Lab 6

Iterative Solvers

Lab Objective: *Many real-world problems of the form $A\mathbf{x} = \mathbf{b}$ have tens of thousands of parameters. Solving such systems with Gaussian elimination or matrix factorizations could require trillions of floating point operations (FLOPs), which is of course infeasible. Solutions of large systems must therefore be approximated iteratively. In this lab, we implement three popular iterative methods for solving large systems: Jacobi, Gauss-Seidel, and Successive Over-Relaxation.*

Iterative Methods

The general idea behind any iterative method is to make an initial guess at the solution to a problem, apply a few easy computations to better approximate the solution, use that approximation as the new initial guess, and repeat until done. Throughout this lab, we use the notation $\mathbf{x}^{(k)}$ to denote the kth approximation for the solution vector $\mathbf{x}$ and $x_i^{(k)}$ to denote the i^{th} component of $\mathbf{x}^{(k)}$. With this notation, every iterative method can be summarized as

$$\mathbf{x}^{(k+1)} = f(\mathbf{x}^{(k)}), \tag{6.1}$$

where f is some function used to approximate the true solution $\mathbf{x}$.

In the best case, the iteration converges to the true solution ($\mathbf{x}^{(k)} \to \mathbf{x}$). In the worst case, the iteration continues forever without approaching the solution. Iterative methods therefore require carefully chosen *stopping criteria* to prevent iterating forever. The general approach is to continue until the difference between two consecutive approximations is sufficiently small, and to iterate no more than a specific number of times. More precisely, choose a very small $\epsilon > 0$ and an integer $N \in \mathbb{N}$, and update the approximation using Equation 6.1 until either

$$\|\mathbf{x}^{(k-1)} - \mathbf{x}^{(k)}\| < \epsilon \qquad \text{or} \qquad k > N. \tag{6.2}$$

The choices for ϵ and N are significant: a "large" ϵ (such as 10^{-6}) produces a less accurate result than a "small" ϵ (such 10^{-16}), but demands less computations; a small N (10) also potentially lowers accuracy, but detects and halts non-convergent iterations sooner than with a large N (10,000).

The Jacobi Method

The *Jacobi Method* is a simple but powerful method used for solving certain kinds of large linear systems. The main idea is simple: solve for each variable in terms of the others, then use the previous values to update each approximation. As a (very small) example, consider the following 3×3 system.

$$\begin{array}{rcrcrcr} 2x_1 & & & - & x_3 & = & 3 \\ -x_1 & + & 3x_2 & + & 2x_3 & = & 3 \\ & + & x_2 & + & 3x_3 & = & -1 \end{array}$$

Solving the first equation for x_1, the second for x_2, and the third for x_3 yields the following.

$$\begin{array}{rcl} x_1 & = & \frac{1}{2}(3 + x_3) \\ x_2 & = & \frac{1}{3}(3 + x_1 - 2x_3) \\ x_3 & = & \frac{1}{3}(-1 - x_2) \end{array}$$

Now begin with an initial guess $\mathbf{x}^{(0)} = [x_1^{(0)}, x_2^{(0)}, x_3^{(0)}]^\mathsf{T} = [0, 0, 0]^\mathsf{T}$. To compute the first approximation $\mathbf{x}^{(1)}$, use the entries of $\mathbf{x}^{(0)}$ as the variables on the right side of the previous equation.

$$\begin{array}{rcccccc} x_1^{(1)} & = & \frac{1}{2}(3 + x_3^{(0)}) & = & \frac{1}{2}(3 + 0) & = & \frac{3}{2} \\ x_2^{(1)} & = & \frac{1}{3}(3 + x_1^{(0)} - 2x_3^{(0)}) & = & \frac{1}{3}(3 + 0 - 0) & = & 1 \\ x_3^{(1)} & = & \frac{1}{3}(-1 - x_2^{(0)}) & = & \frac{1}{3}(-1 - 0) & = & -\frac{1}{3} \end{array}$$

