Part II

Sparse Linear Algebra

Chapter 5

Sparse Matrices

5.1 Introduction to Sparse Matrices

Sparse matrices are fundamental structures in computational physics, characterized by having most elements equal to zero. These matrices naturally arise in many physical systems where interactions are local or limited, such as in quantum mechanics, finite element methods, and network analysis.

5.1.1 Definition and Properties

A matrix $A \in \mathbb{R}^{m \times n}$ is considered sparse if the number of non-zero elements is $O(\max(m, n))$. The sparsity of a matrix can be quantified by:

$$\text{sparsity} = \frac{\text{number of zero elements}}{\text{total number of elements}} = 1 - \frac{n_{nz}}{mn}, \tag{5.1.1}$$

where n_{nz} is the number of non-zero elements.

Key properties that make sparse matrices computationally advantageous include:

- Memory efficiency: Only non-zero elements need storage
- Computational efficiency: Operations can skip zero elements
- Structure preservation: Many operations preserve sparsity

5.2 Storage Formats for Sparse Matrices

Efficient storage and manipulation of sparse matrices require specialized formats. We present three fundamental storage schemes, each with distinct advantages for different computational tasks. Consider the following example sparse matrix for illustration:

$$A = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 3 & 4 & 0 \\ 5 & 0 & 0 & 6 \\ 0 & 7 & 8 & 0 \end{pmatrix} \tag{5.2.1}$$

5.2.1 Compressed Sparse Row (CSR)

The CSR format organizes data in three arrays:

- `values`: Non-zero elements in row-major order
- `col_indices`: Column indices of non-zero elements
- `row_ptr`: Starting positions of each row in `values`

For matrix A, the CSR representation is:

values	1	2	3	4	5	6	7	8
col_indices	0	3	1	2	0	3	1	2
row_ptr	0	2	4	6	8			

5.2.2 Compressed Sparse Column (CSC)

The CSC format is the column-oriented analog of CSR, using:

- `values`: Non-zero elements in column-major order
- `row_indices`: Row indices of non-zero elements
- `col_ptr`: Starting positions of each column

For matrix A, the CSC representation is:

values	1	5	3	7	4	8	2	6
row_indices	0	2	1	3	1	3	0	2
col_ptr	0	2	4	6	8			

5.2.3 Coordinate List (COO)

The COO format stores explicit coordinates for each non-zero element:

- `row`: Row indices of non-zero elements
- `col`: Column indices of non-zero elements
- `data`: Values of non-zero elements

For matrix A, the COO representation is:

row	0	0	1	1	2	2	3	3
col	0	3	1	2	0	3	1	2
data	1	2	3	4	5	6	7	8

5.2.4 Format Comparison and Selection

Each storage format has its advantages and specific use cases:

- **CSR** excels in:
  - Row-wise operations and matrix-vector multiplication
  - Integration with standard linear algebra libraries
  - Most common format in sparse linear algebra software
- **CSC** is preferred for:
  - Column-wise operations
  - Direct solver algorithms
  - Column-oriented matrix updates
- **COO** is optimal for:
  - Matrix construction and modification
  - Incremental matrix building
  - Simple and intuitive format for initial data input
  - Intermediate format before conversion to CSR/CSC

In practice, CSR is often the default choice due to its balance of simplicity and efficiency, particularly in scientific computing applications where matrix-vector multiplication is a dominant operation.

5.3 Operations on Sparse Matrices

5.3.1 Matrix-Vector Multiplication

Matrix-vector multiplication is a fundamental operation in many numerical algorithms. For a sparse matrix A and a vector x, the product $y = Ax$ can be computed efficiently.

For CSR format:

$$y_i = \sum_{j=\texttt{row\_ptr}[i]}^{\texttt{row\_ptr}[i+1]-1} \texttt{values}[j] \cdot x_{\texttt{col\_indices}[j]} \tag{5.3.1}$$

5.3.2 Implementation and Performance Comparison

Let's implement matrix-vector multiplication for different formats and compare their performance:

```python
import numpy as np
import scipy.sparse as sp
import time

def create_sparse_matrix(n, density=0.01):
    return sp.random(n, n, density=density, format='csr')

def benchmark_mv_mult(n, density=0.01, num_runs=100):
    A_csr = create_sparse_matrix(n, density)
    A_csc = A_csr.tocsc()
    A_coo = A_csr.tocoo()
    A_dense = A_csr.toarray()
    x = np.random.rand(n)

    times = {}

    for format_name, A in [('CSR', A_csr), ('CSC', A_csc), ('COO', A_coo), ('Dense
', A_dense)]:
        start = time.time()
        for _ in range(num_runs):
            y = A @ x
        end = time.time()
        times[format_name] = (end - start) / num_runs

    return times

n = 10000
results = benchmark_mv_mult(n)

for format_name, avg_time in results.items():
    print(f"{format_name}: {avg_time:.6f} seconds")
```

This code benchmarks matrix-vector multiplication for different sparse matrix formats and compares them with dense matrix multiplication.

5.4 Applications in Computational Physics

5.4.1 Finite Difference Method

Sparse matrices are particularly useful in finite difference methods for solving partial differential equations. Consider the 1D heat equation:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} \tag{5.4.1}$$

Here, $u(x,t)$ is a function of position x and time t, representing the state of the system at each point in space and time. The precise physical interpretation of $u(x,t)$ depends on the specific application of the equation:

- In thermal systems, $u(x,t)$ typically represents temperature. In this case, the equation models how heat spreads through a one-dimensional medium over time.
- In diffusion processes, $u(x,t)$ might represent the concentration of a substance. The equation then describes how the substance diffuses through space over time.
- In probability theory, $u(x,t)$ could represent a probability density function. The equation would then model the evolution of probability distributions in certain stochastic processes.

The parameter α is called the diffusion coefficient. Its physical interpretation also depends on the context:

- In heat conduction, $\alpha = k/(\rho c)$, where k is thermal conductivity, ρ is density, and c is specific heat capacity.
- In particle diffusion, α is related to the mobility of the diffusing particles.
- In probability theory, α is related to the variance of the underlying random process.

