Mathematical Models and Numerical Methods for Big Data

Comprehensive Course Notes

Academic Year 2024/2025

Table of Contents

Introduction to Data Analysis Methods

Motivation and Basic Concepts

This course explores advanced mathematical and computational methods for analyzing large-scale data. We focus on matrix and tensor decompositions, graph-based methods, dimension reduction techniques, and numerical algorithms for large-scale problems that form the foundation of modern data science applications.

The key challenges in big data analysis include:

- Efficient handling of high-dimensional data
- Extracting meaningful patterns from massive datasets
- Developing scalable algorithms for large-scale problems
- Dimensionality reduction while preserving essential information
- Finding appropriate mathematical representations for complex data

Big data typically has characteristics often referred to as the "4 Vs":

- **Volume**: Extremely large amounts of data
- **Velocity**: Data that is generated or must be processed quickly
- **Variety**: Heterogeneous data from multiple sources
- **Veracity**: Uncertainty or inconsistency in data

Mathematical Foundations

Throughout this course, we rely on concepts from linear algebra, numerical analysis, graph theory, and optimization. Essential mathematical tools include:

- **Vector spaces and linear transformations**: The foundational concept for representing data and transformations
- **Matrix decompositions (particularly SVD)**: For uncovering latent structure in data matrices
- **Graph theory and spectral methods**: For analyzing relational data and networks
- **Numerical algorithms for large-scale eigenproblems**: Essential for processing big data efficiently

- **Tensor representations and decompositions**: For analyzing multi-way data
- **Optimization techniques**: For fitting models to data

Overview of Applications

The methods covered in this course are applied in various domains:

- **Recommender systems and collaborative filtering**: Suggesting products, movies, or content to users based on their preferences and behavior
- **Web search and page ranking**: Determining the importance of web pages in search results
- **Image and signal processing**: Denoising, compression, and feature extraction
- **Network analysis and community detection**: Finding structures in social, biological, or information networks
- **Dimensionality reduction and data visualization**: Converting high-dimensional data to lower dimensions for analysis and visualization
- **Clustering and classification**: Grouping similar data points or assigning data to predefined categories

---

Data Analysis with the SVD

Singular Value Decomposition (SVD)

The Singular Value Decomposition (SVD) is a fundamental matrix factorization technique that provides insights into the structure of a matrix and forms the basis for many data analysis methods.

Definition and Properties

Definition (Singular Value Decomposition): For any matrix $A \in \mathbb{R}^{m \times n}$ with $m \geq n$, there exists a decomposition

$$A = U\Sigma V^T$$

where:

- $U \in \mathbb{R}^{m \times m}$ is an orthogonal matrix whose columns are the left singular vectors of A
- $\Sigma \in \mathbb{R}^{m \times n}$ is a diagonal matrix containing the singular values $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$
- $V \in \mathbb{R}^{n \times n}$ is an orthogonal matrix whose columns are the right singular vectors of A

For computational efficiency, we often use the "thin" or "economy" SVD:

$$A = U_n \Sigma_n V^T$$

where $U_n \in \mathbb{R}^{m\times n}$ contains only the first n columns of U and $\Sigma_n \in \mathbb{R}^{n\times n}$ is a square diagonal matrix.

The SVD provides important relationships:

$$Av_i = \sigma_i u_i$$

$$A^T u_i = \sigma_i v_i$$

Theorem: The columns of U form an orthonormal basis for the column space of A, while the columns of V form an orthonormal basis for the row space of A.

Proposition: The SVD provides direct connections to the fundamental subspaces of A:

- Left singular vectors $u_1, \ldots, u_r$ form a basis for the range of A: $\mathcal{R}(A)$
- Right singular vectors $v_1, \ldots, v_r$ form a basis for the range of A^T: $\mathcal{R}(A^T)$
- Right singular vectors $v_{r+1}, \ldots, v_n$ form a basis for the null space of A: $\mathcal{N}(A)$
- Left singular vectors $u_{r+1}, \ldots, u_m$ form a basis for the null space of A^T: $\mathcal{N}(A^T)$

where $r = \mathrm{rank}(A)$ is the number of non-zero singular values.

Theorem (Matrix Norms via SVD): For matrix A with SVD $A = U\Sigma V^T$:

$$|A|_2 = \sigma_1$$

$$|A|_F = \sqrt{\sum_{i=1}^{n} \sigma_i^2}$$

Computational Considerations:
Computing the full SVD requires $O(mn^2)$ operations when $m \geq n$, which can be prohibitive for large matrices. Randomized algorithms and iterative methods can be used to compute approximate SVDs more efficiently.

Example: Consider the matrix $A = [4 \quad 0\, 3 \quad -5]$. We can compute its SVD as follows:

First, we compute $A^T A = [25 \quad -15\, -15 \quad 25]$. The eigenvalues are $\lambda_1 = 40$ and $\lambda_2 = 10$, so the singular values are $\sigma_1 = \sqrt{40} = 2\sqrt{10}$ and $\sigma_2 = \sqrt{10}$.

The right singular vectors are the eigenvectors of $A^T A$, which are $v_1 = \frac{1}{\sqrt{2}}[1, -1]^T$ and $v_2 = \frac{1}{\sqrt{2}}[1, 1]^T$.

The left singular vectors are computed from $u_i = \frac{1}{\sigma_i} A v_i$, giving $u_1 = \frac{1}{\sqrt{10}}[2\sqrt{2}, \sqrt{2}]^T$ and $u_2 = \frac{1}{\sqrt{10}}[0, -\sqrt{10}]^T$.

Therefore, the SVD of A is:

$$A = U\Sigma V^T = \left[\frac{2\sqrt{2}}{\sqrt{10}} \quad 0\, \frac{\sqrt{2}}{\sqrt{10}} \quad -1\right] \left[2\sqrt{10} \quad 0\, 0 \quad \sqrt{10}\right] \left[\frac{1}{\sqrt{2}} \quad \frac{1}{\sqrt{2}} - \frac{1}{\sqrt{2}} \quad \frac{1}{\sqrt{2}}\right]$$

Optimal Low-Rank Matrix Approximation

One of the most important applications of the SVD is finding optimal low-rank approximations to matrices.

Theorem (Eckart-Young-Mirsky): Let $A \in \mathbb{R}^{m\times n}$ have rank $r > k \geq 1$. The rank-k matrix A_k that minimizes $|A - A_k|_F$ is given by

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T = U_k \Sigma_k V_k^T$$

where $U_k = [u_1, \ldots, u_k]$, $V_k = [v_1, \ldots, v_k]$, and $\Sigma_k = \mathrm{diag}(\sigma_1, \ldots, \sigma_k)$.

The approximation error is

$$|A - A_k|_F = \sqrt{\sum_{i=k+1}^{r} \sigma_i^2}$$

This optimality also holds for the spectral norm:

$$|A - A_k|_2 = \sigma_{k+1}$$

Proof Sketch: For any rank-k matrix B, the dimension of its null space $\mathcal{N}(B)$ is at least $n - k$. Therefore, there exists a unit vector $w \in \mathcal{N}(B) \cap \mathrm{span} v_1, \ldots, v_{k+1}$. For such a vector:

$$|A - B|_2 \geq |(A - B)w|_2 = |Aw|_2 \geq \sigma_{k+1}$$

For A_k, we have $|A - A_k|_2 = \sigma_{k+1}$, showing that A_k achieves the minimum possible error.

Remark: The truncated SVD provides the optimal rank-k approximation in terms of both the Frobenius norm and the 2-norm. This is a powerful result with applications in dimensionality reduction, denoising, and data compression.

Implementation (Python):

```python
import numpy as np
from scipy.linalg import svd

def low_rank_approximation(A, k):
    # Compute the SVD
    U, s, Vt = svd(A, full_matrices=False)

    # Truncate to rank k
    U_k = U[:, :k]
    s_k = s[:k]
    Vt_k = Vt[:k, :]

    # Reconstruct the rank-k approximation
    A_k = U_k @ np.diag(s_k) @ Vt_k
    # Compute the error
    error_f = np.sqrt(np.sum(s[k:]**2))
    error_2 = s[k]

    return A_k, error_f, error_2
```

Example: Consider the matrix $A = [1 \quad 1 \quad 1 \quad 1\,1 \quad 2 \quad 3 \quad 4\,1 \quad 3 \quad 6 \quad 10]$. Its singular values are approximately $\sigma_1 \approx 14.07$, $\sigma_2 \approx 1.30$, and $\sigma_3 \approx 0.10$. If we create a rank-2 approximation, the error in the Frobenius norm will be $|A - A_2|_F = \sigma_3 \approx 0.10$, which is quite small compared to the largest singular value. This indicates that the data in A is well-approximated by a rank-2 structure.

Principal Component Analysis (PCA)

Principal Component Analysis is a statistical method for dimensionality reduction that uses the SVD as its computational foundation.

Mathematical Formulation

Consider a data matrix $X \in \mathbb{R}^{m \times n}$ where rows represent m observations and columns represent n features. We typically center the data by subtracting the mean of each column to obtain the centered data matrix A.

Definition (PCA): The principal components of the centered data matrix A are the eigenvectors of the covariance matrix $C = \frac{1}{m-1} A^T A$.

The SVD provides a direct way to compute the principal components:

Theorem: If $A = U \Sigma V^T$ is the SVD of the centered data matrix, then:

- The columns of V are the principal components (eigenvectors of C)
- The eigenvalues of C are $\lambda_i = \frac{\sigma_i^2}{m-1}$
- The projections of the data onto the principal components are given by $AV = U\Sigma$ (these are called the principal component scores)

Example (PCA in Action): In a dataset of face images, each row of A might represent a single image (flattened to a vector), and each column represents a pixel position. PCA can extract "eigenfaces" (principal components) that capture the main directions of variation in the image set. The first eigenface might capture variations in lighting, the second might capture facial expressions, etc.

Variance Explained and Component Selection

A key aspect of PCA is selecting the number of components to retain. This is typically done by examining the proportion of variance explained:

$$\text{Proportion of variance explained by first } k \text{ components} = \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{n} \lambda_i} = \frac{\sum_{i=1}^{k} \sigma_i^2}{\sum_{i=1}^{n} \sigma_i^2}$$

Common approaches for selecting k include:

- Setting a threshold for cumulative explained variance (e.g., 90%)
- Examining the scree plot for an "elbow" point (where the explained variance drops off sharply)
- Using cross-validation to determine optimal k for a specific task

Computational Complexity: For a data matrix of size $m \times n$:

- Computing the covariance matrix: $O(mn^2)$
- Computing the eigendecomposition of the covariance matrix: $O(n^3)$
- Computing the SVD directly on the data matrix: $O(mn^2)$ when $m \geq n$

For high-dimensional data where $n \gg m$ (e.g., gene expression data), it's more efficient to compute the SVD of A rather than the eigendecomposition of $A^T A$.

Implementation (Python):

```python
import numpy as np
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Create a simple dataset
np.random.seed(42)
n_samples = 100
n_features = 2
X = np.dot(np.random.randn(n_samples, 1), np.random.randn(1, n_features))
X += np.random.randn(n_samples, n_features) * 0.3

# Apply PCA
pca = PCA()
X_pca = pca.fit_transform(X)

# Plot the data and principal components
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], alpha=0.5)
for i, (component, variance) in enumerate(zip(pca.components_,
pca.explained_variance_)):
    plt.arrow(pca.mean_[0], pca.mean_[1],
              component[0] * variance, component[1] * variance,
              head_width=0.1, head_length=0.1, color=f'C{i+2}')
plt.axis('equal')
plt.title(f'PCA: Explained variance ratio =
{pca.explained_variance_ratio_}')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)
```

PCA vs SVD

While PCA and SVD are mathematically related, they have different interpretations:

- PCA is a statistical technique that finds directions of maximum variance in the data
- SVD is a matrix factorization method that decomposes a matrix into orthogonal factors

In practice, when computing PCA:

1. Center the data matrix X to get A
2. Either:
    - Compute the eigendecomposition of $\frac{1}{m-1}A^TA$
    - Compute the SVD of A and use the right singular vectors as principal components

Latent Semantic Analysis

Latent Semantic Analysis (LSA) is an application of SVD to natural language processing for discovering hidden relationships between terms and documents.