So $\mathbf{x}^{(1)} = [\frac{3}{2}, 1, -\frac{1}{3}]^\mathsf{T}$. Computing $\mathbf{x}^{(2)}$ is similar.

$$\begin{array}{rcccccc} x_1^{(2)} & = & \frac{1}{2}(3 + x_3^{(1)}) & = & \frac{1}{2}(3 - \frac{1}{3}) & = & \frac{4}{3} \\ x_2^{(2)} & = & \frac{1}{3}(3 + x_1^{(1)} - 2x_3^{(1)}) & = & \frac{1}{3}(3 + \frac{3}{2} + \frac{2}{3}) & = & \frac{31}{18} \\ x_3^{(2)} & = & \frac{1}{3}(-1 - x_2^{(1)}) & = & \frac{1}{3}(-1 - 1) & = & -\frac{2}{3} \end{array}$$

The process is repeated until at least one of the two stopping criteria in Equation 6.2 is met. For this particular problem, convergence to 8 decimal places ($\epsilon = 10^{-8}$) is reached in 29 iterations.

	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$
$\mathbf{x}^{(0)}$	0	0	0
$\mathbf{x}^{(1)}$	1.5	1	-0.33333
$\mathbf{x}^{(2)}$	1.33333333	1.72222222	-0.66666667
$\mathbf{x}^{(3)}$	1.16666667	1.88888889	-0.90740741
$\mathbf{x}^{(4)}$	1.04629630	1.99382716	-0.96296296
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\mathbf{x}^{(28)}$	0.99999999	2.00000001	-0.99999999
$\mathbf{x}^{(29)}$	1	2	-1

Matrix Representation

The iterative steps performed above can be expressed in matrix form. First, decompose A into its diagonal entries, its entries below the diagonal, and its entries above the diagonal, as $A = D + L + U$.

$$\underset{D}{\begin{bmatrix} a_{11} & 0 & \dots & 0 \\ 0 & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{nn} \end{bmatrix}} \quad \underset{L}{\begin{bmatrix} 0 & 0 & \dots & 0 \\ a_{21} & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ a_{n1} & \dots & a_{n,n-1} & 0 \end{bmatrix}} \quad \underset{U}{\begin{bmatrix} 0 & a_{12} & \dots & a_{1n} \\ 0 & 0 & \ddots & \vdots \\ \vdots & \vdots & \ddots & a_{n-1,n} \\ 0 & 0 & \dots & 0 \end{bmatrix}}$$

With this decomposition, we solve for $\mathbf{x}$ in the following way.

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

Now using $\mathbf{x}^{(k)}$ as the variables on the right side of the equation to produce $\mathbf{x}^{(k+1)}$ on the left, and noting that $L + U = A - D$, we have the following.

$$\begin{aligned} \mathbf{x}^{(k+1)} &= D^{-1}(-(A - D)\mathbf{x}^{(k)} + \mathbf{b}) \\ &= D^{-1}(D\mathbf{x}^{(k)} - A\mathbf{x}^{(k)} + \mathbf{b}) \\ &= \mathbf{x}^{(k)} + D^{-1}(\mathbf{b} - A\mathbf{x}^{(k)}) \end{aligned} \tag{6.3}$$

There is a potential problem with Equation 6.3: calculating a matrix inverse is the cardinal sin of numerical linear algebra, yet the equation contains D^{-1}. However, since D is a diagonal matrix, D^{-1} is also diagonal, and is easy to compute.

$$D^{-1} = \begin{bmatrix} \frac{1}{a_{11}} & 0 & \dots & 0 \\ 0 & \frac{1}{a_{22}} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{1}{a_{nn}} \end{bmatrix}$$

Because of this, the Jacobi method requires that A have nonzero diagonal entries.