The heat equation is a parabolic partial differential equation that describes how the quantity u "smooths out" over time without creating or destroying the total amount of u. This conservation property is fundamental to many physical processes and is reflected in the mathematical structure of the equation.

The term $\frac{\partial u}{\partial t}$ represents the rate of change of u with respect to time at a fixed position. The term $\frac{\partial^2 u}{\partial x^2}$ represents the curvature of u with respect to position at a fixed time. The equation essentially states that the rate of change of u at any point is proportional to the curvature of u at that point.

In our numerical solution, we will discretize this continuous equation, approximating the derivatives with finite differences and representing the system state at discrete points in space and time. This discretization allows us to leverage sparse matrix techniques for efficient computation.

Discretizing this equation using central differences in space and forward difference in time yields:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \alpha \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{(\Delta x)^2} \tag{5.4.2}$$

This can be represented as a matrix equation:

$$\mathbf{u}^{n+1} = \left(I + \frac{\alpha \Delta t}{(\Delta x)^2} A \right) \mathbf{u}^n \tag{5.4.3}$$

where A is a tridiagonal sparse matrix:

$$A = \begin{bmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \cdots & 0 \\ 0 & 1 & -2 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 1 \\ 0 & 0 & \cdots & 1 & -2 \end{bmatrix} \tag{5.4.4}$$

5.4.2 Implementation of 1D Heat Equation Solver

Let's implement a solver for the 1D heat equation using sparse matrices. We discretize the domain in both space and time:

- Spatial discretization: $x_i = i\Delta x$, where $i = 0, 1, ..., nx - 1$ and $\Delta x = L/(nx - 1)$
- Temporal discretization: $t_n = n\Delta t$, where $n = 0, 1, ..., nt$ and $\Delta t = T/nt$

Here, L is the length of the domain, T is the total simulation time, nx is the number of spatial points, and nt is the number of time steps.

5.4.2.1 Initial Condition

For this example, we'll use a sinusoidal initial condition:

$$u(x, 0) = \sin(\pi x/L) \tag{5.4.5}$$

This initial condition represents a temperature distribution that varies smoothly from 0 at the boundaries to a maximum of 1 at the center of the domain. It's a common choice for testing heat equation solvers because:

- It satisfies the boundary conditions $u(0, t) = u(L, t) = 0$ for all t

- It's an eigenfunction of the Laplacian operator, which means the solution will decay exponentially in time while maintaining its shape
- It's smooth and easy to implement numerically

5.4.2.2 Boundary Conditions

We'll use Dirichlet boundary conditions, keeping the temperature at the ends of the domain fixed at zero:

$$u(0,t) = u(L,t) = 0 \quad \forall t \tag{5.4.6}$$

These boundary conditions are implicitly enforced by our choice of discretization and initial condition.

Now, let's implement this solver:

```python
import numpy as np
import scipy.sparse as sp
import matplotlib.pyplot as plt

def solve_heat_equation_1d(L, nx, nt, alpha, T):
    dx = L / (nx - 1)
    dt = T / nt

    # Create sparse matrix
    diagonals = [-2*np.ones(nx), np.ones(nx-1), np.ones(nx-1)]
    offsets = [0, -1, 1]
    A = sp.diags(diagonals, offsets, shape=(nx, nx), format='csr')

    # Initial condition
    u = np.sin(np.pi * np.linspace(0, L, nx) / L)

    # Time stepping
    for _ in range(nt):
        u = u + alpha * dt / (dx**2) * (A @ u)

    return u

# Parameters
L = 1.0  # Length of domain
nx = 100  # Number of spatial points
nt = 1000  # Number of time steps
alpha = 0.01  # Diffusion coefficient
T = 0.5  # Total simulation time

u_final = solve_heat_equation_1d(L, nx, nt, alpha, T)

# Plotting
```

```python
x = np.linspace(0, L, nx)
plt.figure(figsize=(10, 6))
plt.plot(x, u_final)
plt.title('1D Heat Equation Solution')
plt.xlabel('x')
plt.ylabel('Temperature')
plt.grid(True)
plt.savefig('heat_equation_solution.pdf')
plt.close()
```

This code solves the 1D heat equation using a sparse matrix representation of the finite difference operator.

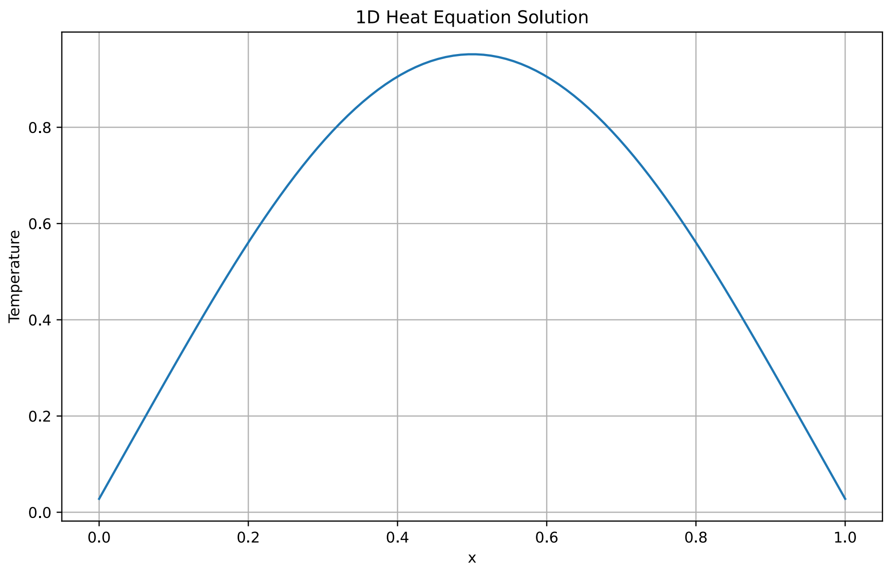

Figure 5.4.1: Solution of the 1D Heat Equation

Chapter 6

Sparse Linear Systems

6.1 Conjugate Gradient Method

In many applications, one must solve the linear system

$$A\,|x\rangle = |b\rangle, \tag{6.1.1}$$

where A is real, symmetric, and positive-definite on an n-dimensional space, and $|b\rangle$ is given. Due to the positive-definiteness of A, the quadratic functional

$$f(|x\rangle) \;=\; \tfrac{1}{2}\,\langle x|A|x\rangle \;-\; \langle x|b\rangle \tag{6.1.2}$$

is strictly convex, with a unique minimizer that solves (6.1.1).

