15 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 are often useful to solve large systems of equations. In this lab, let $\mathbf{x}^{(k)}$ denote the kth iteration of the iterative method for solving the problem $A\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$. Furthermore, let x_i be the ith component of $\mathbf{x}$ so that $x_i^{(k)}$ is the ith component of $\mathbf{x}$ in the kth iteration. Like other iterative methods, there are two stopping parameters: a very small $\varepsilon > 0$ and an integer $N \in \mathbb{N}$. Iterations continue until either

$$\|\mathbf{x}^{(k-1)} - \mathbf{x}^{(k)}\| < \varepsilon \qquad \text{or} \qquad k > N. \tag{15.1}$$

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 3×3

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

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

$$\begin{aligned} x_1 &= \tfrac{1}{2}(3 + x_3), \\ x_2 &= \tfrac{1}{3}(3 + x_1 - 2x_3), \\ x_3 &= \tfrac{1}{3}(-1 - x_2). \end{aligned}$$

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 equations:

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

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

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

The process is repeated until at least one of the two stopping criteria in (15.1) is met. For this particular problem, convergence to 8 decimal places ($\varepsilon = 10^{-8}$) 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	−0.33333333
$\mathbf{x}^{(2)}$	1.33333333	1.72222222	−0.66666667
$\mathbf{x}^{(3)}$	1.16666667	1.88888889	−0.90740741
$\mathbf{x}^{(4)}$	1.04629630	1.99382716	−0.96296296
⋮	⋮	⋮	⋮
$\mathbf{x}^{(28)}$	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, $\mathbf{x}$ can be expressed 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.

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

There is a potential problem with (15.2): 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" $\mathbf{x}/\mathbf{d}$.

Problem 1. Write a function that accepts a matrix A, a vector $\mathbf{b}$, 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 to the equation $A\mathbf{x} = \mathbf{b}$.

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

Your function should be robust enough to accept systems of any size. To test your function, generate a random $\mathbf{b}$ with `np.random.random()` and use the following function to generate an $n \times n$ matrix A for which the Jacobi method is guaranteed to converge. Run the iteration, then check that $A\mathbf{x}^{(k)}$ and $\mathbf{b}$ are close using `np.allclose()`.

```
def diag_dom(n, num_entries=None, as_sparse=False):
    """Generate a strictly diagonally dominant (n, n) matrix.
    Parameters:
        n (int): The dimension of the system.
        num_entries (int): The number of nonzero values.
            Defaults to n^(3/2)-n.
        as_sparse: If True, an equivalent sparse CSR matrix is returned.
    Returns:
        A ((n,n) ndarray): A (n, n) strictly diagonally dominant matrix.
    """
    if num_entries is None:
        num_entries = int(n**1.5) - n
    A = sparse.dok_matrix((n,n))
    rows = np.random.choice(n, size=num_entries)
    cols = np.random.choice(n, size=num_entries)
    data = np.random.randint(-4, 4, size=num_entries)
    for i in range(num_entries):
        A[rows[i], cols[i]] = data[i]
    B = A.tocsr()          # convert to row format for the next step
    for i in range(n):
        A[i,i] = abs(B[i]).sum() + 1
    return A.tocsr() if as_sparse else A.toarray()
```