The diagonal D can be represented by the 1-dimensional array $\mathbf{d}$ of the diagonal entries. Then the matrix multiplication $D\mathbf{x}$ is equivalent to the component-wise vector multiplication $\mathbf{d} * \mathbf{x} = \mathbf{x} * \mathbf{d}$. Likewise, the matrix multiplication $D^{-1}\mathbf{x}$ is equivalent to the component-wise "vector division" $\mathbf{x}/\mathbf{d}$.

```
>>> import numpy as np

>>> D = np.array([[2,0],[0,16]])    # Let D be a diagonal matrix.
>>> d = np.diag(D)                  # Extract the diagonal as a 1-D array.
>>> x = np.random.random(2)
>>> np.allclose(D.dot(x), d*x)
True
```

Problem 1. Write a function that accepts a matrix A, a vector $\mathbf{b}$, a convergence tolerance ϵ, and a maximum number of iterations N. Implement the Jacobi method using Equation 6.3, returning the approximate solution to the equation $A\mathbf{x} = \mathbf{b}$.

Run the iteration until $\|\mathbf{x}^{(k-1)} - \mathbf{x}^{(k)}\|_\infty < \epsilon$, and only iterate at most N times. Avoid using `la.inv()` to calculate D^{-1}, but use `la.norm()` to calculate the vector ∞-norm $\|\mathbf{x}\|_\infty = \sup |x_i|$.

```
>>> from scipy import linalg as la

>>> x = np.random.random(10)
>>> la.norm(x, ord=np.inf)          # Use la.norm() for ||x||.
0.74623726404168045
>>> np.max(np.abs(x))               # Use pure NumPy for ||x||.
0.74623726404168045
```

Your function should be robust enough to accept systems of any size. To test your function, use the following function to generate an $n \times n$ matrix A for which the Jacobi method is guaranteed to converge.

```
def diag_dom(n, num_entries=None):
    """Generate a strictly diagonally dominant nxn matrix.

    Inputs:
        n (int): the dimension of the system.
        num_entries (int): the number of nonzero values
            Defaults to n^(3/2)-n.

    Returns:
        A ((n,n) ndarray): An nxn strictly diagonally dominant matrix.
    """
    if num_entries is None:
        num_entries = int(n**1.5) - n
    A = np.zeros((n,n))
    rows = np.random.choice(np.arange(0,n), size=num_entries)
    cols = np.random.choice(np.arange(0,n), size=num_entries)
    data = np.random.randint(-4, 4, size=num_entries)
    for i in xrange(num_entries):
        A[rows[i], cols[i]] = data[i]
    for i in xrange(n):
        A[i,i] = np.sum(np.abs(A[i])) + 1
    return A
```

Generate a random $\mathbf{b}$ with `np.random.random()`. Run the iteration, then check that $A\mathbf{x}^{(k)}$ and $\mathbf{b}$ are close using `np.allclose()`.

Also test your function on random $n \times n$ matrices. If the iteration is non-convergent, the successive approximations will have increasingly large entries.

Convergence

Most iterative methods only converge under certain conditions. For the Jacobi method, convergence mostly depends on the nature of the matrix A. If the entries a_{ij} of A satisfy the property

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}| \text{ for all } i = 1, 2, \ldots, n,$$

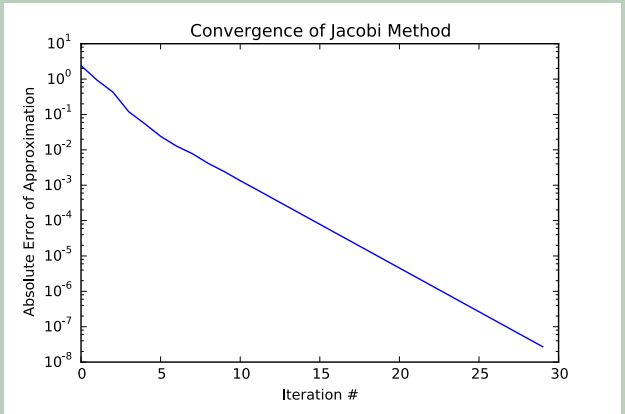
then A is called *strictly diagonally dominant* (for example, `diag_dom()` in Problem 1 generates a strictly diagonally dominant $n \times n$ matrix). If this is the case, then the Jacobi method always converges, regardless of the initial guess $\mathbf{x}_0$.[1] Other iterative methods, such as Newton's method, depend mostly on the initial guess.