6.1.1 Motivation: Expansion in A-Conjugate Directions

Suppose there exists a basis $\{|p_0\rangle, \ldots, |p_{n-1}\rangle\}$ such that

$$\langle p_i|\,A\,|p_j\rangle \;=\; 0 \quad \text{for} \quad i \neq j, \tag{6.1.3}$$

i.e. the vectors are mutually A-conjugate. Then one solves $A\,|x\rangle = |b\rangle$ by writing

$$|x\rangle \;=\; \sum_{i=0}^{n-1} \omega_i\,|p_i\rangle,$$

and substituting into $\langle p_k|A|x\rangle = \langle p_k|b\rangle$, which gives

$$\omega_k \;=\; \frac{\langle p_k|b\rangle}{\langle p_k|A|p_k\rangle}.$$

Because such directions are not known in advance, the Conjugate Gradient (CG) method constructs them iteratively.

6.1.2 Derivation of the Iterative Updates

Initial Step. Start from an initial guess $|x_0\rangle$. Define the residual

$$|r_0\rangle \;=\; |b\rangle - A\,|x_0\rangle. \tag{6.1.4}$$

Since $-\,|r_0\rangle$ is the negative gradient of f at $|x_0\rangle$, choose

$$|p_0\rangle \;=\; |r_0\rangle. \tag{6.1.5}$$

Optimal Step Length α_k. At iteration k, consider

$$|x(\alpha)\rangle \;=\; |x_k\rangle + \alpha\,|p_k\rangle.$$

Minimizing $f(|x(\alpha)\rangle)$ with respect to α gives

$$\alpha_k \;=\; \frac{\langle p_k|r_k\rangle}{\langle p_k|A|p_k\rangle}. \tag{6.1.6}$$

Then we update

$$|x_{k+1}\rangle = |x_k\rangle + \alpha_k\,|p_k\rangle, \quad |r_{k+1}\rangle = |r_k\rangle - \alpha_k\,A\,|p_k\rangle. \tag{6.1.7}$$

By construction, it can also be shown that the next residual is perpendicular to the current search direction $\langle p_k|r_{k+1}\rangle = 0$.

Moreover, one shows by induction that

$$\langle p_k|r_k\rangle \;=\; \langle r_k|r_k\rangle \quad \text{for all } k. \tag{6.1.8}$$

Hence α_k can also be expressed in its final form often used in computations as

$$\alpha_k \;=\; \frac{\langle r_k|r_k\rangle}{\langle p_k|A|p_k\rangle}.$$

Proof that $\alpha_k = \frac{\langle p_k | r_k \rangle}{\langle p_k | A | p_k \rangle}$

Statement: For the line search

$$|x(\alpha)\rangle = |x_k\rangle + \alpha\, |p_k\rangle,$$

minimizing

$$f(|x\rangle) = \tfrac{1}{2}\, \langle x|A|x\rangle - \langle x|b\rangle$$

with respect to α gives

$$\alpha_k = \frac{\langle p_k|r_k\rangle}{\langle p_k|A|p_k\rangle},$$

where $|r_k\rangle = |b\rangle - A\, |x_k\rangle$.
Proof:

1. Substitute $|x(\alpha)\rangle$ into f:

$$f(\alpha) = \tfrac{1}{2}\, \langle x_k + \alpha p_k|A|x_k + \alpha p_k\rangle - \langle x_k + \alpha p_k|b\rangle.$$

2. Differentiate $f(\alpha)$ and set to zero:

$$\tfrac{d}{d\alpha} f(\alpha) = \langle p_k|A|x_k\rangle + \alpha\, \langle p_k|A|p_k\rangle - \langle p_k|b\rangle = 0.$$

3. Recognize that $\langle p_k|b\rangle - \langle p_k|A|x_k\rangle = \langle p_k|r_k\rangle$. Hence

$$\alpha = \frac{\langle p_k|r_k\rangle}{\langle p_k|A|p_k\rangle}.$$

Proof that $\langle p_k|r_{k+1}\rangle = 0$

Statement: Given

$$\alpha_k \;=\; \frac{\langle p_k|r_k\rangle}{\langle p_k|A|p_k\rangle} \quad \text{and} \quad |r_{k+1}\rangle \;=\; |r_k\rangle \;-\; \alpha_k\, A\, |p_k\rangle,$$

then $\langle p_k|r_{k+1}\rangle = 0$.
Proof:

$$\langle p_k|r_{k+1}\rangle \;=\; \langle p_k|r_k\rangle \;-\; \alpha_k\, \langle p_k|A|p_k\rangle \;=\; \langle p_k|r_k\rangle \;-\; \frac{\langle p_k|r_k\rangle}{\langle p_k|A|p_k\rangle}\, \langle p_k|A|p_k\rangle \;=\; 0.$$

Proof of $\langle p_k|r_k\rangle = \langle r_k|r_k\rangle$

Base Case: For $k = 0$, $|p_0\rangle = |r_0\rangle$, so $\langle p_0|r_0\rangle = \langle r_0|r_0\rangle$.
Inductive Step: Suppose $\langle p_k|r_k\rangle = \langle r_k|r_k\rangle$. Define

$$|p_{k+1}\rangle = |r_{k+1}\rangle + \beta_k\,|p_k\rangle,$$

with $\langle p_k|r_{k+1}\rangle = 0$. Then

$$\langle p_{k+1}|r_{k+1}\rangle = \langle r_{k+1}|r_{k+1}\rangle + \beta_k\,\langle p_k|r_{k+1}\rangle = \langle r_{k+1}|r_{k+1}\rangle.$$

Hence the property holds for $k + 1$.