Term-Document Matrix

In LSA, we construct a term-document matrix $A \in \mathbb{R}^{m \times n}$ where:

- Each row represents a term in the vocabulary
- Each column represents a document
- Entry A_{ij} represents the frequency of term i in document j, often weighted by term frequency-inverse document frequency (TF-IDF):

  $$A_{ij} = \text{TF}_{ij} \times \text{IDF}_i$$

  where:

  $$\text{TF}_{ij} = \frac{\text{Count of term } i \text{ in document } j}{\text{Total terms in document } j}$$

  $$\text{IDF}_i = \log \frac{\text{Total number of documents}}{\text{Number of documents containing term } i}$$

LSA via SVD

LSA applies the truncated SVD to the term-document matrix:

$$A \approx A_k = U_k \Sigma_k V_k^T$$

where $k \ll \min(m, n)$.

The resulting decomposition reveals:

- U_k: term vectors in the latent semantic space
- V_k: document vectors in the latent semantic space
- Σ_k: importance of each latent semantic dimension

Example: After performing LSA, document similarity can be measured by the cosine distance between document vectors in the latent space:

$$\text{similarity}(d_i, d_j) = \frac{(V_k\Sigma_k)_i \cdot (V_k\Sigma_k)_j}{|(V_k\Sigma_k)_i||(V_k\Sigma_k)_j|}$$

This allows for semantic search, where documents can be retrieved based on conceptual content rather than exact keyword matching.

Concrete Example: Consider a small corpus with three documents:

1. "The cat sat on the mat."
2. "The dog ran across the yard."
3. "The cat chased the dog across the yard."

The term-document matrix (after stemming and removing stop words) might look like:

```
       Doc1  Doc2  Doc3
cat     1     0     1
mat     1     0     0
dog     0     1     1
ran     0     1     0
yard    0     1     1
chase   0     0     1
```

After applying the SVD and reducing to 2 dimensions, terms and documents are mapped to the same latent semantic space, revealing that "cat" and "dog" are semantically related (both are pets), and that Doc1 and Doc2 are more similar than either is to Doc3.

Implementation Considerations:

- For large vocabularies and document collections, the term-document matrix is extremely sparse
- Specialized algorithms for sparse SVD computation should be used
- Dimensionality reduction typically retains 100-300 dimensions, depending on corpus size

Movie Recommendation with Latent Factor Models

Recommender systems use latent factor models to predict user preferences for items based on observed ratings.

Matrix Completion Problem

Given a partially observed rating matrix $R \in \mathbb{R}^{m \times n}$ where R_{ij} represents the rating of user i for item j (when known), the goal is to predict the missing entries.

Latent Factor Model

We approximate the rating matrix as a product of lower-dimensional matrices:

$$R \approx PQ^T$$

where:

- $P \in \mathbb{R}^{m \times k}$ represents user factors
- $Q \in \mathbb{R}^{n \times k}$ represents item factors
- $k \ll \min(m, n)$ is the number of latent factors

Intuitively, each user and each item is represented by a k-dimensional vector of latent factors. The predicted rating is the inner product of these vectors.

Computing the Latent Factors

When R is fully observed, we can use the truncated SVD:

$$R \approx U_k \Sigma_k V_k^T = (U_k \Sigma_k^{1/2})(\Sigma_k^{1/2} V_k^T) = PQ^T$$

When R is partially observed, we use methods like Funk SVD, which minimizes:

$$\min_{P,Q} \sum_{(i,j) \in \Omega} (R_{ij} - p_i^T q_j)^2 + \lambda(|P|_F^2 + |Q|_F^2)$$

where Ω is the set of observed entries and λ is a regularization parameter.

This is typically solved using stochastic gradient descent (SGD):

```
For each observed rating R_ij:
    Compute error: e_ij = R_ij - p_i^T q_j
    Update p_i: p_i = p_i + α(e_ij q_j - λp_i)
    Update q_j: q_j = q_j + α(e_ij p_i - λq_j)
```

where α is the learning rate.

Advanced Models:

- **Biased Matrix Factorization**: Adds user and item biases

$$R_{ij} \approx \mu + b_i + b_j + p_i^T q_j$$

- **Temporal Dynamics**: Incorporates time-varying factors

- **Factorization Machines**: Generalizes matrix factorization to include additional features

Implementation (Python):

```python
import numpy as np

def funk_svd(R, k=10, alpha=0.01, lambda_reg=0.1, iterations=50):
    """
    Implement Funk SVD (Regularized Matrix Factorization)

    Parameters:
    R: Rating matrix with missing values as NaN
    k: Number of latent factors
    alpha: Learning rate
    lambda_reg: Regularization parameter
    iterations: Number of SGD iterations

    Returns:
    P, Q: Factor matrices
    """
    m, n = R.shape
    P = np.random.normal(0, 0.1, (m, k))
    Q = np.random.normal(0, 0.1, (n, k))

    # Indices of observed ratings
    observed = np.where(~np.isnan(R))

    for _ in range(iterations):
        # Shuffle the observed ratings
        indices = np.arange(len(observed[0]))
        np.random.shuffle(indices)

        # SGD update
        for idx in indices:
            i, j = observed[0][idx], observed[1][idx]
            error = R[i, j] - np.dot(P[i], Q[j])

            # Update latent factors
            P[i] += alpha * (error * Q[j] - lambda_reg * P[i])
            Q[j] += alpha * (error * P[i] - lambda_reg * Q[j])

    return P, Q
```

Evaluation Metrics:

- **RMSE (Root Mean Squared Error)**: $\sqrt{\frac{1}{|\text{test set}|}\sum_{(i,j)\in\text{test set}}(R_{ij}-\hat{R}_{ij})^2}$
- **MAE (Mean Absolute Error)**: $\frac{1}{|\text{test set}|}\sum_{(i,j)\in\text{test set}}|R_{ij}-\hat{R}_{ij}|$

- **Precision@k**: Proportion of recommended items in the top-k that are relevant
- **Recall@k**: Proportion of relevant items found in the top-k recommendations

Practical Considerations:

- Cold start problem: How to handle new users or items with no ratings
- Scalability: For large systems with millions of users and items
- Interpretability: Understanding why certain recommendations are made

SVD and Least-Squares Problems

The SVD provides a robust method for solving least-squares problems, particularly when the system is ill-conditioned.

Least-Squares via SVD

For the overdetermined system $Ax \approx b$ where $A \in \mathbb{R}^{m\times n}$ with $m > n$, the least-squares solution minimizes $|Ax - b|_2$.

Using the SVD $A = U\Sigma V^T$, the solution is:

$$x = V\Sigma^\dagger U^T b$$

where $\Sigma^\dagger$ is the pseudoinverse of Σ, obtained by taking the reciprocal of each non-zero singular value and leaving zeros unchanged.

Theorem (Moore-Penrose Pseudoinverse): The Moore-Penrose pseudoinverse of A is:

$$A^\dagger = V\Sigma^\dagger U^T$$

For a full-rank matrix with $m > n$, this simplifies to:

$$A^\dagger = (A^T A)^{-1} A^T$$

Derivation: Starting with $Ax \approx b$, we want to minimize $|Ax - b|_2$. Using the SVD:

$$|Ax - b|_2 = |U\Sigma V^T x - b|_2 = |\Sigma V^T x - U^T b|_2$$

Let $z = V^T x$ and $c = U^T b$. Then we want to minimize $|\Sigma z - c|_2$. This has the component-wise solution:

$$z_i = \{c_i/\sigma_i \quad \text{if } \sigma_i > 0 \text{ arbitrary} \quad \text{if } \sigma_i = 0$$

Setting $z_i = 0$ when $\sigma_i = 0$ gives the minimum-norm solution:

$$z = \Sigma^\dagger c$$

Therefore, $x = Vz = V\Sigma^\dagger U^T b = A^\dagger b$.

Numerical Example: Consider the overdetermined system:

$$[1 \quad 1\,2 \quad 0\,0 \quad 3]x = [2\,1\,3]$$

Using the SVD to compute the pseudoinverse:

1. Find the SVD: $A = U\Sigma V^T$
2. Compute $\Sigma^\dagger$
3. Calculate $x = V\Sigma^\dagger U^T b$

The solution is approximately $x = [0.3, 1.0]^T$, which minimizes $|Ax - b|_2$.

Regularized Least-Squares

When A is ill-conditioned (i.e., some singular values are very small but nonzero), the solution can be highly sensitive to perturbations in b. Regularization can improve stability.

Tikhonov Regularization: Instead of minimizing $|Ax - b|_2^2$, we minimize $|Ax - b|_2^2 + \lambda|x|_2^2$ for some $\lambda > 0$.

The solution is:

$$x_\lambda = (A^T A + \lambda I)^{-1} A^T b$$

Using the SVD, this becomes:

$$x_\lambda = \sum_{i=1}^{r} \frac{\sigma_i}{\sigma_i^2 + \lambda}(u_i^T b)v_i$$

Filter Factors: The regularization can be seen as applying filter factors to the SVD components:

$$x_\lambda = \sum_{i=1}^{r} f_i \frac{u_i^T b}{\sigma_i} v_i$$

where $f_i = \frac{\sigma_i^2}{\sigma_i^2 + \lambda}$ are filter factors that reduce the contribution of small singular values.

L-Curve Method for Choosing λ: The L-curve plots the norm of the solution $|x_\lambda|_2$ against the residual norm $|Ax_\lambda - b|_2$ for different values of λ. The optimal λ is often found at the "corner" of this curve.

Implementation (Python):

```python
import numpy as np
from scipy.linalg import svd

def tikhonov_regularization(A, b, lambda_reg):
    # Compute the SVD
    U, s, Vt = svd(A, full_matrices=False)

    # Compute filter factors
    filter_factors = s**2 / (s**2 + lambda_reg)
    # Compute regularized solution
    UTb = U.T @ b
    x_lambda = Vt.T @ (filter_factors * (UTb / s))

    # Compute residual norm and solution norm
    residual_norm = np.linalg.norm(A @ x_lambda - b)
    solution_norm = np.linalg.norm(x_lambda)

    return x_lambda, residual_norm, solution_norm
```

Multiway Data Analysis

Higher-Order SVD of Tensors

Tensors are multi-dimensional arrays that generalize the concept of matrices. For complex data with multiple modes, tensor decompositions offer more flexibility than matrix methods.

Tensor Basics

Definition (Tensor): A k-way or k-mode tensor $\mathcal{A} \in \mathbb{R}^{m_1 \times m_2 \times \cdots \times m_k}$ is a multidimensional array with elements $\mathcal{A}_{i_1,i_2,...,i_k}$ for indices $i_j \in 1, \ldots, m_j$ and $j \in 1, \ldots, k$.