There are a few ways to determine whether or not an iterative method is converging. For example, since the approximation $\mathbf{x}^{(k)}$ should satisfy $A\mathbf{x}^{(k)} \approx \mathbf{b}$, the normed difference $\|A\mathbf{x}^{(k)} - \mathbf{b}\|_\infty$ should be small. This value is called the *absolute error* of the approximation. If the iterative method converges, the absolute error should decrease to ϵ.

Problem 2. Modify your Jacobi method function in the following ways:

1. Add a keyword argument called `plot`, defaulting to `False`.
2. Keep track of the absolute error $\|A\mathbf{x}^{(k)} - \mathbf{b}\|_\infty$ of the approximation for each value of k.
3. If `plot` is `True`, produce a lin-log plot the error against iteration count (use `plt.semilogy()` instead of `plt.plot()`). Return the approximate solution $\mathbf{x}$ even if `plot` is `True`.

If the iteration converges, your plot should resemble the following figure.



[1] Although this seems like a strong requirement, most real-world linear systems can be represented by strictly diagonally dominant matrices.

The Gauss-Seidel Method

The Gauss-Seidel method is essentially a slight modification of the Jacobi method. The main difference is that in Gauss-Seidel, new information is used immediately. Consider the same system as in the previous section.

$$\begin{array}{rcrcrcr} 2x_1 & & & - & x_3 & = & 3 \\ -x_1 & + & 3x_2 & + & 2x_3 & = & 3 \\ & + & x_2 & + & 3x_3 & = & -1 \end{array}$$

As with the Jacobi method, solve for x_1 in the first equation, x_2 in the second equation, and x_3 in the third equation.

$$\begin{array}{rcl} x_1 & = & \frac{1}{2}(3 + x_3) \\ x_2 & = & \frac{1}{3}(3 + x_1 - 2x_3) \\ x_3 & = & \frac{1}{3}(-1 - x_2) \end{array}$$

Use $\mathbf{x}^{(0)}$ to compute $x_1^{(1)}$ in the first equation.

$$x_1^{(1)} = \frac{1}{2}(3 + x_3^{(0)}) = \frac{1}{2}(3 + 0) = \frac{3}{2}$$

Now, however, use the updated value of $x_1^{(1)}$ in the calculation of $x_2^{(1)}$.

$$x_2^{(1)} = \frac{1}{3}(3 + x_1^{(1)} - 2x_3^{(0)}) = \frac{1}{3}(3 + \frac{3}{2} - 0) = \frac{3}{2}$$

Likewise, use the updated values of $x_1^{(1)}$ and $x_2^{(1)}$ to calculate $x_3^{(1)}$.

$$x_3^{(1)} = \frac{1}{3}(-1 - x_2^{(1)}) = \frac{1}{3}(-1 - \frac{3}{2}) = -\frac{5}{6}$$

This process of using calculated information immediately is called *forward substitution*, and causes the algorithm to (generally) converge much faster.

	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$
$x^{(0)}$	0	0	0
$x^{(1)}$	1.5	1.5	-0.833333
$x^{(2)}$	1.08333333	1.91666667	-0.97222222
$x^{(3)}$	1.01388889	1.98611111	-0.99537037
$x^{(4)}$	1.00231481	1.99768519	-0.9992284
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$x^{(11)}$	1.00000001	1.99999999	-1
$x^{(12)}$	1	2	-1

Notice that Gauss-Seidel converged in less than half as many iterations.