Determination of β_k. To maintain A-conjugacy,

$$\langle p_k|A|p_{k+1}\rangle = 0.$$

With

$$|p_{k+1}\rangle = |r_{k+1}\rangle + \beta_k\,|p_k\rangle,$$

we obtain

$$\beta_k \;=\; -\,\frac{\langle p_k|A|r_{k+1}\rangle}{\langle p_k|A|p_k\rangle}.$$

Using $|r_{k+1}\rangle = |r_k\rangle - \alpha_k\,A\,|p_k\rangle$ and the symmetry of A one shows

$$\langle p_k|A|r_{k+1}\rangle = -\,\frac{\langle r_{k+1}|r_{k+1}\rangle}{\alpha_k},$$

leading to

$$\beta_k = \frac{\langle r_{k+1}|r_{k+1}\rangle}{\alpha_k\,\langle p_k|A|p_k\rangle} = \frac{\langle r_{k+1}|r_{k+1}\rangle}{\langle r_k|r_k\rangle}.$$

6.1.3 Convergence Properties

In exact arithmetic, CG converges in at most n iterations. Let

$$|e_k\rangle = |x^*\rangle - |x_k\rangle, \quad \|\,|e_k\rangle\,\|_A = \sqrt{\langle e_k|A|e_k\rangle}.$$

Then

$$\|\,|e_k\rangle\,\|_A \;\le\; 2\left(\tfrac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^k \|\,|e_0\rangle\,\|_A, \tag{6.1.9}$$

where $\kappa(A)$ is the condition number of A. Thus, if A is well-conditioned, CG converges rapidly.

Algorithm 6.1 Conjugate Gradient Method

```
1: Input: SPD operator $A$, vector $|b\rangle$, initial guess $|x_0\rangle$, tolerance $\epsilon$.
2: $|r_0\rangle \leftarrow |b\rangle - A\,|x_0\rangle$
3: $|p_0\rangle \leftarrow |r_0\rangle$
4: $k \leftarrow 0$
5: while $\| \,|r_k\rangle\, \| > \epsilon$ do
6:     $\alpha_k \leftarrow \dfrac{\langle r_k|r_k\rangle}{\langle p_k|A|p_k\rangle}$
7:     $|x_{k+1}\rangle \leftarrow |x_k\rangle + \alpha_k\,|p_k\rangle$
8:     $|r_{k+1}\rangle \leftarrow |r_k\rangle - \alpha_k\,A\,|p_k\rangle$
9:     $\beta_k \leftarrow \dfrac{\langle r_{k+1}|r_{k+1}\rangle}{\langle r_k|r_k\rangle}$
10:    $|p_{k+1}\rangle \leftarrow |r_{k+1}\rangle + \beta_k\,|p_k\rangle$
11:    $k \leftarrow k+1$
12: end while
13: Output: Approximate solution $|x_k\rangle$
```

6.1.4 Complete CG Algorithm

6.1.5 Implementation in Python

Here's a Python implementation of the Conjugate Gradient method:

```python
import numpy as np

def conjugate_gradient(A, b, x0=None, tol=1e-10, max_iter=1000):
    n = len(b)
    if x0 is None:
        x = np.zeros(n)
    else:
        x = x0.copy()

    r = b - A @ x
    p = r.copy()

    for i in range(max_iter):
        Ap = A @ p
        alpha = np.dot(r, r) / np.dot(p, Ap)
        x += alpha * p
        r_new = r - alpha * Ap

        if np.linalg.norm(r_new) < tol:
            return x, i+1

        beta = np.dot(r_new, r_new) / np.dot(r, r)
        p = r_new + beta * p
        r = r_new

    raise ValueError(f"CG did not converge within {max_iter} iterations")

# Example usage
A = np.array([[4, 1], [1, 3]])
b = np.array([1, 2])
x, iterations = conjugate_gradient(A, b)
print(f"Solution: {x}")
print(f"Iterations: {iterations}")
```

6.2 Extensions for Non–Positive Definite or Non–Symmetric Matrices

The derivation of the conjugate gradient method in the previous sections assumes that the coefficient matrix A is symmetric and positive–definite. These properties are crucial because they ensure that the quadratic functional

$$f(|x\rangle) = \frac{1}{2}\langle x|A|x\rangle - \langle x|b\rangle \tag{6.2.1}$$

is strictly convex, which guarantees a unique minimizer that coincides with the solution of

$$A\,|x\rangle = |b\rangle. \tag{6.2.2}$$

When A is not positive definite—or if it is non–symmetric—the functional (6.2.1) may lose its convexity, and the standard CG method cannot be applied directly. In these cases, one may consider the following approaches:

1. CG on the Normal Equations

If A is not positive definite or is non–symmetric but has full column rank, one strategy is to form the normal equations:

$$A^T A\,|x\rangle = A^T\,|b\rangle. \tag{6.2.3}$$

The matrix $A^T A$ is symmetric and, provided A has full column rank, positive–definite. The CG method can then be applied to (6.2.3) with the guarantee of convergence. However, note that:

- The condition number of $A^T A$ is the square of that of A, potentially leading to slower convergence.
- Forming $A^T A$ explicitly is generally discouraged because of increased computational cost and the possibility of exacerbating numerical errors.

2. Modified Methods for Symmetric Indefinite Matrices

If A is symmetric but indefinite (i.e., it has both positive and negative eigenvalues), the quadratic form in (6.2.1) is no longer convex. In such cases, the standard CG method may not converge. Two popular alternatives are:

- **MINRES (Minimum Residual Method):** This algorithm is designed for symmetric (possibly indefinite) systems. It minimizes the residual over the Krylov subspace at each iteration while maintaining the symmetry of the underlying operator.
- **SYMMLQ:** Similar in spirit to MINRES, SYMMLQ is another iterative method for symmetric indefinite systems and is often preferred when the residual norm is not a reliable indicator of convergence.