Definition (Tensor Operations): For a 3-way tensor $\mathcal{A} \in \mathbb{R}^{m_1 \times m_2 \times m_3}$:

- **Fibers**: One-dimensional slices, e.g., $\mathcal{A}_{:,i_2,i_3}$, $\mathcal{A}_{i_1,:,i_3}$, $\mathcal{A}_{i_1,i_2,:}$
- **Slices**: Two-dimensional slices, e.g., $\mathcal{A}_{i_1,:,:}$, $\mathcal{A}_{:,i_2,:}$, $\mathcal{A}_{:,:,i_3}$
- **Mode-n product**: $\mathcal{B} = \mathcal{A} \times_n U$ means $\mathcal{B}_{i_1,...,j,...,i_k} = \sum_{i_n} \mathcal{A}_{i_1,...,i_n,...,i_k} U_{j,i_n}$
- **Mode-n unfolding**: Reshapes the tensor into a matrix by arranging the mode-n fibers as columns

Inner Product and Norm: The inner product of two tensors $\mathcal{A}, \mathcal{B} \in \mathbb{R}^{m_1 \times m_2 \times \cdots \times m_k}$ is:

$$\langle \mathcal{A}, \mathcal{B} \rangle = \sum_{i_1=1}^{m_1} \sum_{i_2=1}^{m_2} \cdots \sum_{i_k=1}^{m_k} \mathcal{A}_{i_1,i_2,...,i_k} \mathcal{B}_{i_1,i_2,...,i_k}$$

The Frobenius norm is:

$$|\mathcal{A}|_F = \sqrt{\langle \mathcal{A}, \mathcal{A} \rangle} = \sqrt{\sum_{i_1=1}^{m_1} \sum_{i_2=1}^{m_2} \cdots \sum_{i_k=1}^{m_k} \mathcal{A}_{i_1,i_2,...,i_k}^2}$$

Rank-One Tensor: A tensor $\mathcal{A}$ is rank-one if it can be written as an outer product of vectors:

$$\mathcal{A} = a^{(1)} \circ a^{(2)} \circ \cdots \circ a^{(k)}$$

where $\circ$ denotes the outer product and $a^{(n)} \in \mathbb{R}^{m_n}$.

Higher-Order SVD (HOSVD)

Theorem (HOSVD): A 3-way tensor $\mathcal{A} \in \mathbb{R}^{m_1 \times m_2 \times m_3}$ can be decomposed as:

$$\mathcal{A} = \mathcal{S} \times_1 U^{(1)} \times_2 U^{(2)} \times_3 U^{(3)}$$

where:

- $U^{(1)} \in \mathbb{R}^{m_1 \times m_1}$, $U^{(2)} \in \mathbb{R}^{m_2 \times m_2}$, and $U^{(3)} \in \mathbb{R}^{m_3 \times m_3}$ are orthogonal matrices
- $\mathcal{S} \in \mathbb{R}^{m_1 \times m_2 \times m_3}$ is the core tensor

The HOSVD is computed by:

1. For each mode n, unfold the tensor to obtain matrix $A_{(n)}$
2. Compute the SVD of each unfolding: $A_{(n)} = U^{(n)}\Sigma^{(n)}(V^{(n)})^T$
3. Compute the core tensor: $\mathcal{S} = \mathcal{A} \times_1 (U^{(1)})^T \times_2 (U^{(2)})^T \times_3 (U^{(3)})^T$

Algorithm (HOSVD Computation):

```
Input: Tensor A ∈ ℝ^(m₁×m₂×m₃)
Output: Factors U^(1), U^(2), U^(3) and core tensor S

For n = 1 to 3:
    A_(n) = unfold A along mode n
    Compute SVD: A_(n) = U^(n)Σ^(n)(V^(n))^T
End For

S = A ×₁ (U^(1))^T ×₂ (U^(2))^T ×₃ (U^(3))^T
```

Computational Complexity:

- Computing the unfoldings: $O(m_1 m_2 m_3)$
- Computing the SVDs: $O(m_1^2 m_2 m_3 + m_2^2 m_1 m_3 + m_3^2 m_1 m_2)$
- Computing the core tensor: $O(m_1^2 m_2 m_3 + m_1 m_2^2 m_3 + m_1 m_2 m_3^2)$

Low-Rank Tensor Approximation

Similar to matrices, we can truncate the HOSVD to obtain a low-rank approximation:

$$\mathcal{A} \approx \mathcal{S}_r \times_1 U_r^{(1)} \times_2 U_r^{(2)} \times_3 U_r^{(3)}$$

where $r = (r_1, r_2, r_3)$ specifies the multilinear rank, and $U_r^{(n)}$ contains the first r_n columns of $U^{(n)}$.

Theorem: The rank-(r_1, r_2, r_3) approximation from truncated HOSVD satisfies:

$$|\mathcal{A} - \mathcal{A}_r|_F \leq \sqrt{3}|\mathcal{A} - \mathcal{A}_{opt}|_F$$

where $\mathcal{A}_{opt}$ is the optimal rank-r approximation.

Proof Sketch: The error can be bounded by considering one mode at a time and applying the Eckart-Young theorem to each unfolding.

Implementation (Python):

```python
import numpy as np
from scipy.linalg import svd

def hosvd(tensor, ranks=None):
    """
    Compute the Higher-Order SVD of a 3-way tensor

    Parameters:
    tensor: 3D numpy array
    ranks: tuple (r1, r2, r3) for truncation, defaults to full rank

    Returns:
    U_list: List of factor matrices [U1, U2, U3]
    S: Core tensor
    """
    dims = tensor.shape

    if ranks is None:
        ranks = dims

    # Compute factor matrices via SVD of unfoldings
    U_list = []
    for mode in range(3):
        # Unfold tensor along current mode
        unfolded = unfold_tensor(tensor, mode)

        # Compute SVD
        U, _, _ = svd(unfolded, full_matrices=False)

        # Truncate if necessary
        U_truncated = U[:, :ranks[mode]]
        U_list.append(U_truncated)

    # Compute core tensor
    S = tensor.copy()
    for mode in range(3):
        S = mode_n_product(S, U_list[mode].T, mode)
    return U_list, S

def unfold_tensor(tensor, mode):
    # Implementation of tensor unfolding
    # ...

def mode_n_product(tensor, matrix, mode):
    # Implementation of mode-n product
    # ...
```

Applications

Example (Hyperspectral Imaging): In hyperspectral imaging, data can be represented as a 3D tensor where dimensions correspond to spatial coordinates (x, y) and wavelength. The HOSVD can decompose this data into spatial patterns and spectral signatures, enabling efficient compression and feature extraction.

Example (Social Network Analysis): For temporal social networks, a 3D tensor can represent interactions where dimensions correspond to sender, receiver, and time. The HOSVD can identify temporal patterns and community structures.

Example (EEG Analysis): Electroencephalogram (EEG) data can be represented as a 3D tensor (channels × time × trials). The HOSVD can extract spatial, temporal, and trial-dependent patterns.

Canonical Polyadic Decomposition

The Canonical Polyadic (CP) decomposition, also known as CANDECOMP/PARAFAC, expresses a tensor as a sum of rank-one tensors.

Definition (CP Decomposition): The rank-R CP decomposition of a 3-way tensor $\mathcal{A} \in \mathbb{R}^{m_1 \times m_2 \times m_3}$ is:

$$\mathcal{A} \approx \sum_{r=1}^{R} a_r^{(1)} \circ a_r^{(2)} \circ a_r^{(3)}$$

where $\circ$ denotes the outer product, and $a_r^{(n)} \in \mathbb{R}^{m_n}$ are the factor vectors.

In matrix form, the factor matrices are:

$$A^{(n)} = [a_1^{(n)}, a_2^{(n)}, \ldots, a_R^{(n)}] \in \mathbb{R}^{m_n \times R}$$

Tensor Rank: The tensor rank is the minimum number R such that the CP decomposition holds exactly. Unlike matrices, determining the rank of a tensor is NP-hard.

Computing the CP Decomposition

The CP decomposition is typically computed using an alternating least squares (ALS) algorithm:

Algorithm (CP-ALS):

```
Input: Tensor A, target rank R
Output: Factor matrices A^(1), A^(2), A^(3)

Initialize A^(1), A^(2), A^(3) randomly
Repeat until convergence:
    For n = 1 to 3:
        V = (A^(n-1) ⊙ A^(n-2))^T(A^(n-1) ⊙ A^(n-2))
        A_(n) = unfolding of A along mode n
        A^(n) = A_(n)((A^(n-1) ⊙ A^(n-2))V^†)
        Normalize columns of A^(n) if desired
    End For
End Repeat
```

where ⊙ denotes the Khatri-Rao product (columnwise Kronecker product).

Implementation (Python):

```python
import numpy as np

def cp_als(tensor, rank, max_iter=100, tol=1e-6):
    """
    CP decomposition using alternating least squares

    Parameters:
    tensor: 3D numpy array
    rank: Target rank of the decomposition
    max_iter: Maximum number of iterations
    tol: Convergence tolerance

    Returns:
    factors: List of factor matrices [A1, A2, A3]
    """
    dims = tensor.shape

    # Initialize factor matrices
    factors = [np.random.random((dim, rank)) for dim in dims]

    # Normalize columns
    for r in range(rank):
        for n in range(3):
            norm = np.linalg.norm(factors[n][:, r])
            factors[n][:, r] /= norm
    prev_error = float('inf')

    for iteration in range(max_iter):
        for mode in range(3):
            # Compute Khatri-Rao product of all matrices except current mode
            kr_product = khatri_rao_product([factors[n] for n in range(3) if
n != mode])

            # Unfold tensor along current mode
            unfolded = unfold_tensor(tensor, mode)

            # Update factor matrix
            factors[mode] = unfolded @ kr_product @
np.linalg.pinv(kr_product.T @ kr_product)

            # Normalize columns
            for r in range(rank):
                norm = np.linalg.norm(factors[mode][:, r])
                if norm > 0:
                    factors[mode][:, r] /= norm

        # Check convergence
        reconstructed = reconstruct_from_cp(factors)
        error = np.linalg.norm(tensor - reconstructed) /
np.linalg.norm(tensor)

        if abs(prev_error - error) < tol:
            break

        prev_error = error

    return factors

def khatri_rao_product(matrices):
    # Implementation of Khatri-Rao product
    # ...

def reconstruct_from_cp(factors):
    # Reconstruction of tensor from CP factors
    # ...
```

Properties and Challenges

Uniqueness: Unlike the matrix SVD, the CP decomposition is often unique (up to scaling and permutation of the components) under mild conditions, which is a significant advantage for interpretability.

Example (Chemometrics): In chemometrics, fluorescence spectroscopy data forms a 3D tensor (excitation wavelengths × emission wavelengths × samples). The CP decomposition can separate contributions from different chemical compounds, where each rank-one component corresponds to a distinct chemical.

Challenges:

- The CP decomposition may not always exist for a given rank R
- The ALS algorithm can converge slowly or to local minima
- The optimal rank is often unknown and difficult to determine

Degeneracy: For some tensors, attempting to find the best rank-R approximation can lead to a degenerate case where components have extremely large norms that cancel each other. This phenomenon has no matrix analog.

Comparison with HOSVD:

- CP: Decomposes tensor into sum of rank-one tensors; often unique but can be harder to compute
- HOSVD: Generalizes matrix SVD; always exists but typically not optimal for fixed multilinear rank

---

Spectral Graph Theory

How to Calculate Eigenvalues/Eigenvectors and the SVD Numerically

For large-scale problems, direct methods for computing eigenvalues and singular values become impractical. Instead, we use iterative methods that are more efficient for large matrices.

Schur Decomposition

Theorem (Schur Decomposition): For any square matrix $A \in \mathbb{C}^{n\times n}$, there exists a unitary matrix $U \in \mathbb{C}^{n\times n}$ such that:

$$A = UTU^*$$

where $T \in \mathbb{C}^{n\times n}$ is upper triangular with the eigenvalues of A on the diagonal.

The Schur decomposition is the foundation for numerical eigenvalue algorithms, as it can be computed in a stable manner.

QR Algorithm

The QR algorithm is a fundamental method for computing the Schur decomposition:

Algorithm (Basic QR Algorithm):

```
Input: Matrix A
Output: Schur form T with eigenvalues on diagonal

$A_0$ = A
For k = 1, 2, ...
    Compute QR factorization: $A_{k-1} = Q_k R_k$
    Form $A_k = R_k Q_k$
End For
```

With successive iterations, A_k converges to an upper triangular matrix with eigenvalues on the diagonal.

