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0,$$

where $u = u(x, y)$ is the temperature at $(x, y) \in \Omega$. Refer to lecture #10 for the full model and numerical implementation. Solve the large system $AU = b$ by the *Jacobi method*, where the vector $U[i, j]$ stands for an approximation of $u(x_i, y_i)$ at various cells $[i, j]$ of the grid matrix, and then plot the diffusion cell matrix in order to vizualize the diffusion of heat flow inside the plate $\Omega$.

# 4 Steps to complete the project

1. Implement the following step first (see lecture #10, week #7) based on scipy import sparse libraries:



**Problem** Let $I$ be the $n \times n$ identity matrix, and define

$$A = \begin{bmatrix} B & I & & & \\ I & B & I & & \\ & I & \ddots & \ddots & \\ & & \ddots & \ddots & I \\ & & & I & B \end{bmatrix}, \quad B = \begin{bmatrix} -4 & 1 & & & \\ 1 & -4 & 1 & & \\ & 1 & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & -4 \end{bmatrix},$$

where $A$ is $n^2 \times n^2$ and each block $B$ is $n \times n$. The large matrix $A$ is used in finite difference methods for solving Laplace's equation in two dimensions, $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$.

Write a function that accepts an integer $n$ and constructs and returns $A$ as a sparse matrix. Use `plt.spy()` to check that your matrix has nonzero values in the correct places.

You are tasked with constructing a large (*sparse*) matrix A that plays a vital role in solving Laplace's equation using finite difference methods in two dimensions. The matrix is composed of individual blocks, each of size n x n, with specific patterns on the diagonal and off-diagonal elements.

- **Hints:**
- Define two Python functions: create_large_diagonal_matrix(n) and create_large_identity_matrix to create the individual diagonal and off-diagonal blocks, respectively. These blocks will be added together to construct the final matrix A.
- In the create_large_diagonal_matrix(n) function, create the diagonal blocks, where each block contains the following pattern:

  A square block of size n x n.

  The main diagonal of the block consists of the value -4.

  The diagonal just above and just below the main diagonal should have values of 1.

  All other elements in the block should be 0.
- In the create_large_identity_matrix_off_diagonal(n) function, create the off-diagonal blocks, where each block contains the following pattern:

  A square block of size n x n.

  The main diagonal of the block should be an identity matrix of size n x n.

  All other elements in the block should be 0.
- Once you have defined these functions, combine the two functions to create the final matrix A of size $n^2$ x $n^2$

- Ensure your implementation works for any positive integer value of n.
- **Challenge**: Find a way to visualize the final matrix A and check if it has been constructed correctly.
- Define the vector data $b$ of size $n^2$ x 1. Check if it has been constructed properly.

2. Extract and display the diagonal matrix D (as a sparse matrix) which contains only the diagonal entries of the matrix A. Ensure your implementation works for any positive integer value of n. Calculate the inverse of the sparse diagonal matrix D.

3a The Jacobi Method is a simple but powerful method used for solving sparse large linear systems (i.e., Step 1) when the inverse $A^{-1}$ cannot be computed with standard tools from linear algebra (see lecture #10, iteration formula (15.2)). Write a function solveJacobi() that accepts a matrix A, a vector b, a convergence tolerance tol defaulting to 10^(-8), and a maximum number of iterations maxiter defaulting to 100. Implement the Jacobi method using (15.2), returning the approximate solution $x^{(k)}$ to the equation $Ax = b$:

$$\begin{aligned} \mathbf{x}^{(k+1)} &= D^{-1}(-(A-D)\mathbf{x}^{(k)} + \mathbf{b}) \\ &= D^{-1}(D\mathbf{x}^{(k)} - A\mathbf{x}^{(k)} + \mathbf{b}) \\ &= \mathbf{x}^{(k)} + D^{-1}(\mathbf{b} - A\mathbf{x}^{(k)}) \end{aligned} \tag{15.2}$$

Run the iteration until $\left\|b - Ax^{(k)}\right\|_\infty < tol$, and only iterate at most maxiter times, up from the initial guess $x^0$.

Your function should be robust enough to accept systems of any size.

3b The Gauss-Seidel method is essentially a slight modification of the Jacobi method (see lecture #10, formula (15.3)-(15.4)). Write a function GaussSeidel() that accepts a matrix A, a vector b, a convergence tolerance tol defaulting to 10^(-8), and a maximum number of iterations maxiter defaulting to 100. Implement the GS method using (15.3)-(15.4), returning the approximate solution $x^{(k)}$ to the equation $Ax = b$:

### Implementation

Because Gauss Seidel updates only one element of the solution vector at a time, the iteration cannot be summarized by a single matrix equation. Instead, the process is most generally described by the equation

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}\right). \tag{15.3}$$

Let $a_i$ be the $i$th row of $A$. The two sums closely resemble the regular vector product of $a_i$ and $x^{(k)}$ without the $i^{th}$ term $a_{ii}x_i^{(k)}$. This suggests the simplification

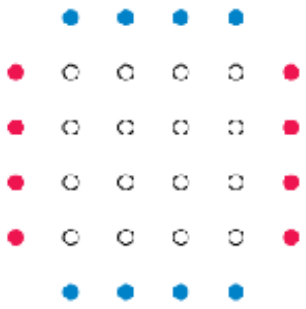
$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - a_i^T x^{(k)} + a_{ii}x_i^{(k)}\right)$$
$$= x_i^{(k)} + \frac{1}{a_{ii}}\left(b_i - a_i^T x^{(k)}\right). \quad (15.4)$$

One sweep through all the entries of x completes one iteration.

4 Run your function from Step 3, with the data vector $b$ from Step 1 to find the solution vector $x^{(k)} = U[i, j]$.

5 **Visualizing the Diffusion Flow Matrix**: you have a diffusion matrix obtained from a simulation, and your task is to visualize it using Python. The matrix represents a diffusion process, and you need to create a heatmap plot to understand the distribution of values. Hint: If plot=True, visualize the solution $U$ with a heatmap using plt.pcolormesh() (the colormap "coolwarm" is a good choice in this case). This shows the distribution of heat over the hot plate after it has reached its steady state.

- Rearrange the result vector $U$ as a full matrix to describe the cell grid for the plate. The matrix should be n x n in size. Each block in the matrix should be of size nx1, and the blocks must be arranged to create a new matrix. Ensure that the result_matrix variable stores the final matrix.
- Define padding values and constants. Your goal is to pad the matrix with the boundary values {0,100} to improve visualization (see figure below). You need to add rows and columns with specified values. Apply padding to the result_matrix using the np.pad function. Specify the number of rows and columns to pad at the top, bottom, left, and right. Also, define the constant values for padding, as specified in the code.



- Create a heatmap plot of the result_padded matrix. Use the plt.imshow function with the 'coolwarm' colormap and the 'nearest' interpolation method. Ensure that the color

map provides a clear representation of the values in the matrix. Add a color bar to the plot to indicate the value scale.

- Ensure your code works for any matrix size n and values in the diffusion matrix.