6.3 Application: The Laplace Equation

As an application of the CG method, let's consider the numerical solution of the Laplace equation in two dimensions:

$$\nabla^2\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 0 \tag{6.3.1}$$

6.3.1 Discretization

We'll use a finite difference method to discretize the equation on a square grid. Let $\phi_{i,j}$ represent the value of ϕ at the grid point (i, j). The discrete approximation of the Laplace equation is:

$$\frac{\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} - 4\phi_{i,j}}{h^2} = 0 \tag{6.3.2}$$

where h is the grid spacing. This discretization leads to a system of linear equations $A\mathbf{x} = \mathbf{b}$, where A is a sparse matrix with a specific structure. For a grid of size $N \times N$, A is an $N^2 \times N^2$ matrix with five non-zero diagonals.

6.3.2 Python Implementation

Here's a Python implementation to solve the Laplace equation using the CG method:

```python
import numpy as np
import matplotlib.pyplot as plt

def laplace_matrix(N):
    n = N*N
    diag = 4*np.ones(n)
    off_diag = -np.ones(n-1)
    off_diag[N-1::N] = 0  # Adjust for boundary

    A = np.diag(diag) + np.diag(off_diag, k=1) + np.diag(off_diag, k=-1)
    A += np.diag(-np.ones(n-N), k=N) + np.diag(-np.ones(n-N), k=-N)
    return A

def solve_laplace(N, boundary_condition):
    A = laplace_matrix(N)
    b = np.zeros(N*N)

    # Apply boundary condition
    b[-N:] = boundary_condition

    x, _ = conjugate_gradient(A, b)
    return x.reshape((N, N))

# Solve and plot
N = 50
boundary_condition = 1.0
phi = solve_laplace(N, boundary_condition)

plt.imshow(phi, cmap='hot', interpolation='nearest')
plt.colorbar(label='Potential')
plt.title('Solution of Laplace Equation')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

6.4 Conclusion

The Conjugate Gradient method is a powerful tool for solving large, sparse linear systems, particularly those arising from the discretization of partial differential equations. Its efficiency and relatively simple implementation make it a popular choice in scientific computing and engineering applications. However, it's important to note that the method's performance can degrade for ill-conditioned matrices, and preconditioning techniques are often employed to improve convergence in such cases.

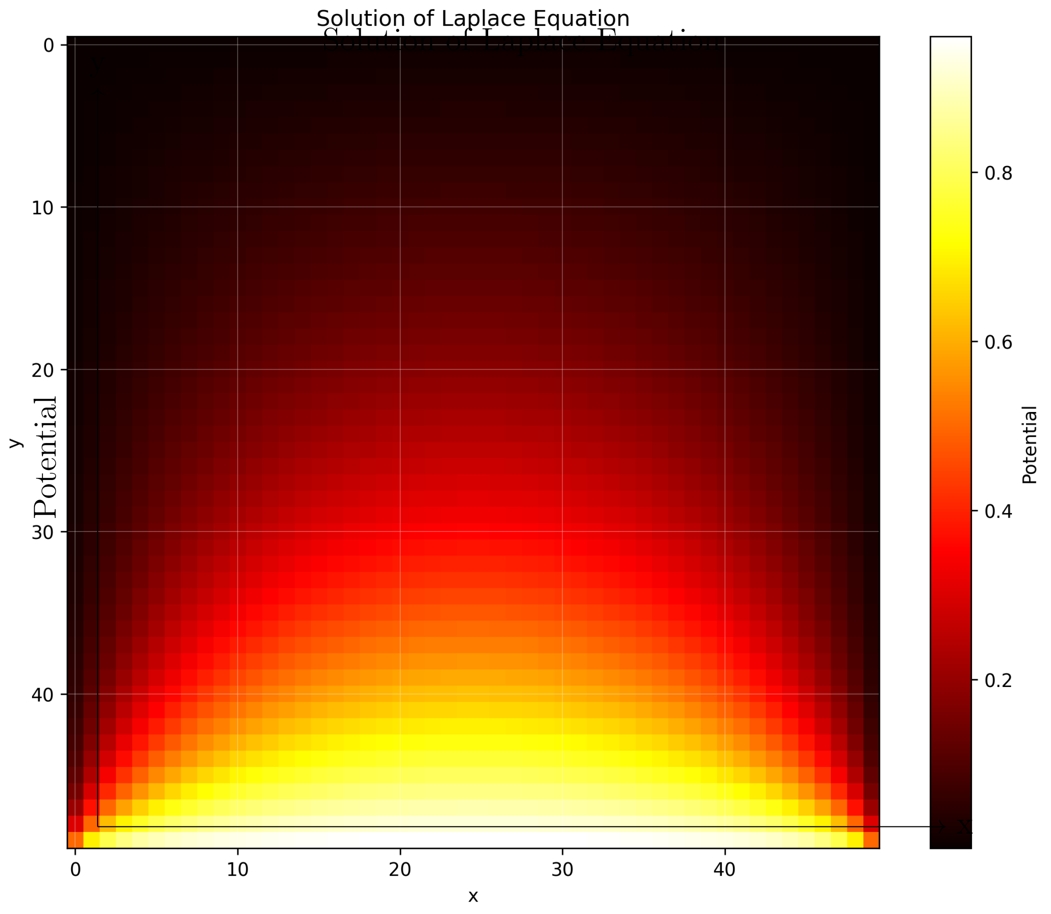

Figure 6.3.1: Visualization of the solution to the Laplace equation using the Conjugate Gradient method. The color represents the potential value at each point in the 2D grid.

Chapter 7

Sparse Eigenvalue Problem

Eigenvalue problems are central in many areas of physics and engineering, particularly when systems are linear and described by matrices. Sparse eigenvalue problems, where the matrix contains predominantly zero elements, are especially important for representing large-scale systems efficiently.

7.1 The Power Method

Consider a linear operator (or matrix) A on an n-dimensional space with eigenvalues

$$\lambda_1,\ \lambda_2,\ \ldots,\ \lambda_n,$$

ordered so that

$$|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|.$$

Let the corresponding (orthonormal) eigenvectors be

$$|v_1\rangle,\ |v_2\rangle,\ \ldots,\ |v_n\rangle.$$

The *Power Method* seeks the eigenvector associated with λ_1 by iterating

$$|\psi_{k+1}\rangle \;=\; \frac{A\,|\psi_k\rangle}{\|A\,|\psi_k\rangle\|}, \tag{7.1.1}$$

starting from some $|\psi_0\rangle$.

Assume the initial vector has a nonzero component along $|v_1\rangle$:

$$|\psi_0\rangle \;=\; \sum_{i=1}^{n} c_i\,|v_i\rangle, \quad c_1 \;\neq\; 0. \tag{7.1.2}$$

If $c_1 = 0$, then $|\psi_0\rangle$ is orthogonal to $|v_1\rangle$ and the method will not converge to $|v_1\rangle$. Otherwise, define

$$|\psi_k\rangle \;=\; \frac{A^k\,|\psi_0\rangle}{\|\,A^k|\psi_0\rangle\,\|}.$$

Below, we show two things:

1. $|\psi_k\rangle$ converges to $|v_1\rangle$.
2. The distance from $|v_1\rangle$ decreases at a rate governed by $\left|\lambda_2/\lambda_1\right|$.

Convergence to $|v_1\rangle$

After k multiplications by A,

$$A^k|\psi_0\rangle \;=\; \sum_{i=1}^{n} c_i\,\lambda_i^k\,|v_i\rangle \;=\; \lambda_1^k\left[c_1\,|v_1\rangle \;+\; \sum_{i=2}^{n} c_i\Big(\tfrac{\lambda_i}{\lambda_1}\Big)^k|v_i\rangle\right]. \tag{7.1.3}$$

Because $\left|\lambda_i/\lambda_1\right| < 1$ for $i \geq 2$, the terms in the sum become negligible for large k. Thus,

$$A^k|\psi_0\rangle \;\approx\; \lambda_1^k\,c_1\,|v_1\rangle, \quad \text{when } k \text{ is large.}$$

Normalizing,

$$|\psi_k\rangle = \frac{A^k|\psi_0\rangle}{\|A^k|\psi_0\rangle\|} \quad\longrightarrow\quad |v_1\rangle \quad \text{as } k \to \infty. \tag{7.1.4}$$

Rate of Convergence

Define the error vector

$$|e_k\rangle \;=\; |\psi_k\rangle \;-\; |v_1\rangle.$$

We wish to show that

$$\|\,|e_k\rangle\| \;=\; O\!\left(\left|\tfrac{\lambda_2}{\lambda_1}\right|^k\right).$$

From (7.1.3) and normalization, one finds

$$|\psi_k\rangle \;=\; \frac{c_1\,|v_1\rangle \;+\; \sum_{i=2}^{n} c_i\big(\frac{\lambda_i}{\lambda_1}\big)^k|v_i\rangle}{\sqrt{|c_1|^2 \;+\; \sum_{i=2}^{n}|c_i|^2\big|\frac{\lambda_i}{\lambda_1}\big|^{2k}}}. \tag{7.1.5}$$

Subtract $|v_1\rangle$ to get

$$|e_k\rangle \;=\; |\psi_k\rangle - |v_1\rangle.$$

We can group terms to show that each component in $|e_k\rangle$ contains a factor of at most $\left|\lambda_2/\lambda_1\right|^k$. A direct bounding argument (by considering the numerator and how the denominator behaves) yields

$$\| \, |e_k\rangle \| = O\Big(\Big|\frac{\lambda_2}{\lambda_1}\Big|^k\Big). \tag{7.1.6}$$

Hence, each iteration reduces the error roughly by a factor $\left|\lambda_2/\lambda_1\right|$. A larger gap between λ_1 and λ_2 gives faster convergence.

Algorithm 7.1 Power Method