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

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

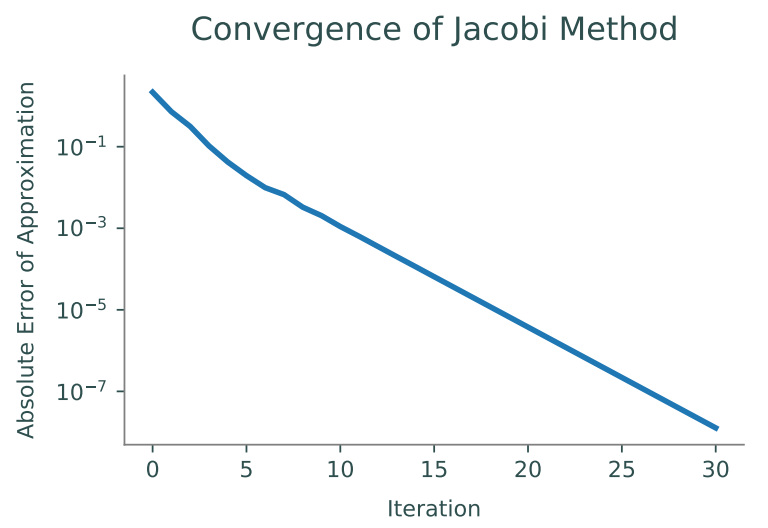
then A is called *strictly diagonally dominant* (`diag_dom()` in Problem 1 generates a strictly diagonally dominant $n \times n$ matrix). If this is the case,[1] then the Jacobi method always converges, regardless of the initial guess $\mathbf{x}_0$. This is a very different convergence result than many other iterative methods such as Newton's method where convergence is highly sensitive to 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 at each iteration.
3. If `plot` is True, produce a lin-log plot (use `plt.semilogy()`) of the error against iteration count. Remember to still return the approximate solution $\mathbf{x}$.

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. As an example, consider again the system from 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:

$$\begin{aligned} x_1 &= \tfrac{1}{2}(3 + x_3), \\ x_2 &= \tfrac{1}{3}(3 + x_1 - 2x_3), \\ x_3 &= \tfrac{1}{3}(-1 - x_2). \end{aligned}$$

Using $\mathbf{x}^{(0)}$ to compute $x_1^{(1)}$ in the first equation as before,

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

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	−0.83333333
$x^{(2)}$	1.08333333	1.91666667	−0.97222222
$x^{(3)}$	1.01388889	1.98611111	−0.99537037
$x^{(4)}$	1.00231481	1.99768519	−0.99922840
⋮	⋮	⋮	⋮
$x^{(11)}$	1.00000001	1.99999999	−1
$x^{(12)}$	1	2	−1

Notice that Gauss-Seidel converges in less than half as many iterations as Jacobi does for this system.

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

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

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

$$\begin{aligned} x_i^{(k+1)} &= \frac{1}{a_{ii}}\left(b_i - \mathbf{a}_i^\mathsf{T}\mathbf{x}^{(k)} + a_{ii}x_i^{(k)}\right) \\ &= x_i^{(k)} + \frac{1}{a_{ii}}\left(b_i - \mathbf{a}_i^\mathsf{T}\mathbf{x}^{(k)}\right). \end{aligned} \tag{15.4}$$

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 `tol` defaulting to 10^{-8}, a maximum number of iterations `maxiter` defaulting to 100, and a keyword argument `plot` that defaults to `False`. Implement the Gauss-Seidel method using (15.4), 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 method converges 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 $\mathbf{x}^{(0)}$.

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.[2]

Problem 4. Write a new function that accepts a **sparse** matrix A, a vector $\mathbf{b}$, a convergence tolerance `tol`, and a maximum number of iterations `maxiter` (plotting the convergence is not required for this problem). Implement the Gauss-Seidel method using (15.4), returning the approximate solution to the equation $A\mathbf{x} = \mathbf{b}$. Use the usual default 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.

```
# Get the indices of where the i-th row of A starts and ends if the
# nonzero entries of A were flattened.
rowstart = A.indptr[i]
rowend = A.indptr[i+1]

# Multiply only the nonzero elements of the i-th row of A with the
# corresponding elements of x.
Aix = A.data[rowstart:rowend] @ x[A.indices[rowstart:rowend]]
```

To test your function, remeber to call `diag_dom()` using `as_sparse=True`

```
>>> A = diag_dom(50000, as_sparse=True)
>>> b = np.random.random(50000)
```

Successive Over-Relaxation

There are many systems that meet the requirements for convergence with the Gauss-Seidel method, but for which convergence is still relatively slow. A slightly altered version of the Gauss-Seidel method, called *Successive Over-Relaxation* (SOR), can result in faster convergence. This is achieved by introducing a *relaxation factor* $\omega \geq 1$ and modifying (15.3) as

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

Simplifying the equation, we have

$$x_i^{(k+1)} = x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \mathbf{a}_i^\mathsf{T}\mathbf{x}^{(k)}\right). \tag{15.5}$$

Note that when $\omega = 1$, SOR reduces to Gauss-Seidel. The relaxation factor ω weights the new iteration between the current best approximation and the next approximation in a way that can sometimes dramatically improve convergence.