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 following equation.

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{(k)} - \sum_{j>i} a_{ij} x_j^{(k)} \right) \tag{6.4}$$

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

$$\begin{aligned} x_i^{(k+1)} &= \frac{1}{a_{ii}} \left(b_i - A_i^\mathsf{T} \mathbf{x}^{(k)} + a_{ii} x_i^{(k)} \right) \\ &= x_i^{(k)} + \frac{1}{a_{ii}} \left(b_i - A_i^\mathsf{T} \mathbf{x}^{(k)} \right) \end{aligned} \tag{6.5}$$

One sweep through all the entries of $\mathbf{x}$ completes one iteration.

Problem 3. Write a function that accepts a matrix A, a vector $\mathbf{b}$, a convergence tolerance ϵ, a maximum number of iterations N, and a keyword argument `plot` that defaults to `False`. Implement the Gauss-Seidel method using Equation 6.5, returning the approximate solution to the equation $A\mathbf{x} = \mathbf{b}$.

Use the same stopping criterion as in Problem 1. Also keep track of the absolute errors of the iteration, as in Problem 2. If `plot` is `True`, plot the error against iteration count. Use `diag_dom()` to generate test cases.

Achtung!

Since the Gauss-Seidel algorithm operates on the approximation vector in place (modifying it one entry at a time), the previous approximation $\mathbf{x}^{(k-1)}$ must be stored at the beginning of the kth iteration in order to calculate $\|\mathbf{x}^{(k-1)} - \mathbf{x}^{(k)}\|_\infty$. Additionally, since NumPy arrays are mutable, the past iteration must be stored as a **copy**.

```
>>> x0 = np.random.random(5)        # Generate a random vector.
>>> x1 = x0                         # Attempt to make a copy.
>>> x1[3] = 1000                    # Modify the "copy" in place.
>>> np.allclose(x0, x1)             # But x0 was also changed!
True
# Instead, make a copy of x0 when creating x1.
>>> x0 = np.copy(x1)                # Make a copy.
>>> x1[3] = -1000
>>> np.allclose(x0, x1)
False
```

Convergence

Whether or not the Gauss-Seidel converges or not also depends on the nature of A. If all of the eigenvalues of A are positive, A is called *positive definite*. If A is positive definite *or* if it is strictly diagonally dominant, then the Gauss-Seidel method converges regardless of the initial guess $\mathbf{x}^{(0)}$.

Problem 4. The Gauss-Seidel method is faster than the standard system solver used by `la.solve()` if the system is sufficiently large and sufficiently sparse. For each vale of $n = 5,\ 6,\ \ldots,\ 11$, generate a random $2^n \times 2^n$ matrix A using `diag_dom()` and a random 2^n vector $\mathbf{b}$. Time how long it takes to solve $A\mathbf{x} = \mathbf{b}$ using your Gauss-Seidel function from Problem 3, and how long it takes to solve using `la.solve()`.

Plot the times against the system size. Use log scales if appropriate.

Solving Sparse Systems Iteratively

Iterative solvers are best suited for solving very large sparse systems. However, using the Gauss-Seidel method on sparse matrices requires translating code from NumPy to `scipy.sparse`. The algorithm is the same, but there are some functions that are named differently between these two packages.

Problem 5. Write a new function that accepts a **sparse** matrix A, a vector $\mathbf{b}$, a convergence tolerance ϵ, and a maximum number of iterations N (plotting the convergence is not required for this problem). Implement the Gauss-Seidel method using Equation 6.5, returning the approximate solution to the equation $A\mathbf{x} = \mathbf{b}$. Use the usual stopping criterion.