```
1: Choose an initial ket $|v_0\rangle$ with $\| \, |v_0\rangle \, \| = 1$.
2: for $k = 1, 2, 3, \ldots$ until convergence do
3:     $|w_k\rangle = A \, |v_{k-1}\rangle$
4:     $|v_k\rangle = \dfrac{|w_k\rangle}{\| \, |w_k\rangle \, \|}$
5:     $\lambda_k = \langle v_k|A|v_k\rangle$
6: end for
```

7.1.1 Python Implementation

```python
import numpy as np

def power_method(A, tol=1e-8, max_iter=1000):
    """
    Finds the dominant eigenvalue and eigenvector of matrix A using the Power Method.
    Parameters:
    -----------
    A : ndarray
        Square numpy array or matrix.
    tol : float
        Tolerance for convergence.
    max_iter : int
        Maximum number of iterations.

    Returns:
    --------
    lambda_new : float
        Approximate dominant eigenvalue.
    v : ndarray
        Approximate corresponding eigenvector.
    """
    n = A.shape[0]
    v = np.random.rand(n)
    v = v / np.linalg.norm(v)
    lambda_old = 0.0

    for _ in range(max_iter):
        w = A @ v
        v = w / np.linalg.norm(w)
        lambda_new = v.T @ A @ v
        if abs(lambda_new - lambda_old) < tol:
            break
        lambda_old = lambda_new

    return lambda_new, v
```

Listing 7.1: Python Implementation of the Power Method

7.2 The Inverse Power Method

The inverse power method is an iterative algorithm used to find the smallest eigenvalue and its corresponding eigenket. It is especially useful in quantum mechanics for finding the ground state of a system described by a Hamiltonian.

7.2.1 Theory

For an operator A acting on an n-dimensional space, the inverse power method seeks the eigenvalue λ with the smallest magnitude and its corresponding eigenket $|v\rangle$. By choosing a shift μ close to the desired eigenvalue, consider the operator

$$(A - \mu I)^{-1}. \tag{7.2.1}$$

Its dominant eigenvalue is

$$\frac{1}{\lambda_1 - \mu}, \tag{7.2.2}$$

where λ_1 is the eigenvalue of A closest to μ.

7.2.2 Algorithm

The inverse power method algorithm is as follows:

7.3 Convergence Rate of the Inverse Power Method

For the standard Power Method, the error after k iterations is proportional to

$$\left|\frac{\lambda_2}{\lambda_1}\right|^k, \tag{7.3.1}$$

where λ_1 and λ_2 are the dominant and subdominant eigenvalues, respectively.

Algorithm 7.2 Inverse Power Method