Practical QR Algorithm: In practice, the QR algorithm incorporates:

1. **Initial reduction to Hessenberg form**: $A = QHQ^*$ where H is upper Hessenberg (zeros below the first subdiagonal)
2. **Shifts**: To accelerate convergence, work with $A_k - \mu_k I$ where μ_k is a carefully chosen shift
3. **Deflation**: Once an eigenvalue has converged, the problem is reduced in size
4. **Implicit QR steps**: Avoid explicit computation of Q and R

Computational Complexity:

- Reduction to Hessenberg form: $O(n^3)$
- Each QR iteration: $O(n^2)$
- Total complexity: $O(n^3)$

Power Method and Inverse Iteration

Algorithm (Power Method):

```
Input: Matrix A, initial vector $x_0$
Output: Dominant eigenvector and eigenvalue

For k = 1, 2, ...
    $y_k = Ax_{k-1}$
    $x_k = y_k/\|y_k\|$
    $\lambda_k = x_k^T A x_k$
End For
```

The power method converges to the eigenvector corresponding to the largest (in magnitude) eigenvalue, with convergence rate $|\lambda_2/\lambda_1|$ where λ_1, λ_2 are the largest and second-largest eigenvalues in magnitude.

Algorithm (Inverse Iteration with Shift):

```
Input: Matrix A, initial vector $x_0$, shift μ
Output: Eigenvector corresponding to eigenvalue closest to μ

For k = 1, 2, ...
    Solve $(A - \mu I)y_k = x_{k-1}$
    $x_k = y_k/\|y_k\|$
    $\lambda_k = x_k^T A x_k$
End For
```

Inverse iteration converges to the eigenvector corresponding to the eigenvalue closest to μ, with convergence rate $|\mu - \lambda_i|/|\mu - \lambda_j|$ where λ_i is the closest eigenvalue to μ and λ_j is the second closest.

Practical Considerations:

- The power method is simple but converges slowly if $|\lambda_1| \approx |\lambda_2|$
- Inverse iteration is more flexible but requires solving linear systems
- For clustered eigenvalues, more sophisticated methods like subspace iteration or the Arnoldi/Lanczos methods are needed

Arnoldi and Lanczos Algorithms

For large, sparse matrices, Krylov subspace methods like Arnoldi and Lanczos are more efficient.

Algorithm (Arnoldi Iteration):

```
Input: Matrix A, initial vector $q_1$ with $\|q_1\| = 1$
Output: Orthonormal basis $\{q_1, ..., q_m\}$ for Krylov subspace and upper
Hessenberg matrix $H_m$

For j = 1, 2, ..., m
    $v = Aq_j$
    For i = 1, 2, ..., j
        $h_{ij} = q_i^T v$
        $v = v - h_{ij} q_i$
    End For
    $h_{j+1,j} = \|v\|$
    If $h_{j+1,j}$ = 0 then break
    $q_{j+1}$ = v/$h_{j+1,j}$
End For
```

Arnoldi iteration computes an orthonormal basis for the Krylov subspace $\mathcal{K}_m(A, q_1) = \text{span}q_1, Aq_1, A^2q_1, \ldots, A^{m-1}q_1$, and reduces A to upper Hessenberg form H_m, whose eigenvalues approximate some eigenvalues of A.

Lanczos Algorithm (for Symmetric Matrices): For symmetric matrices, Arnoldi simplifies to the Lanczos algorithm, which reduces A to a tridiagonal form:

```
Input: Symmetric matrix A, initial vector $q_1$ with ||$q_1$|| = 1
Output: Orthonormal basis {$q_1$, ..., $q_m$} and tridiagonal matrix $T_m$

$\beta_0$ = 0, $q_0$ = 0
For j = 1, 2, ..., m
    v = A$q_j$ - $\beta_{j-1}q_{j-1}$
    $\alpha_j$ = $q_j^T$v
    v = v - $\alpha_j q_j$
    $\beta_j$ = ||v||
    If $\beta_j$ = 0 then break
    $q_{j+1}$ = v/$\beta_j$
End For
```

Lanczos-Golub-Kahan Bidiagonalization (for SVD): For computing the SVD, the Lanczos-Golub-Kahan (LGK) bidiagonalization is useful:

```
Input: Matrix A, initial vector x with ||x|| = 1
Output: Bidiagonal matrix B and orthonormal bases {$u_1$, ..., $u_m$}, {$v_1$, ...,
$v_n$}

$\beta_0$ = 0, $u_0$ = 0, $v_1$ = x
For j = 1, 2, ..., min(m, n)
    $u_j$ = A$v_j$ - $\beta_{j-1}u_{j-1}$
    $\alpha_j$ = ||$u_j$||
    $u_j$ = $u_j$/$\alpha_j$
    $v_{j+1}$ = $A^T u_j$ - $\alpha_j v_j$
    $\beta_j$ = ||$v_{j+1}$||
    $v_{j+1}$ = $v_{j+1}$/$\beta_j$
End For
```

This algorithm computes matrices U and V such that U^TAV is bidiagonal, from which the SVD can be computed efficiently.

Arnoldi/Lanczos in Practice:

- Reorthogonalization is often necessary to maintain numerical stability
- Implicit restarts can reduce memory requirements for large problems
- Preconditioning can improve convergence rates

Implementation and Software Libraries

MATLAB/Octave:

```matlab
% Power method
function [lambda, v] = power_method(A, tol, max_iter)
    n = size(A, 1);
    v = randn(n, 1);
    v = v / norm(v);

    for iter = 1:max_iter
        w = A * v;
        lambda = v' * A * v;
        w = w / norm(w);

        if norm(w - v) < tol
            break;
        end
        v = w;
    end
end

% Arnoldi iteration
function [Q, H] = arnoldi(A, q1, m)
    n = size(A, 1);
    Q = zeros(n, m+1);
    H = zeros(m+1, m);

    Q(:,1) = q1 / norm(q1);

    for j = 1:m
        v = A * Q(:,j);
        for i = 1:j
            H(i,j) = Q(:,i)' * v;
            v = v - H(i,j) * Q(:,i);
        end
        H(j+1,j) = norm(v);
        if H(j+1,j) < 1e-12
            break;
        end
        Q(:,j+1) = v / H(j+1,j);
    end
end
```

Python (NumPy/SciPy):

```python
import numpy as np
from scipy.sparse.linalg import eigs, svds

# For eigenvalue problems
eigenvalues, eigenvectors = eigs(A, k=5, which='LM')

# For singular value problems
U, s, Vt = svds(A, k=5)
```

Graphs and the Graph Laplacian

Graphs provide a powerful way to model relationships between entities in data. Spectral graph theory analyzes graphs through the eigendecomposition of their associated matrices.

Basic Graph Concepts

Definition (Graph): A graph $G = (V, E, A)$ consists of:

- A set of vertices $V = v_1, v_2, \ldots, v_n$
- A set of edges $E \subset V \times V$ connecting pairs of vertices
- A symmetric adjacency matrix $A \in \mathbb{R}^{n\times n}$ with entries $A_{i,j} \geq 0$ representing the weight of edge (v_i, v_j)

Definition (Degree Matrix): The degree matrix $D \in \mathbb{R}^{n\times n}$ is a diagonal matrix with entries:

$$D_{i,i} = \sum_{j=1}^{n} A_{i,j}$$

representing the sum of weights of all edges connected to vertex v_i.

Definition (Graph Types):

- **Undirected Graph**: If A is symmetric, i.e., $A_{i,j} = A_{j,i}$
- **Weighted Graph**: If $A_{i,j}$ can take values other than 0 and 1
- **Complete Graph**: If all vertices are connected, i.e., $A_{i,j} > 0$ for all $i \neq j$
- **Regular Graph**: If all vertices have the same degree
- **Bipartite Graph**: If vertices can be divided into two disjoint sets such that no edges connect vertices in the same set

Construction of Graphs from Data: Given a dataset of points in $\mathbb{R}^d$, we can construct a similarity graph by:

1. ε**-Ball Graph**: Connect points within distance ε
2. k**-Nearest Neighbor Graph**: Connect each point to its k nearest neighbors
3. **Fully Connected Graph**: Connect all points with weights based on similarity (e.g., Gaussian kernel $w_{ij} = \exp(-|x_i - x_j|^2/\sigma^2)$)

Graph Laplacian and its Properties

Definition (Graph Laplacian): The graph Laplacian matrix $L \in \mathbb{R}^{n \times n}$ is defined as:

$$L = D - A$$

Definition (Normalized Laplacian): The normalized Laplacian $L_N \in \mathbb{R}^{n \times n}$ is defined as:

$$L_N = D^{-1/2} L D^{-1/2} = I - D^{-1/2} A D^{-1/2}$$

Definition (Random Walk Laplacian): The random walk Laplacian $L_{RW} \in \mathbb{R}^{n \times n}$ is defined as:

$$L_{RW} = D^{-1} L = I - D^{-1} A$$

Theorem (Properties of the Graph Laplacian): For the Laplacian matrix L:

- L is symmetric and positive semidefinite
- The smallest eigenvalue of L is $\lambda_1 = 0$ with eigenvector $\mathbf{1}$ (the constant vector)
- The multiplicity of the eigenvalue 0 equals the number of connected components in the graph
- For any vector $x \in \mathbb{R}^n$: $x^T L x = \frac{1}{2} \sum_{i,j=1}^{n} A_{i,j} (x_i - x_j)^2$

Proof Sketch: The positive semidefiniteness follows from:

$$x^T L x = \frac{1}{2} \sum_{i,j=1}^{n} A_{i,j} (x_i - x_j)^2 \geq 0$$

The constant vector is an eigenvector with eigenvalue 0 because:

$$L\mathbf{1} = (D - A)\mathbf{1} = D\mathbf{1} - A\mathbf{1} = \mathbf{d} - \mathbf{d} = \mathbf{0}$$

where $\mathbf{d}$ is the vector of vertex degrees.

Theorem (Properties of the Normalized Laplacian): For the normalized Laplacian L_N:

- All eigenvalues lie in the interval $[0, 2]$
- The multiplicity of the eigenvalue 0 equals the number of connected components
- A bipartite graph is characterized by having an eigenvalue $\lambda_n = 2$

Example: For the path graph with n vertices, the eigenvalues of the Laplacian are:

$$\lambda_k = 2 - 2\cos\left(\frac{\pi(k-1)}{n}\right), \quad k = 1, 2, \ldots, n$$

Cheeger Inequality: The second-smallest eigenvalue λ_2 of the normalized Laplacian (also called the algebraic connectivity or Fiedler value) provides bounds on the Cheeger constant h_G, which measures how well-connected the graph is:

$$\frac{h_G^2}{2} \leq \lambda_2 \leq 2h_G$$

Graph Construction from Data

When working with high-dimensional data, constructing an appropriate graph is crucial. Common approaches include:

1. **Complete Graph with Gaussian Weights:**

   $$A_{ij} = \exp\left(-\frac{|x_i - x_j|^2}{2\sigma^2}\right)$$

   where σ is a scale parameter.
2. **k-Nearest Neighbor Graph:**

   $$A_{ij} = \{1 \quad \text{if } x_j \text{ is among the } k \text{ nearest neighbors of } x_i \; 0 \quad \text{otherwise}$$

   To ensure symmetry, either make an edge if either x_i is a neighbor of x_j or vice versa (symmetric kNN), or only if both are neighbors of each other (mutual kNN).
3. **ε-Ball Graph:**

   $$A_{ij} = \{1 \quad \text{if } |x_i - x_j| < \varepsilon \; 0 \quad \text{otherwise}$$

Implementation (Python):

```python
import numpy as np
from sklearn.neighbors import kneighbors_graph
from scipy.spatial.distance import pdist, squareform

def construct_graph(X, method='knn', param=5, weighted=True):
    """
    Construct a graph from data points

    Parameters:
    X: Data points, shape (n_samples, n_features)
    method: 'knn', 'epsilon', or 'full'
    param: k for knn, epsilon for epsilon-ball
    weighted: whether to use Gaussian weights

    Returns:
    A: Adjacency matrix
    D: Degree matrix
    L: Laplacian matrix
    """
    n_samples = X.shape[0]

    if method == 'knn':
        # k-nearest neighbors graph
        A = kneighbors_graph(X, param, mode='connectivity',
 include_self=False).toarray()
        # Make symmetric
        A = np.maximum(A, A.T)

    elif method == 'epsilon':
        # Epsilon-ball graph
        dist_matrix = squareform(pdist(X))
        A = (dist_matrix < param).astype(float)
        np.fill_diagonal(A, 0)

    elif method == 'full':
        # Fully connected graph
        dist_matrix = squareform(pdist(X))
        A = np.ones((n_samples, n_samples)) - np.eye(n_samples)
        if weighted:
            sigma = param
            A = np.exp(-dist_matrix**2 / (2 * sigma**2))
            np.fill_diagonal(A, 0)

    # Compute degree matrix
    D = np.diag(np.sum(A, axis=1))

    # Compute Laplacian
    L = D - A

    return A, D, L
```

Graph Fourier Transform

The graph Fourier transform extends the classical Fourier transform to functions defined on graphs, providing a way to analyze signals in the graph spectral domain.