The Gauss-Seidel method requires extracting the rows A_i from the matrix A and computing $A_i^\mathsf{T}\mathbf{x}$. There are many ways to do this that cause some fairly serious runtime issues, so we provide the code for this specific portion of the algorithm.

```
# Slice the i-th row of A and dot product the vector x.
rowstart = A.indptr[i]
rowend = A.indptr[i+1]
Aix = np.dot(A.data[rowstart:rowend], x[A.indices[rowstart:rowend]])
```

To test your function, cast the result of `diag_dom()` as a sparse matrix.

```
from scipy import sparse

>>> A = sparse.csr_matrix(diag_dom(50000))
>>> b = np.random.random(50000)
```

Successive Over-Relaxation (SOR)

Some systems meet the requirements for convergence with the Gauss-Seidel method, but that do not converge very quickly. A slightly altered version of the Gauss-Seidel method, called *Successive Over-Relaxation*, can result in faster convergence. This is achieved by introducing a *relaxation factor*, ω. The iterative equation for Gauss-Seidel, Equation 6.4 becomes the following.

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

Simplifying the equation results in the following.

$$x_i^{(k+1)} = x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - A_i^{\mathsf{T}}\mathbf{x}^{(k)}\right) \tag{6.6}$$

Note that when $\omega = 1$, Successive Over-Relaxation reduces to Gauss-Seidel.

Problem 6. Write a function that accepts a *sparse* matrix A, a vector $\mathbf{b}$, a convergence tolerance ϵ, and a maximum number of iterations N. Implement Successive Over-Relaxation using Equation 6.6, returning the approximate solution to the equation $A\mathbf{x} = \mathbf{b}$. Use the usual stopping criterion.
(Hint: this requires changing only one line of code from the sparse Gauss-Seidel function.)

Finite Difference Method

Laplace's equation is an important partial differential equation that arises often in both pure and applied mathematics. In two dimensions, the equation has the following form.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y) \tag{6.7}$$

Laplace's equation can be used to model heat flow. Consider a square metal plate where the top and bottom sides are fixed at 0° Celsius and the left and right sides are fixed at 100° Celsius. Given these boundary conditions, we want to describe how heat diffuses through the rest of the plate. If $f(x,y) = 0$, then 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.

It is possible to solve Equation 6.7 analytically when $f(x,y) = 0$. Instead, however, we use a *finite difference method* to solve the problem numerically. To begin, we impose a discrete, square grid on the plate (see Figure 6.1). Denote the points on the grid by $u_{i,j}$. Then the interior points of the grid can be numerically approximated as follows:

$$\frac{\partial^2 u_{i,j}}{\partial x^2} + \frac{\partial^2 u_{i,j}}{\partial y^2} \approx \frac{1}{h^2}\left(-4u_{i,j} + u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}\right), \tag{6.8}$$

where h is the distance between $u_{i,j}$ and $u_{i+1,j}$ (and between $u_{i,j}$ and $u_{i,j+1}$).[2]

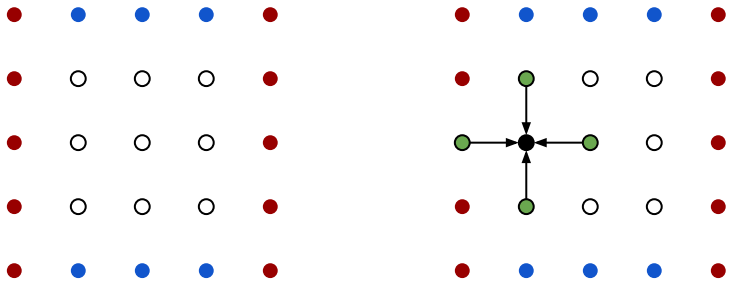


Figure 6.1: On the left, an example of a 6×6 grid where the red dots are hot boundary zones and the blue dots are cold boundary zones. On the right, the green dots are the neighbors of the interior black dot that are used to approximate the heat at the black dot.