```
 1: Choose an initial ket $|v_0\rangle$ such that $\| |v_0\rangle \| = 1$.
 2: Choose a shift $\mu$ close to the expected smallest eigenvalue (often $\mu = 0$).
 3: for $k = 1, 2, \ldots$ until convergence do
 4:     Solve $(A - \mu I)|w_k\rangle = |v_{k-1}\rangle$ for $|w_k\rangle$.
 5:     $|v_k\rangle = \frac{|w_k\rangle}{\| |w_k\rangle \|}$
 6:     $\lambda_k = \langle v_k|A|v_k\rangle$                         ▷ Rayleigh quotient
 7:     if $|\lambda_k - \lambda_{k-1}| <$ tolerance then
 8:         break
 9:     end if
10: end for
11: return $\lambda_k$, $|v_k\rangle$
```

7.3.1 Convergence Analysis for the Inverse Power Method

In the inverse power method, we apply the Power Method to the operator

$$(A - \mu I)^{-1}. \tag{7.3.2}$$

If λ is an eigenvalue of A, then $\lambda - \mu$ is an eigenvalue of $(A - \mu I)$ and the eigenvalues of $(A - \mu I)^{-1}$ are

$$\mu_i = \frac{1}{\lambda_i - \mu}, \quad i = 1, 2, \ldots, n. \tag{7.3.3}$$

Thus, the convergence rate for the inverse power method is given by

$$\left|\frac{\mu_2}{\mu_1}\right|^k = \left|\frac{\lambda_1 - \mu}{\lambda_2 - \mu}\right|^k, \tag{7.3.4}$$

where λ_1 is the eigenvalue closest to μ (typically the smallest eigenvalue) and λ_2 is the next closest.

7.3.2 Analysis of the Convergence Rate

Key observations include:

1. When seeking the smallest eigenvalue, the shift μ is chosen close to, but slightly less than, the smallest eigenvalue λ_1.

2. As μ approaches λ_1, the quantity $|\lambda_1 - \mu|$ becomes very small, causing

$$\left|\frac{\lambda_1 - \mu}{\lambda_2 - \mu}\right|$$

to be much smaller than $\left|\frac{\lambda_2}{\lambda_1}\right|$ from the standard Power Method.

3. This results in a faster convergence rate when using the inverse power method.

7.4 Application to Quantum Mechanics: The Harmonic Oscillator

In quantum mechanics, we often need to find the ground state of a system, which corresponds to the smallest eigenvalue of the Hamiltonian. The inverse power method is well-suited for this task, especially when dealing with sparse Hamiltonians.

7.4.1 The Harmonic Oscillator Hamiltonian

Consider the one-dimensional harmonic oscillator, described by the Hamiltonian:

$$H = -\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + \frac{1}{2}m\omega^2 x^2 \tag{7.4.1}$$

where m is the mass of the particle, ω is the angular frequency of the oscillator, and $\hbar$ is the reduced Planck constant. For simplicity, we often use natural units where $\hbar = m = \omega = 1$, which gives us:

$$H = -\frac{1}{2}\frac{d^2}{dx^2} + \frac{1}{2}x^2 \tag{7.4.2}$$

7.4.2 Discretization of the Hamiltonian

To apply numerical methods like the inverse power method, we need to discretize this continuous Hamiltonian. We do this by approximating the system on a grid of points.

We define a spatial grid with N points, spanning a region $[-L/2, L/2]$:

$$x_i = -\frac{L}{2} + i\Delta x, \quad i = 0, 1, ..., N-1 \tag{7.4.3}$$

where $\Delta x = \frac{L}{N-1}$ is the grid spacing.

The kinetic energy term involves the second derivative operator. We can approximate this using the finite difference method:

$$-\frac{1}{2}\frac{d^2\psi}{dx^2} \approx -\frac{1}{2\Delta x^2}(\psi_{i+1} + \psi_{i-1} - 2\psi_i) \tag{7.4.4}$$

This approximation has an error of order $O(\Delta x^2)$. The potential energy term is simply a function of position, so it can be directly evaluated at each grid point:

$$V(x_i) = \frac{1}{2}x_i^2 \tag{7.4.5}$$

Combining these discretizations, we can write the Hamiltonian as a matrix:

$$H = \begin{pmatrix} d & -t & 0 & 0 & 0 & \cdots & 0 \\ -t & d & -t & 0 & 0 & \cdots & 0 \\ 0 & -t & d & -t & 0 & \cdots & 0 \\ 0 & 0 & -t & d & -t & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & d \end{pmatrix} \quad \text{where} \quad \begin{aligned} d &= \tfrac{1}{\Delta x^2} + \tfrac{1}{2}x_i^2 \\ t &= \tfrac{1}{2\Delta x^2} \end{aligned}$$

Figure 7.4.1: Structure of the discretized Hamiltonian matrix for the harmonic oscillator

$$H_{ij} = -\frac{1}{2\Delta x^2}(\delta_{i,j+1} + \delta_{i,j-1} - 2\delta_{i,j}) + \frac{1}{2}x_i^2\delta_{i,j} \tag{7.4.6}$$

where $\delta_{i,j}$ is the Kronecker delta.

Figure 7.4.1 shows the structure of the discretized Hamiltonian matrix. It is a tridiagonal matrix, where:

- The main diagonal contains the sum of the kinetic energy term $\frac{1}{\Delta x^2}$ and the potential energy term $\frac{1}{2}x_i^2$.
- The off-diagonals contain the constant value $-\frac{1}{2\Delta x^2}$ from the kinetic energy term.

This sparse structure makes the Hamiltonian particularly suitable for efficient numerical computations.

7.4.3 Boundary Conditions

The choice of boundary conditions affects the structure of the Hamiltonian at the edges of the grid. Common choices include:

- Dirichlet boundary conditions: $\psi(x_0) = \psi(x_{N-1}) = 0$
- Periodic boundary conditions: $\psi(x_0) = \psi(x_{N-1})$

For the harmonic oscillator, we typically use Dirichlet boundary conditions, assuming the wavefunction vanishes at the boundaries of our computational domain. This is a good approximation if we choose L to be sufficiently large compared to the characteristic length of the oscillator.

7.4.4 Implementation Considerations

When implementing this discretized Hamiltonian:

- Use sparse matrix representations to take advantage of the tridiagonal structure.
- Choose N and L carefully: N should be large enough for good resolution, and L should be large enough to contain the relevant part of the wavefunction.
- Be aware of discretization errors, which decrease as Δx decreases (as N increases for fixed L).

7.5 Python Implementation for the Harmonic Oscillator

Let's implement the inverse power method for finding the ground state of the harmonic oscillator. For simplicity, we'll use SciPy's sparse matrix functionality to solve the linear systems.