Definition and Properties

Definition (Graph Fourier Transform): Let $G = (V, E, A)$ be a graph with Laplacian $L = U\Lambda U^T$, where $U = [u_1, u_2, \ldots, u_n]$ contains the eigenvectors and $\Lambda = \text{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$ contains the eigenvalues.

For a signal $x \in \mathbb{R}^n$ defined on the vertices, the graph Fourier transform is:

$$\hat{x} = U^T x$$

The inverse graph Fourier transform is:

$$x = U\hat{x}$$

Interpretation:

- The classical Fourier transform decomposes a signal into sine and cosine waves of different frequencies
- The graph Fourier transform decomposes a signal into the eigenvectors of the Laplacian, which can be interpreted as oscillation modes on the graph
- Low eigenvalues correspond to smooth variations across the graph, while high eigenvalues correspond to rapid oscillations

Example (Path Graph): For the path graph, the eigenvectors of the Laplacian resemble discrete cosine functions, making the graph Fourier transform analogous to the discrete cosine transform.

Analogies with Classical Fourier Transform:

Classical Domain	Graph Domain
Frequency	Eigenvalue
Sine/Cosine	Laplacian eigenvector
Convolution	Graph convolution
Low-pass filter	Eigenvalue function that attenuates high eigenvalues

Graph Filtering

Graph filtering is performed via pointwise multiplication in the graph Fourier domain:

$$y = Ug(\Lambda)U^T x$$

where $g(\Lambda) = \mathrm{diag}(g(\lambda_1), g(\lambda_2), \ldots, g(\lambda_n))$ is a function applied to the eigenvalues.

Common filter types:

- **Low-pass**: $g(\lambda) = e^{-\alpha\lambda}$ (heat kernel)
- **Band-pass**: $g(\lambda) = e^{-\alpha(\lambda-\mu)^2}$ (spectral graph wavelet)
- **High-pass**: $g(\lambda) = 1 - e^{-\alpha\lambda}$

For large graphs, explicitly computing the eigendecomposition becomes infeasible. Instead, we can approximate filters using polynomial expansions:

$$g(L) \approx \sum_{k=0}^{K} \alpha_k L^k$$

which requires only matrix-vector multiplications.

Chebyshev Polynomial Approximation: For a filter function $g(\lambda)$ defined on $[0, \lambda_{max}]$, we can use Chebyshev polynomials $T_k(x)$ to approximate g:

$$g(L) \approx \sum_{k=0}^{K} c_k T_k \left(\frac{2L}{\lambda_{max}} - I \right)$$

where c_k are the Chebyshev coefficients of g.

Implementation (Python):

```python
import numpy as np
from scipy.sparse.linalg import eigsh

def graph_fourier_transform(L, x):
    """
    Compute the graph Fourier transform

    Parameters:
    L: Graph Laplacian
    x: Signal on graph vertices

    Returns:
    x_hat: Fourier coefficients
    U: Eigenvectors of L
    """
    # Compute eigendecomposition of L
    eigenvalues, U = eigsh(L, k=len(x), which='SM')

    # Compute Fourier coefficients
    x_hat = U.T @ x

    return x_hat, U, eigenvalues

def graph_inverse_fourier_transform(U, x_hat):
    """
    Compute the inverse graph Fourier transform

    Parameters:
    U: Eigenvectors of L
    x_hat: Fourier coefficients

    Returns:
    x: Signal on graph vertices
    """
    return U @ x_hat

def graph_filter(L, x, filter_func):
    """
    Apply a filter to a graph signal
    Parameters:
    L: Graph Laplacian
    x: Signal on graph vertices
    filter_func: Function that takes eigenvalues and returns filter
coefficients

    Returns:
    y: Filtered signal
    """
    # Compute eigendecomposition of L
    eigenvalues, U = eigsh(L, k=len(x), which='SM')

    # Compute Fourier coefficients
    x_hat = U.T @ x

    # Apply filter in frequency domain
    y_hat = filter_func(eigenvalues) * x_hat

    # Transform back to vertex domain
    y = U @ y_hat

    return y

# Example filter functions
def low_pass_filter(alpha):
    return lambda lambda_: np.exp(-alpha * lambda_)

def high_pass_filter(alpha):
    return lambda lambda_: 1 - np.exp(-alpha * lambda_)

def band_pass_filter(alpha, mu):
    return lambda lambda_: np.exp(-alpha * (lambda_ - mu)**2)
```

Applications

Signal Denoising: Graph-based denoising applies a low-pass filter to remove high-frequency noise while preserving the signal structure:

```
# Denoise a signal on a graph
noisy_signal = original_signal + noise
denoised_signal = graph_filter(L, noisy_signal, low_pass_filter(alpha=0.1))
```

Community Detection: The eigenvectors corresponding to small non-zero eigenvalues can be used to detect communities in the graph:

```
# Get the Fiedler vector (eigenvector corresponding to λ₂)
_, U, _ = graph_fourier_transform(L, np.ones(n))
fiedler_vector = U[:, 1]
communities = fiedler_vector > 0  # Simple thresholding
```

Graph Signal Compression: By keeping only the most significant Fourier coefficients, graph signals can be efficiently compressed:

```
# Compress a signal by keeping top k Fourier coefficients
x_hat, U, _ = graph_fourier_transform(L, signal)
k = 10  # Number of coefficients to keep
indices = np.argsort(np.abs(x_hat))[-k:]
x_hat_compressed = np.zeros_like(x_hat)
x_hat_compressed[indices] = x_hat[indices]
signal_compressed = graph_inverse_fourier_transform(U, x_hat_compressed)
```

Dimensionality Reduction with Laplacian Eigenmaps

Laplacian Eigenmaps is a nonlinear dimensionality reduction technique that preserves the local structure of the data by embedding it in a low-dimensional space using the graph Laplacian.

Algorithm

Algorithm (Laplacian Eigenmaps):

```
Input: Data points $\{x_1, x_2, ..., x_n\} \in \mathbb{R}^d$, target dimension k
Output: Embedding coordinates $\{y_1, y_2, ..., y_n\} \in \mathbb{R}^k$

1. Construct a similarity graph G from the data points
2. Compute the graph Laplacian L = D - A
3. Solve the generalized eigenvalue problem Lf = λDf
4. Use the eigenvectors corresponding to the k smallest non-zero eigenvalues
as the embedding coordinates
```

Mathematical Formulation

Laplacian Eigenmaps finds the embedding coordinates y that minimize:

$$\min_{y} \sum_{i,j=1}^{n} A_{i,j} |y_i - y_j|^2 \quad \text{subject to } y^T D y = I$$

This can be reformulated as:

$$\min_y \frac{y^T L y}{y^T D y} \quad \text{subject to } y^T D\mathbf{1} = 0$$

The solution is given by the eigenvectors of the generalized eigenvalue problem $Lf = \lambda Df$ corresponding to the smallest non-zero eigenvalues.

One-Dimensional Embedding: For a 1D embedding, the solution is the Fiedler vector (eigenvector corresponding to λ_2). This minimizes:

$$\min_{y^T D\mathbf{1}=0, y^T D y=1} y^T L y$$

Why It Works: The objective function penalizes placing connected vertices far apart in the embedding space, proportional to their edge weights. This preserves the local neighborhood structure.

Implementation (Python):

```
import numpy as np
from scipy.sparse.linalg import eigsh
from sklearn.neighbors import kneighbors_graph

def laplacian_eigenmaps(X, n_components=2, n_neighbors=5):
    """
    Perform dimensionality reduction using Laplacian Eigenmaps

    Parameters:
    X: Data points, shape (n_samples, n_features)
    n_components: Dimension of the embedding
    n_neighbors: Number of neighbors for graph construction

    Returns:
    Y: Embedding coordinates, shape (n_samples, n_components)
    """
    n_samples = X.shape[0]

    # Construct the graph
    A = kneighbors_graph(X, n_neighbors, mode='connectivity',
include_self=False)
    A = 0.5 * (A + A.T)  # Make symmetric
    A = A.toarray()

    # Compute the degree matrix
    D = np.diag(np.sum(A, axis=1))

    # Compute the Laplacian matrix
    L = D - A

    # Solve the generalized eigenvalue problem
    eigenvalues, eigenvectors = eigsh(L, k=n_components+1, M=D, which='SM')
    # Sort eigenvalues and eigenvectors
    idx = np.argsort(eigenvalues)
    eigenvalues = eigenvalues[idx]
    eigenvectors = eigenvectors[:, idx]

    # Discard the eigenvector corresponding to eigenvalue 0
    Y = eigenvectors[:, 1:n_components+1]

    return Y
```

Comparison with Other Methods

Laplacian Eigenmaps vs. PCA:

- PCA: Linear method that maximizes variance; global structure preservation
- Laplacian Eigenmaps: Nonlinear method that preserves local distances; local structure preservation

Laplacian Eigenmaps vs. t-SNE:

- t-SNE: Probabilistic method that emphasizes cluster structure; better for visualization
- Laplacian Eigenmaps: Spectral method with solid mathematical foundation; faster for large datasets

Laplacian Eigenmaps vs. Diffusion Maps:

- Diffusion Maps: Based on diffusion process on the graph; captures multi-scale structures
- Laplacian Eigenmaps: Special case of Diffusion Maps with specific time parameter

Example (Swiss Roll): The Swiss roll dataset is a classic example where nonlinear dimensionality reduction is needed. Laplacian Eigenmaps can successfully "unroll" the data to reveal its intrinsic 2D structure, while linear methods like PCA fail.

Practical Considerations:

- Choice of graph construction method (kNN, ε-ball, fully connected) affects results
- Selection of eigenvectors impacts the quality of the embedding
- For large datasets, approximate eigensolvers like Lanczos can be used

---

Ranking with Eigenvectors

Vector Iteration

Vector iteration (or power method) is a simple iterative technique for finding the dominant eigenvector of a matrix, with applications in ranking algorithms like PageRank.

Power Method

Algorithm (Power Method):

```
Input: Matrix A, initial vector $x^{(0)}$ with $\|x^{(0)}\| = 1$
Output: Dominant eigenvector and eigenvalue

For k = 1, 2, ...
    $y^{(k)} = Ax^{(k-1)}$
    $\mu_k = \|y^{(k)}\|$
    $x^{(k)} = y^{(k)}/\mu_k$
End For
```

Theorem (Convergence of Power Method): Let the eigenvalues of a diagonalizable matrix $A \in \mathbb{R}^{n\times n}$ be ordered as $|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|$. Then the sequence $x^{(k)}$ generated by the power method converges to the eigenvector corresponding to λ_1 at a rate:

$$\sin\theta^{(k)} \leq C\left(\frac{|\lambda_2|}{|\lambda_1|}\right)^k$$

where $\theta^{(k)}$ is the angle between $x^{(k)}$ and the dominant eigenvector, provided the initial vector has a non-zero component in the direction of the dominant eigenvector.