[2]See the lab on Linear Systems for a review of `scipy.sparse` matrices and syntax.

Problem 5. Write a function that accepts a sparse matrix A, a vector $\mathbf{b}$, a relaxation factor ω, a convergence tolerance `tol`, and a maximum number of iterations `maxiter`. Implement SOR using (15.5), compute the approximate solution to the equation $A\mathbf{x} = \mathbf{b}$. Use the usual stopping criterion. Return the approximate solution $\mathbf{x}$ as well as a boolean indicating whether the function converged and the number of iterations computed.
(Hint: this requires changing only one line of code from the sparse Gauss-Seidel function.)

A 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.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{15.6}$$

Laplace's equation can be used to model heat flow. Consider a square metal plate where the top and bottom borders 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. 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 (15.6) analytically. However, the problem can also be solved numerically using a *finite difference method*. To begin, we impose a discrete, square grid on the plate with uniform spacing. Denote the points on the grid by (x_i, y_j) and the value of u at these points (the heat) as $u(x_i, y_j) = U_{i,j}$. Using the centered difference quotient for second derivatives to approximate the partial derivatives,

$$\begin{aligned} 0 &= \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \\ &\approx \frac{U_{i+1,j} - 2U_{i,j} + U_{i-1,j}}{h^2} + \frac{U_{i,j+1} - 2U_{i,j} + U_{i,j-1}}{h^2} \\ &= \frac{1}{h^2}\left(-4U_{i,j} + U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1}\right), \end{aligned} \tag{15.7}$$

where $h = x_{i+1} - x_i = y_{j+1} - y_j$ is the distance between the grid points in either direction. 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 (where $u(x, y)$ is unknown) 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 nth row.

$$\mathbf{u} = \begin{bmatrix} U_{1,1} & U_{1,2} & \cdots & U_{1,n} & U_{2,1} & U_{2,2} & \cdots & U_{2,n} & \cdots & U_{n,n} \end{bmatrix}^\mathsf{T}$$

From (15.7), for an interior point $U_{i,j}$, we have

$$-4U_{i,j} + U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1} = 0. \tag{15.8}$$

If any of the neighbors to $U_{i,j}$ is a boundary point on the grid, its value is already determined by the boundary conditions. For example, the neighbor $U_{3,0}$ of the gridpoint for $U_{3,1}$ is fixed at $U_{3,0} = 100$. In this case, (15.8) becomes

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

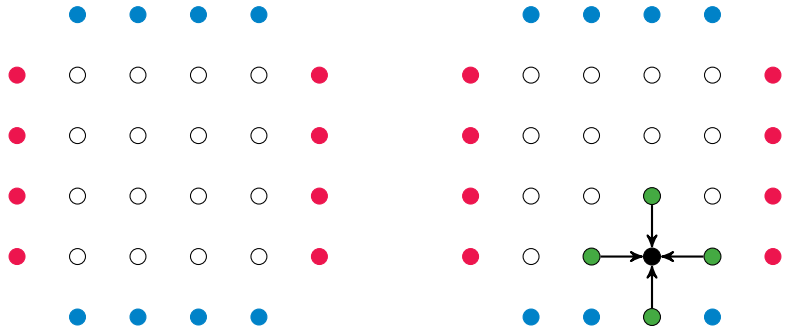


Figure 15.1: On the left, an example of a 6×6 grid ($n = 4$) where the red dots are hot boundary zones and the blue dots are cold boundary zones. On the right, the green dots are the neighbors of the interior black dot that are used to approximate the heat at the black dot.