```python
import numpy as np
import scipy.sparse as sp
import scipy.sparse.linalg as spla
import matplotlib.pyplot as plt

def create_hamiltonian(N, L):
    """Create the sparse Hamiltonian for a harmonic oscillator."""
    dx = L / (N - 1)
    x = np.linspace(-L/2, L/2, N)

    # Kinetic energy term
    T = sp.diags([-1, 2, -1], [-1, 0, 1], shape=(N, N)) * (0.5 / dx**2)

    # Potential energy term
    V = sp.diags(0.5 * x**2)

    return T + V

def inverse_power_method(A, tol=1e-10, max_iter=1000):
    """Implement the inverse power method."""
    n = A.shape[0]
    v = np.random.rand(n)
    v /= np.linalg.norm(v)

    energies = []

    for _ in range(max_iter):
        w = spla.spsolve(A, v)
        v = w / np.linalg.norm(w)
        energy = v.T @ (A @ v)
        energies.append(energy)

        if len(energies) > 1 and abs(energies[-1] - energies[-2]) < tol:
            break

    return energy, v, energies

# Set up the problem
N = 1000  # Number of grid points
L = 10.0  # Length of the well

# Create Hamiltonian
H = create_hamiltonian(N, L)

# Find the ground state
E0, psi0, energy_convergence = inverse_power_method(H)

print(f"Computed ground state energy: {E0:.6f}")
print(f"Analytical ground state energy: {0.5:.6f}")
print(f"Relative error: {abs(E0 - 0.5) / 0.5:.6%}")

# Plot the ground state wavefunction
x = np.linspace(-L/2, L/2, N)
plt.figure(figsize=(10, 6))
plt.plot(x, psi0)
plt.title("Ground State Wavefunction of Harmonic Oscillator")
plt.xlabel("Position")
plt.ylabel("Amplitude")
plt.grid(True)
plt.savefig('ground_state_wavefunction.png', dpi=300, bbox_inches='tight')
plt.close()

# Plot the energy convergence
plt.figure(figsize=(10, 6))
plt.plot(energy_convergence)
plt.title("Convergence of Ground State Energy")
plt.xlabel("Iteration")
plt.ylabel("Energy")
plt.grid(True)
plt.savefig('energy_convergence.png', dpi=300, bbox_inches='tight')
plt.close()

# Print the final few energy values
print("\nFinal few energy values:")
for i, e in enumerate(energy_convergence[-5:], start=len(energy_convergence)-4):
    print(f"Iteration {i}: {e:.8f}")
```

Figure 7.5.1 (left) shows the computed ground state wavefunction. It has a Gaussian shape centered at x = 0, which is characteristic of the harmonic oscillator ground state. Figure 7.5.1 (right) illustrates the convergence of the ground state energy during the inverse power method iterations. As you can see, the algorithm converges quickly to the exact ground state energy of $E_0 = \frac{1}{2}$.

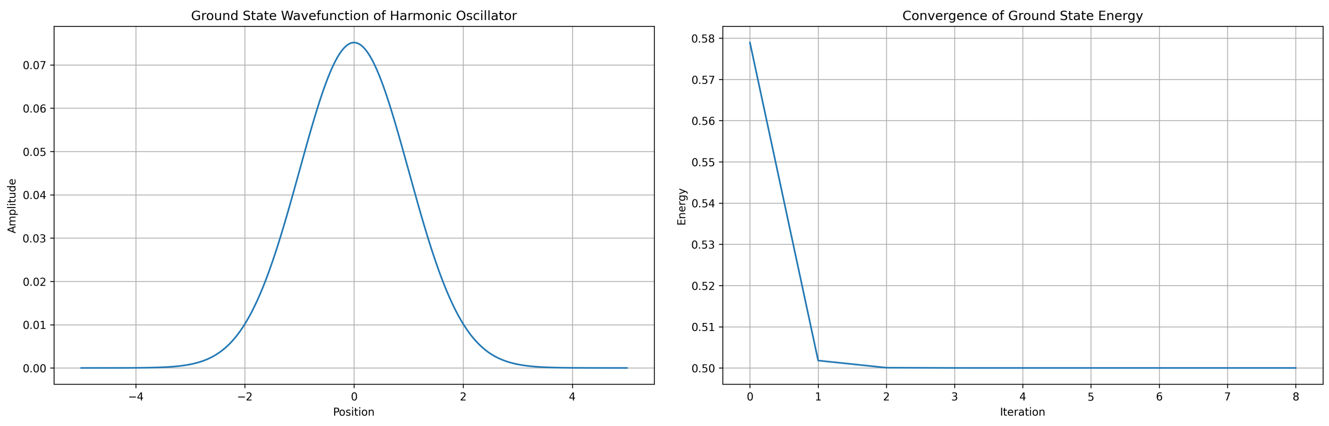

Figure 7.5.1: Ground State Wavefunction of the Harmonic Oscillator (left) and Convergence of the Ground State Energy (right)