Proof Sketch: For a diagonalizable matrix $A = X\Lambda X^{-1}$, we can write:

$$x^{(0)} = \sum_{i=1}^{n} \alpha_i v_i$$

where v_i are the eigenvectors. Then:

$$A^k x^{(0)} = \sum_{i=1}^{n} \alpha_i \lambda_i^k v_i = \lambda_1^k \left(\alpha_1 v_1 + \sum_{i=2}^{n} \alpha_i \left(\frac{\lambda_i}{\lambda_1}\right)^k v_i\right)$$

As k increases, the terms with $i > 1$ become negligible since $|\lambda_i/\lambda_1| < 1$, and the sequence converges to a multiple of v_1.

Example: Consider the matrix $A = [3 \quad 1\, 1 \quad 2]$. The eigenvalues are $\lambda_1 = 3.62$ and $\lambda_2 = 1.38$. Starting with $x^{(0)} = [1, 0]^T$, the power method converges to the dominant eigenvector $v_1 \approx [0.92, 0.38]^T$ at a rate determined by $|\lambda_2/\lambda_1| \approx 0.38$.

Implementation (Python):

```python
import numpy as np
```

```python
def power_method(A, tol=1e-10, max_iter=1000):
    """
    Compute the dominant eigenvector and eigenvalue using the power method

    Parameters:
    A: Input matrix
    tol: Convergence tolerance
    max_iter: Maximum number of iterations

    Returns:
    eigenvalue: Dominant eigenvalue
    eigenvector: Corresponding eigenvector
    """
    n = A.shape[0]
    x = np.random.rand(n)
    x = x / np.linalg.norm(x)

    for i in range(max_iter):
        # Power iteration
        y = A @ x

        # Compute Rayleigh quotient (eigenvalue estimate)
        lambda_est = x.T @ A @ x

        # Normalize
        y_norm = np.linalg.norm(y)
        y = y / y_norm

        # Check convergence
        if np.linalg.norm(y - x) < tol:
            return lambda_est, y

        x = y

    return lambda_est, x
```

Inverse Iteration

Inverse iteration is a variant of the power method that can find eigenvectors corresponding to eigenvalues close to a specified shift.

Algorithm (Inverse Iteration with Shift):

```
Input: Matrix A, initial vector $x^{(0)}$ with $\|x^{(0)}\| = 1$, shift μ
Output: Eigenvector corresponding to eigenvalue closest to μ

For k = 1, 2, ...
    Solve $(A - \mu I)y^{(k)} = x^{(k-1)}$
    $x^{(k)} = y^{(k)}/\|y^{(k)}\|$
End For
```

Theorem: If μ is close to an eigenvalue λ_i of A, then inverse iteration converges to the corresponding eigenvector at a rate determined by the ratio of distances to the nearest eigenvalues:

$$\left|\frac{\mu - \lambda_j}{\mu - \lambda_i}\right|^k$$

where λ_j is the second closest eigenvalue to μ.

Implementation (Python):

```python
import numpy as np
from scipy.linalg import solve

def inverse_iteration(A, mu, tol=1e-10, max_iter=1000):
    """
    Compute the eigenvector corresponding to eigenvalue closest to mu

    Parameters:
    A: Input matrix
    mu: Shift parameter
    tol: Convergence tolerance
    max_iter: Maximum number of iterations

    Returns:
    eigenvalue: Estimated eigenvalue
    eigenvector: Corresponding eigenvector
    """
    n = A.shape[0]
    x = np.random.rand(n)
    x = x / np.linalg.norm(x)

    for i in range(max_iter):
        # Inverse iteration
        y = solve(A - mu * np.eye(n), x)

        # Compute Rayleigh quotient
        lambda_est = x.T @ A @ x

        # Normalize
        y_norm = np.linalg.norm(y)
        y = y / y_norm

        # Check convergence
        if np.linalg.norm(y - x) < tol:
            return lambda_est, y
        x = y

    return lambda_est, x
```

Rayleigh Quotient Iteration

Rayleigh quotient iteration combines the power method with a shifting strategy based on the Rayleigh quotient.

Algorithm (Rayleigh Quotient Iteration):

```
Input: Matrix A, initial vector $x^{(0)}$ with $\|x^{(0)}\| = 1$
Output: Eigenvector and eigenvalue

For k = 1, 2, ...
    Compute $\mu_k = (x^{(k-1)})^T A(x^{(k-1)})$
    Solve $(A - \mu_k I)y^{(k)} = x^{(k-1)}$
    $x^{(k)} = y^{(k)}/\|y^{(k)}\|$
End For
```

Convergence Rate: For symmetric matrices, Rayleigh quotient iteration exhibits cubic convergence near the solution, making it much faster than the power method or inverse iteration.

Implementation (Python):

```python
import numpy as np
from scipy.linalg import solve

def rayleigh_quotient_iteration(A, tol=1e-10, max_iter=100):
    """
    Compute an eigenvector and eigenvalue using Rayleigh quotient iteration

    Parameters:
    A: Input matrix (symmetric)
    tol: Convergence tolerance
    max_iter: Maximum number of iterations

    Returns:
    eigenvalue: Estimated eigenvalue
    eigenvector: Corresponding eigenvector
    """
    n = A.shape[0]
    x = np.random.rand(n)
    x = x / np.linalg.norm(x)
    for i in range(max_iter):
        # Compute Rayleigh quotient
        mu = x.T @ A @ x

        # Inverse iteration with current shift
        try:
            y = solve(A - mu * np.eye(n), x)
        except np.linalg.LinAlgError:
            # If (A - μI) is singular, perturb μ slightly
            mu += 1e-10
            y = solve(A - mu * np.eye(n), x)

        # Normalize
        y_norm = np.linalg.norm(y)
        y = y / y_norm

        # Check convergence
        if np.linalg.norm(y - x) < tol:
            return mu, y

        x = y

    return mu, x
```

Perron-Frobenius Theorem

The Perron-Frobenius theorem provides important guarantees about the dominant eigenvalue and eigenvector of non-negative matrices, which is crucial for ranking algorithms.

Non-negative Matrices

Definition (Non-negative Matrix): A matrix A is non-negative if all its entries are non-negative: $A_{ij} \geq 0$ for all i, j.

Definition (Positive Matrix): A matrix A is positive if all its entries are positive: $A_{ij} > 0$ for all i, j.

Definition (Irreducible Matrix): A non-negative matrix A is irreducible if for any i, j, there exists a positive integer k such that $(A^k)_{ij} > 0$. Equivalently, the directed graph associated with A is strongly connected.

Perron-Frobenius Theorems

Theorem (Perron-Frobenius, Version 1): If $A \in \mathbb{R}^{n \times n}$ is non-negative, then:

- The spectral radius $\rho(A)$ is an eigenvalue of A
- There exists a non-negative eigenvector x such that $Ax = \rho(A)x$

Theorem (Perron-Frobenius, Version 2): If $A \in \mathbb{R}^{n\times n}$ is non-negative and irreducible, then:

- The spectral radius $\rho(A)$ is an eigenvalue of A
- $\rho(A) > 0$
- There exists a positive eigenvector x such that $Ax = \rho(A)x$
- $\rho(A)$ is a simple eigenvalue

Theorem (Perron): If $A \in \mathbb{R}^{n\times n}$ is positive, then:

- The spectral radius $\rho(A)$ is an eigenvalue of A
- $\rho(A) > 0$
- There exists a positive eigenvector x such that $Ax = \rho(A)x$
- $\rho(A)$ is a simple eigenvalue and is larger in magnitude than all other eigenvalues

Consequences for the Power Method:

- For a positive matrix, the power method always converges to the unique positive eigenvector corresponding to $\rho(A)$
- For a non-negative irreducible matrix, the power method converges to the positive eigenvector if $\rho(A)$ is the only eigenvalue with magnitude $\rho(A)$
- If there are multiple eigenvalues with magnitude $\rho(A)$, the power method may not converge

Example (Stochastic Matrix): For a stochastic matrix S (non-negative with column sums equal to 1), $\rho(S) = 1$. If S is irreducible, then there exists a unique positive vector π such that $S\pi = \pi$ and $\sum_i \pi_i = 1$. This vector π is the stationary distribution of the Markov chain represented by S.

Example (Circulant Matrix): Consider the circulant matrix $A = [0 \quad 1 \quad 0\,0 \quad 0 \quad 1\,1 \quad 0 \quad 0]$. This matrix is irreducible but has three eigenvalues of magnitude 1: $\lambda_1 = 1$, $\lambda_2 = e^{2\pi i/3}$, $\lambda_3 = e^{4\pi i/3}$. The power method applied to this matrix will not converge.

PageRank

PageRank is an algorithm developed by Larry Page and Sergey Brin that assigns importance scores to web pages based on the link structure of the web.

The PageRank Model

The PageRank model is based on a random surfer who follows links on web pages with probability μ and randomly jumps to any page with probability $1 - \mu$.

Let x_i be the importance of page i. The basic model equation is:

$$x_i = \sum_{j:A_{j,i}=1} \frac{x_j}{D_{j,j}}$$

where A is the adjacency matrix of the web graph and D is the diagonal matrix of out-degrees.

In matrix form:

$$x = Sx$$

where $S = A^T D^{-1}$ is the stochastic transition matrix.

Problems with the Basic Model:

- Pages with no incoming links get zero importance
- Pages with no outgoing links act as "sinks"
- The graph may not be strongly connected, leading to multiple eigenvectors with eigenvalue 1

To handle these issues, the model is modified to:

$$x = \frac{1-\mu}{n}\mathbf{1} + \mu S x = Gx$$

where $G = (1-\mu)\frac{1}{n}\mathbf{1}\mathbf{1}^T + \mu S$ is the Google matrix and $\mu \approx 0.85$ is a damping factor.

Theorem: The Google matrix G has the following properties:

- G is stochastic and positive
- The largest eigenvalue of G is 1, and all other eigenvalues have magnitude at most μ
- The power method applied to G converges to the PageRank vector at a rate of approximately μ^k

Proof Sketch: Since G is positive, by the Perron theorem, it has a unique dominant eigenvalue $\lambda_1 = 1$ with a positive eigenvector. For any eigenvalue $\lambda \neq 1$ of G, we can show that $|\lambda| \leq \mu$.

Algorithm (PageRank Computation):

```
Input: Adjacency matrix A, damping factor μ
Output: PageRank vector x

Initialize $x^{(0)}$ = (1/n)1
Compute D = diag(sum(A, 2))  # Out-degree matrix
Compute S = A^T D^(-1)  # Transition matrix

For k = 1, 2, ...
    $x^{(k)}$ = ((1-μ)/n)1 + μS $x^{(k-1)}$
    If $\|x^{(k)} - x^{(k-1)}\|$ < tol
```

```
        Return $x^{(k)}$
End For
```

Implementation (Python):

```python
def pagerank(A, mu=0.85, tol=1e-10, max_iter=100):
    """
    Compute the PageRank vector for a web graph

    Parameters:
    A: Adjacency matrix (A[i,j] = 1 if there is a link from j to i)
    mu: Damping factor
    tol: Convergence tolerance
    max_iter: Maximum number of iterations

    Returns:
    x: PageRank vector
    """
    n = A.shape[0]

    # Initialize PageRank vector
    x = np.ones(n) / n

    # Compute out-degree matrix
    out_degrees = np.sum(A, axis=0)

    # Handle dangling nodes (pages with no outlinks)
    dangling = np.where(out_degrees == 0)[0]
    if len(dangling) > 0:
        for j in dangling:
            A[:, j] = 1/n  # Uniform transition from dangling nodes
        out_degrees = np.sum(A, axis=0)

    # Normalize adjacency matrix by out-degrees to get transition matrix
    S = A / out_degrees

    # Power iteration
    for _ in range(max_iter):
        x_new = (1-mu)/n * np.ones(n) + mu * S @ x

        # Check convergence
        if np.linalg.norm(x_new - x, 1) < tol:
            return x_new

        x = x_new

    return x
```

Personalized PageRank

A generalization of PageRank allows for personalization by replacing the uniform teleportation distribution with a custom vector:

$$x = (1 - \mu)v + \mu S x$$

where v is a personalization vector. This allows emphasizing certain types of pages based on user preferences or query context.