This problem can be formulated as a linear system. Suppose the grid has exactly $(n+2)\times(n+2)$ entries. Then the interior of the grid is $n\times n$, and can be flattened into an $n^2 \times 1$ vector $\mathbf{u}$. The entire first row goes first, then the second row, proceeding to the n^{th} row.

$$\mathbf{u} = [u_{1,1},\ u_{1,2},\ \cdots,\ u_{1,n},\ u_{2,1},\ u_{2,2},\ \cdots,\ u_{2,n}\ \ldots,\ u_{n,n}]^\mathsf{T}$$

From Equations 6.7 and 6.8 with $f(x, y) = 0$, we have the following for an interior point $u_{i,j}$:

$$-4u_{i,j} + u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} = 0. \tag{6.9}$$

If any of the neighbors to $u_{i,j}$ is a boundary point on the grid, its value is already determined. For example, for $u_{3,1}$, the neighbor $u_{3,0} = 100$, so

$$-4u_{3,1} + u_{4,1} + u_{2,1} + u_{3,2} = -100.$$

The constants on the right side of the equation become the $n^2 \times 1$ vector $\mathbf{b}$.

For example, writing Equation 6.9 for the 9 interior points of the grid in Figure

[2]The derivation of Equation 6.8 will be studied in the lab on numerical differentiation.

6.1 result the a 9×9 system, $A\mathbf{u} = \mathbf{b}$.

$$\begin{bmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{bmatrix} \begin{bmatrix} u_{1,1} \\ u_{1,2} \\ u_{1,3} \\ u_{2,1} \\ u_{2,2} \\ u_{2,3} \\ u_{3,1} \\ u_{3,2} \\ u_{3,3} \end{bmatrix} = \begin{bmatrix} -100 \\ 0 \\ -100 \\ -100 \\ 0 \\ -100 \\ -100 \\ 0 \\ -100 \end{bmatrix}$$

More generally, for any positive integer n, the corresponding system $A\mathbf{u} = \mathbf{b}$ can be expressed as follows.

$$A = \begin{bmatrix} B_1 & I & & & \\ I & B_2 & I & & \\ & I & \ddots & \ddots & \\ & & \ddots & \ddots & I \\ & & & I & B_n \end{bmatrix}, \quad B_i = \begin{bmatrix} -4 & 1 & & & \\ 1 & -4 & 1 & & \\ & 1 & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & -4 \end{bmatrix},$$

where each B_i is $n \times n$. All nonzero entries of $\mathbf{b}$ correspond to interior points that touch the left or right boundaries (since the top and bottom boundaries are held constant at 0).

Problem 7. Write a function that accepts an integer n for the number of interior grid points. Return the corresponding sparse matrix A and NumPy array b.
(Hint: Consider using `scipy.sparse.block_diag` and the `setdiag()` method of scipy sparse matrices for dynamically creating the matrix A.)

Problem 8. To demonstrate how convergence is affected by the value of ω in SOR, time your function from Problem 6 with $\omega = 1,\ 1.05,\ 1.1,\ \ldots, 1.9, 1.95$ using the A and $\mathbf{b}$ generated by problem 7 with $n = 20$. Plot the times as a function of ω.

Note that the matrix A is not strictly diagonally dominant. However, A is positive definite, so the algorithm will converge. Unfortunately, convergence for these kinds of systems usually require more iterations than for strictly diagonally dominant systems. Therefore, set `tol=1e-2` and `maxiters = 1000` on the SOR function.

Problem 9. Write a function that accepts an integer n. Use Problem 7 to generate the corresponding system $A\mathbf{u} = \mathbf{b}$, then solve the system using SciPy's sparse system solver, `scipy.sparse.linalg.spsolve()` (`spla.spsolve()` from the previous lab). Visualize the solution using a heatmap using `np.meshgrid()` and `plt.pcolormesh()` (`"seismic"` is a good color map in this case). This shows the distribution of heat over the hot plate after it has reached its steady state. Note that the solution vector $\mathbf{u}$ must be reshaped to properly visualize the result.