The constants on the right side of (15.8) become the $n^2 \times 1$ vector $\mathbf{b}$. All nonzero entries of $\mathbf{b}$ correspond to interior points that touch the left or right boundaries.

As an example, writing (15.8) for the 16 interior points of the grid in Figure 15.1 results in the following 16×16 system $A\mathbf{u} = \mathbf{b}$. Note the block structure (empty blocks are all zeros).

$$\left[\begin{array}{cccc|cccc|cccc|cccc}
-4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & & & & & & & & \\
1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & & & & & & & & \\
0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & & & & & & & & \\
0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & & & & & & & & \\
\hline
1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & & & & \\
0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & & & & \\
0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & & & & \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & & & & \\
\hline
 & & & & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
 & & & & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 \\
 & & & & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 \\
 & & & & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 \\
\hline
 & & & & & & & & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 \\
 & & & & & & & & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 \\
 & & & & & & & & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 \\
 & & & & & & & & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4
\end{array}\right]
\left[\begin{array}{c} U_{1,1} \\ U_{1,2} \\ U_{1,3} \\ U_{1,4} \\ U_{2,1} \\ U_{2,2} \\ U_{2,3} \\ U_{2,4} \\ U_{3,1} \\ U_{3,2} \\ U_{3,3} \\ U_{3,4} \\ U_{4,1} \\ U_{4,2} \\ U_{4,3} \\ U_{4,4} \end{array}\right]
=
\left[\begin{array}{c} -100 \\ 0 \\ 0 \\ -100 \\ -100 \\ 0 \\ 0 \\ -100 \\ -100 \\ 0 \\ 0 \\ -100 \\ -100 \\ 0 \\ 0 \\ -100 \end{array}\right]$$

More concisely, for any positive integer n, the matrix A can be written as

$$A = \begin{bmatrix} B & I & & & \\ I & B & I & & \\ & I & \ddots & \ddots & \\ & & \ddots & \ddots & I \\ & & & I & B \end{bmatrix}, \quad \text{where } B = \begin{bmatrix} -4 & 1 & & & \\ 1 & -4 & 1 & & \\ & 1 & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & -4 \end{bmatrix} \text{ is } n \times n.$$

Problem 6. Write a function that accepts an integer n, a relaxation factor ω, a convergence tolerance `tol` that defaults to 10^{-8}, a maximum number of iterations `maxiter` that defaults to 100, and a bool `plot` that defaults to `False`. Generate and solve the corresponding system $A\mathbf{u} = \mathbf{b}$ using Problem 5. Also return a boolean indicating whether the function converged and the number of iterations computed.
(Hint: see Problem 5 of the Linear Systems lab for the construction of A. Also, `np.tile()` may be useful for constructing $\mathbf{b}$.)

If `plot=True`, visualize the solution $\mathbf{u}$ 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. Note that the $\mathbf{u}$ must be reshaped as an $n \times n$ array to properly visualize the result.

Problem 7. To demonstrate how convergence is affected by the value of the relaxation factor ω in SOR, run your function from Problem 6 with $\omega = 1,\ 1.05,\ 1.1,\ \ldots, 1.9, 1.95$ and $n = 20$. Plot the number of computed iterations as a function of ω. Return the value of ω that results in the least number of iterations.

Note that the matrix A from Problem 6 is not strictly diagonally dominant. However, A is positive definite, so the algorithm will converge. Unfortunately, convergence for these kinds of systems usually requires more iterations than for strictly diagonally dominant systems. Therefore, set `tol=1e-2` and `maxiter=1000`.

Recall that $\omega = 1$ corresponds to the Gauss-Seidel method. Choosing a more optimal relaxation factor saves a large number of iterations. This could translate to saving days or weeks of computation time while solving extremely large linear systems on a supercomputer.