Implementation (Python):

```python
def personalized_pagerank(A, v, mu=0.85, tol=1e-10, max_iter=100):
    """
    Compute the Personalized PageRank vector

    Parameters:
    A: Adjacency matrix
    v: Personalization vector (must sum to 1)
    mu: Damping factor
    tol: Convergence tolerance
    max_iter: Maximum number of iterations

    Returns:
    x: Personalized PageRank vector
    """
    n = A.shape[0]

    # Initialize PageRank vector
    x = np.ones(n) / n

    # Compute out-degree matrix
    out_degrees = np.sum(A, axis=0)

    # Handle dangling nodes
    dangling = np.where(out_degrees == 0)[0]
    if len(dangling) > 0:
        for j in dangling:
            A[:, j] = v  # Personalized transition from dangling nodes
        out_degrees = np.sum(A, axis=0)

    # Normalize adjacency matrix
    S = A / out_degrees

    # Power iteration
    for _ in range(max_iter):
        x_new = (1-mu) * v + mu * S @ x

        # Check convergence
        if np.linalg.norm(x_new - x, 1) < tol:
            return x_new

        x = x_new

    return x
```

Applications and Extensions

HITS Algorithm (Hyperlink-Induced Topic Search): An alternative to PageRank that computes both hub scores (nodes that point to many authorities) and authority scores (nodes that are pointed to by many hubs):

```
def hits(A, max_iter=100, tol=1e-10):
    """
    Compute HITS hub and authority scores

    Parameters:
    A: Adjacency matrix
    max_iter: Maximum number of iterations
    tol: Convergence tolerance

    Returns:
    hub: Hub scores
    authority: Authority scores
    """
    n = A.shape[0]

    # Initialize hub and authority scores
    hub = np.ones(n) / np.sqrt(n)
    authority = np.ones(n) / np.sqrt(n)

    for _ in range(max_iter):
        # Update authority scores
        authority_new = A.T @ hub
        norm_auth = np.linalg.norm(authority_new)
        if norm_auth > 0:
            authority_new = authority_new / norm_auth

        # Update hub scores
        hub_new = A @ authority_new
        norm_hub = np.linalg.norm(hub_new)
        if norm_hub > 0:
            hub_new = hub_new / norm_hub

        # Check convergence
        if (np.linalg.norm(authority_new - authority) < tol and
            np.linalg.norm(hub_new - hub) < tol):
```

```
            return hub_new, authority_new

        hub = hub_new
        authority = authority_new

    return hub, authority
```

Trust Rank: A variant of PageRank that combats web spam by biasing the random jump to a set of trusted pages.

SimRank: A measure of similarity between vertices based on the structural context, computed as:

$$s(a, b) = \frac{C}{|I(a)||I(b)|} \sum_{i \in I(a)} \sum_{j \in I(b)} s(i, j)$$

where $I(a)$ is the set of in-neighbors of vertex a and C is a decay factor.

Numerical Methods for Large Scale Linear Systems

Krylov Methods for Eigenvalue Problems

Krylov subspace methods are iterative techniques for solving large-scale eigenproblems, based on projections onto Krylov subspaces.

Krylov Subspaces

Definition (Krylov Subspace): For a matrix $A \in \mathbb{R}^{n \times n}$ and a vector $x \in \mathbb{R}^n$, the m-th Krylov subspace is:

$$\mathcal{K}_m(A, x) = \text{span} x, Ax, A^2x, \ldots, A^{m-1}x$$

Properties:

- $\dim(\mathcal{K}_m(A, x)) \leq m$
- $\mathcal{K}_m(A, x) \subseteq \mathcal{K}_{m+1}(A, x)$
- $A\mathcal{K}_m(A, x) \subseteq \mathcal{K}_{m+1}(A, x)$
- $\mathcal{K}_m(\alpha A + \beta I, x) = \mathcal{K}_m(A, x)$ for $\alpha \neq 0$

Theorem: If A is an $n \times n$ matrix and p is the degree of the minimal polynomial of A with respect to x, then $\dim(\mathcal{K}_m(A, x)) = \min(m, p)$ and $\mathcal{K}_p(A, x) = \mathcal{K}_{p+1}(A, x) = \ldots = \mathcal{K}_n(A, x)$.

Arnoldi Method

The Arnoldi method builds an orthonormal basis for the Krylov subspace and reduces A to upper Hessenberg form:

Algorithm (Arnoldi):

```
Input: Matrix A, initial vector $q_1$ with $\|q_1\| = 1$, dimension m
Output: Orthonormal basis $\{q_1, ..., q_m\}$ for $K_m(A, q_1)$ and upper Hessenberg
matrix $H_m$

For j = 1, 2, ..., m
    v = $Aq_j$
    For i = 1, 2, ..., j
        $h_{ij} = q_i^T v$
        v = $v - h_{ij}q_i$
    End For
    $h_{j+1,j} = \|v\|$
    If $h_{j+1,j} = 0$ then break
    $q_{j+1} = v/h_{j+1,j}$
End For
```

The Arnoldi relation can be written as:

$$AQ_m = Q_mH_m + h_{m+1,m}q_{m+1}e_m^T$$

where $Q_m = [q_1, \ldots, q_m]$ and H_m is the $m \times m$ upper Hessenberg matrix with elements h_{ij}.

Implicit Restarting: To manage storage and computational requirements, implicit restarting selectively keeps information about desired eigenvalues while discarding the rest:

```
1. Run m steps of Arnoldi to get AQₘ = QₘHₘ + h_{m+1,m}q_{m+1}e_m^T
2. Compute eigenvalues of Hₘ and identify k wanted and (m-k) unwanted
eigenvalues
3. Apply (m-k) shifted QR steps on Hₘ with shifts equal to unwanted
eigenvalues
4. Truncate to k-step Arnoldi decomposition
5. Expand back to m steps
```

Implementation (Python):

```python
import numpy as np
from scipy.linalg import hessenberg

def arnoldi(A, q1, m):
    """
    Arnoldi iteration for computing an orthonormal basis of the Krylov
```

```
subspace

    Parameters:
    A: Input matrix
    q1: Starting vector (will be normalized)
    m: Maximum dimension of Krylov subspace

    Returns:
    Q: Orthonormal basis for Krylov subspace
    H: Upper Hessenberg matrix
    """
    n = A.shape[0]
    Q = np.zeros((n, m+1))
    H = np.zeros((m+1, m))

    # Normalize the initial vector
    Q[:, 0] = q1 / np.linalg.norm(q1)

    for j in range(m):
        # Compute v = A*q_j
        v = A @ Q[:, j]

        # Orthogonalize against previous vectors
        for i in range(j+1):
            H[i, j] = Q[:, i].T @ v
            v = v - H[i, j] * Q[:, i]

        # Get next vector
        H[j+1, j] = np.linalg.norm(v)

        # Check for invariant subspace
        if abs(H[j+1, j]) < 1e-12:
            return Q[:, :j+1], H[:j+1, :j+1]

        Q[:, j+1] = v / H[j+1, j]

    return Q[:, :m], H[:m, :m]
```

Lanczos Method for Symmetric Matrices

For symmetric matrices, the Arnoldi method simplifies to the Lanczos algorithm, generating a tridiagonal matrix:

Algorithm (Lanczos):

```
Input: Symmetric matrix A, initial vector $q_1$ with $\|q_1\| = 1$, dimension m
Output: Orthonormal basis $\{q_1, ..., q_m\}$ for $K_m(A, q_1)$ and tridiagonal matrix
$T_m$
```

```
$\beta_0 = 0, q_0 = 0$
For $j = 1, 2, ..., m$
    $v = Aq_j - \beta_{j-1}q_{j-1}$
    $\alpha_j = q_j^T v$
    $v = v - \alpha_j q_j$
    $\beta_j = ||v||$
    If $\beta_j = 0$ then break
    $q_{j+1} = v/\beta_j$
End For
```

The tridiagonal matrix T_m has diagonal elements α_j and subdiagonal elements β_j.

Numerical Issues: In practice, the computed Lanczos vectors lose orthogonality due to round-off errors. This can be addressed by reorthogonalization:

```python
def lanczos_with_reorthogonalization(A, q1, m):
    """
    Lanczos algorithm with full reorthogonalization

    Parameters:
    A: Symmetric matrix
    q1: Starting vector
    m: Maximum dimension of Krylov subspace

    Returns:
    Q: Orthonormal basis for Krylov subspace
    T: Tridiagonal matrix
    """
    n = A.shape[0]
    Q = np.zeros((n, m+1))
    T = np.zeros((m+1, m+1))

    # Normalize the initial vector
    Q[:, 0] = q1 / np.linalg.norm(q1)

    beta = 0
    q_prev = np.zeros(n)

    for j in range(m):
        # Compute v = A*q_j - beta*q_{j-1}
        v = A @ Q[:, j] - beta * q_prev

        # Full reorthogonalization
        for i in range(j+1):
            coef = Q[:, i].T @ v
            v = v - coef * Q[:, i]
            if i == j:
                T[i, i] = coef  # Diagonal element

        # Reorthogonalize again for numerical stability
        for i in range(j+1):
            coef = Q[:, i].T @ v
            v = v - coef * Q[:, i]

        beta = np.linalg.norm(v)

        # Check for invariant subspace
        if beta < 1e-12:
            return Q[:, :j+1], T[:j+1, :j+1]

        if j < m:
            Q[:, j+1] = v / beta
            T[j, j+1] = beta  # Subdiagonal element
            T[j+1, j] = beta  # Superdiagonal element

        q_prev = Q[:, j]

    return Q[:, :m], T[:m, :m]
```

Eigenvalue Approximation and Convergence

The eigenvalues of the Hessenberg matrix H_m (or tridiagonal matrix T_m for symmetric A) are called Ritz values and approximate some eigenvalues of A.

Theorem (Convergence of Ritz Values): The Ritz values tend to approximate the extreme eigenvalues of A first. For symmetric matrices, the convergence rate depends on the separation of eigenvalues.

Theorem (Kaniel-Paige-Saad): For a symmetric matrix A with eigenvalues $\lambda_1 > \lambda_2 > \cdots > \lambda_n$ and corresponding orthonormal eigenvectors $u_1, u_2, \ldots, u_n$, the error in the largest Ritz value $\theta_1^{(m)}$ after m steps of the Lanczos method satisfies:

$$0 \leq \lambda_1 - \theta_1^{(m)} \leq (\lambda_1 - \lambda_2) \tan^2(\angle(q_1, u_1)) \cdot \frac{1}{T_{m-1}^2(\frac{\lambda_1 - \lambda_2}{\lambda_2 - \lambda_n})},$$

where T_{m-1} is the Chebyshev polynomial of degree $m-1$.

Practical Considerations

Choosing the Starting Vector: The choice of starting vector q_1 affects convergence. A random vector typically works well as it will have components in all eigendirections with high probability.

Stopping Criteria: A common criterion is the residual norm $|Ax - \lambda x|_2 < \text{tol}$ for approximate eigenpairs (λ, x).

Implicit Restarting: The ARPACK library implements an implicitly restarted Arnoldi method, which SciPy's `eigsh` and `eigs` functions use.

Krylov Methods for Large Systems of Equations

Krylov subspace methods provide efficient iterative solvers for large linear systems Ax = b, especially when A is sparse.

Conjugate Gradient Method

For symmetric positive definite matrices, the Conjugate Gradient (CG) method minimizes the quadratic function $f(x) = \frac{1}{2}x^T Ax - x^T b$ over successive Krylov subspaces.

Algorithm (Conjugate Gradient):

```
Input: SPD matrix A, right-hand side b, initial guess $x_0$
Output: Approximate solution x

$r_0 = b - Ax_0$
$p_0 = r_0$
For k = 0, 1, ...
    $\alpha_k = (r_k^T r_k)/(p_k^T A p_k)$
    $x_{k+1} = x_k + \alpha_k p_k$
    $r_{k+1} = r_k - \alpha_k A p_k$
    If $\|r_{k+1}\|$ < tol then break
    $\beta_k = (r_{k+1}^T r_{k+1})/(r_k^T r_k)$
    $p_{k+1} = r_{k+1} + \beta_k p_k$
End For
```

Theorem (Convergence of CG): If A is SPD with eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n > 0$, then the error in the CG method satisfies:

|x_k - x__|_A \leq 2 \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right)^k |x_0 - x__|_A,

where $|v|_A = \sqrt{v^T Av}$ is the A-norm, x_* is the exact solution, and $\kappa = \lambda_1/\lambda_n$ is the condition number of A.

Implementation (Python):

```python
import numpy as np

def conjugate_gradient(A, b, x0=None, tol=1e-10, max_iter=None):
    """
    Conjugate Gradient method for solving Ax = b

    Parameters:
    A: Symmetric positive definite matrix
    b: Right-hand side vector
    x0: Initial guess (default: zero vector)
    tol: Convergence tolerance
    max_iter: Maximum number of iterations

    Returns:
    x: Approximate solution
    """
    n = len(b)
    if x0 is None:
        x = np.zeros(n)
    else:
        x = x0.copy()

    if max_iter is None:
        max_iter = n

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

    for i in range(max_iter):
        Ap = A @ p
        alpha = rsold / (p.T @ Ap)
        x = x + alpha * p
        r = r - alpha * Ap

        # Check convergence
        rsnew = r.T @ r
        if np.sqrt(rsnew) < tol:
            break

        p = r + (rsnew / rsold) * p
        rsold = rsnew

    return x
```

Generalized Minimal Residual Method (GMRES)

For general (non-symmetric) matrices, GMRES minimizes the residual norm $|b - Ax|_2$ over successive Krylov subspaces.

Algorithm (GMRES):

```
Input: Matrix A, right-hand side b, initial guess $x_0$
Output: Approximate solution x

$r_0 = b - Ax_0$
$\beta = \|r_0\|_2$
$q_1 = r_0/\beta$
For j = 1, 2, ..., until convergence
    Compute $w = Aq_j$
    For i = 1, 2, ..., j
        $h_{ij} = q_i^T w$
        $w = w - h_{ij} q_i$
    End For
    $h_{j+1,j} = \|w\|_2$
    If $h_{j+1,j} \approx 0$ then break
    $q_{j+1} = w/h_{j+1,j}$

    # Minimize $\|\beta e_1 - \bar{H}^k y\|_2$ to find $y^k$
    # Update $x_k = x_0 + Q_k y^k$
End For
```

The least-squares problem $\min_y |\beta e_1 - \bar{H}_j y|_2$ is typically solved using Givens rotations to transform $\bar{H}_j$ into upper triangular form.

Restarted GMRES: Since GMRES requires storing all basis vectors, a restarted version GMRES(m) is often used, which restarts after m iterations:

```python
def gmres_restarted(A, b, x0=None, m=20, tol=1e-10, max_restarts=100):
    """
    Restarted GMRES method for solving Ax = b

    Parameters:
    A: Square matrix
    b: Right-hand side vector
    x0: Initial guess (default: zero vector)
    m: Maximum subspace dimension before restart
    tol: Convergence tolerance
    max_restarts: Maximum number of restarts

    Returns:
    x: Approximate solution
    """
    n = len(b)
    if x0 is None:
        x = np.zeros(n)
    else:
        x = x0.copy()

    for restart in range(max_restarts):
        r = b - A @ x
        beta = np.linalg.norm(r)

        if beta < tol:
            break

        # Initialize Krylov subspace
        Q = np.zeros((n, m+1))
        H = np.zeros((m+1, m))
        Q[:, 0] = r / beta

        # Build Krylov subspace
        for j in range(m):
            # Arnoldi process
            w = A @ Q[:, j]

            for i in range(j+1):
                H[i, j] = Q[:, i].T @ w
                w = w - H[i, j] * Q[:, i]

            H[j+1, j] = np.linalg.norm(w)

            if abs(H[j+1, j]) < 1e-14:
                # Lucky breakdown
                m = j+1
                break

            Q[:, j+1] = w / H[j+1, j]

            # Apply Givens rotations to H
            # ... (implementation details omitted)

            # Check convergence
            if residual < tol:
                break

        # Solve least squares problem and update x
        y = np.linalg.lstsq(H[:m, :m], beta * np.eye(m+1, 1)[:m],
rcond=None)[0]
        x = x + Q[:, :m] @ y

    return x
```

Biconjugate Gradient Stabilized (BiCGSTAB)

BiCGSTAB is another Krylov subspace method for non-symmetric systems that avoids some of the issues with the original BiCG method:

```python
def bicgstab(A, b, x0=None, tol=1e-10, max_iter=None):
    """
    BiCGSTAB method for solving Ax = b

    Parameters:
    A: Square matrix
    b: Right-hand side vector
    x0: Initial guess
    tol: Convergence tolerance
    max_iter: Maximum number of iterations

    Returns:
    x: Approximate solution
    """
    n = len(b)
    if x0 is None:
        x = np.zeros(n)
    else:
        x = x0.copy()

    if max_iter is None:
        max_iter = n

    r = b - A @ x
    r0_hat = r.copy()  # Shadow residual

    rho_prev = 1
    alpha = 1
    omega = 1
    v = np.zeros(n)
    p = np.zeros(n)

    for i in range(max_iter):
        rho = r0_hat.T @ r

        # Check for breakdown
        if abs(rho) < 1e-14:
            break

        beta = (rho / rho_prev) * (alpha / omega)
        p = r + beta * (p - omega * v)

        v = A @ p
        alpha = rho / (r0_hat.T @ v)

        s = r - alpha * v
        t = A @ s
        omega = (t.T @ s) / (t.T @ t)

        x = x + alpha * p + omega * s
        r = s - omega * t

        # Check convergence
        if np.linalg.norm(r) < tol:
            break

        rho_prev = rho

    return x
```

Preconditioning

Preconditioning is crucial for accelerating the convergence of Krylov methods. Instead of solving $Ax = b$, we solve the equivalent system $M^{-1}Ax = M^{-1}b$ where M is a preconditioner that approximates A but is easier to invert.

Common Preconditioners:

- **Jacobi**: $M = \text{diag}(A)$
- **Symmetric Gauss-Seidel**: $M = (D + L)D^{-1}(D + L)^T$ where $A = D + L + L^T$
- **Incomplete LU (ILU)**: $M = \tilde{L}\tilde{U}$ where $\tilde{L}$ and $\tilde{U}$ are sparse approximations to the LU factors
- **Algebraic Multigrid (AMG)**: Hierarchical preconditioner based on coarse-grid corrections

Example (Preconditioned CG):

```
def preconditioned_cg(A, b, M_inv, x0=None, tol=1e-10, max_iter=None):
    """
    Preconditioned Conjugate Gradient method

    Parameters:
    A: Symmetric positive definite matrix
    b: Right-hand side vector
    M_inv: Function that computes M^{-1}v
    x0: Initial guess
    tol: Convergence tolerance
    max_iter: Maximum number of iterations

    Returns:
    x: Approximate solution
    """
    n = len(b)
    if x0 is None:
        x = np.zeros(n)
    else:
        x = x0.copy()

    if max_iter is None:
        max_iter = n

    r = b - A @ x
    z = M_inv(r)
    p = z.copy()
    rz_old = r.T @ z

    for i in range(max_iter):
        Ap = A @ p
        alpha = rz_old / (p.T @ Ap)
        x = x + alpha * p
        r = r - alpha * Ap

        # Check convergence
        if np.linalg.norm(r) < tol:
            break

        z = M_inv(r)
        rz_new = r.T @ z
        beta = rz_new / rz_old
        p = z + beta * p
        rz_old = rz_new

    return x
```

Krylov Methods for Matrix Functions

Krylov subspace methods can also be used to compute matrix functions $f(A)v$ without explicitly forming $f(A)$.

Functions of Matrices

Definition (Matrix Function): For a function f with sufficient regularity, $f(A)$ is defined via the Jordan canonical form or by a contour integral:

$$f(A) = \frac{1}{2\pi i} \oint_C f(z)(zI - A)^{-1} dz$$

where C is a contour enclosing the spectrum of A.

Example (Matrix Exponential): The matrix exponential e^A is especially important in solving systems of ODEs.

Krylov Approximation

The idea is to project the problem onto a smaller Krylov subspace:

$$f(A)v \approx |v|_2 Q_m f(H_m) e_1$$

where Q_m and H_m are from the Arnoldi (or Lanczos) process.

Algorithm (Arnoldi Method for $f(A)v$):

```
Input: Matrix A, vector v, function f
Output: Approximation to f(A)v

β = ||v||₂
q₁ = v/β
Apply m steps of Arnoldi to get Qₘ and Hₘ
Compute f(Hₘ) (small problem)
Return βQₘf(Hₘ)e₁
```

Implementation (Python):

```python
import numpy as np
from scipy.linalg import expm

def krylov_matrix_function(A, v, f, m=30):
    """
    Compute f(A)v using Krylov subspace approximation

    Parameters:
    A: Square matrix
    v: Vector
    f: Matrix function (e.g., expm for matrix exponential)
    m: Dimension of Krylov subspace

    Returns:
    w: Approximation to f(A)v
    """
    n = len(v)
    beta = np.linalg.norm(v)
    q = v / beta

    # Initialize Krylov subspace
    Q = np.zeros((n, m+1))
    H = np.zeros((m+1, m))
    Q[:, 0] = q

    # Arnoldi process
    for j in range(m):
        w = A @ Q[:, j]

        for i in range(j+1):
            H[i, j] = Q[:, i].T @ w
            w = w - H[i, j] * Q[:, i]

        H[j+1, j] = np.linalg.norm(w)

def krylov_matrix_function(A, v, f, m=30):
    """
    Compute f(A)v using Krylov subspace approximation

    Parameters:
    A: Square matrix
    v: Vector
    f: Matrix function (e.g., expm for matrix exponential)
    m: Dimension of Krylov subspace

    Returns:
    w: Approximation to f(A)v
    """
    n = len(v)
    beta = np.linalg.norm(v)
    q = v / beta

    # Initialize Krylov subspace
    Q = np.zeros((n, m+1))
    H = np.zeros((m+1, m))
    Q[:, 0] = q

    # Arnoldi process
    for j in range(m):
        w = A @ Q[:, j]

        for i in range(j+1):
            H[i, j] = Q[:, i].T @ w
            w = w - H[i, j] * Q[:, i]

        H[j+1, j] = np.linalg.norm(w)

        # Check for invariant subspace
        if abs(H[j+1, j]) < 1e-14:
            # Reduce dimension
            H_reduced = H[:j+1, :j+1]
            Q_reduced = Q[:, :j+1]

            # Compute f(H) (small matrix)
            f_H = f(H_reduced)

            # Project back to full space
```

```
                return beta * Q_reduced @ f_H[:, 0]

            Q[:, j+1] = w / H[j+1, j]

        # Compute f(H) (small matrix)
        H_reduced = H[:m, :m]
        f_H = f(H_reduced)

        # Project back to full space
        return beta * Q[:, :m] @ f_H[:, 0]
```