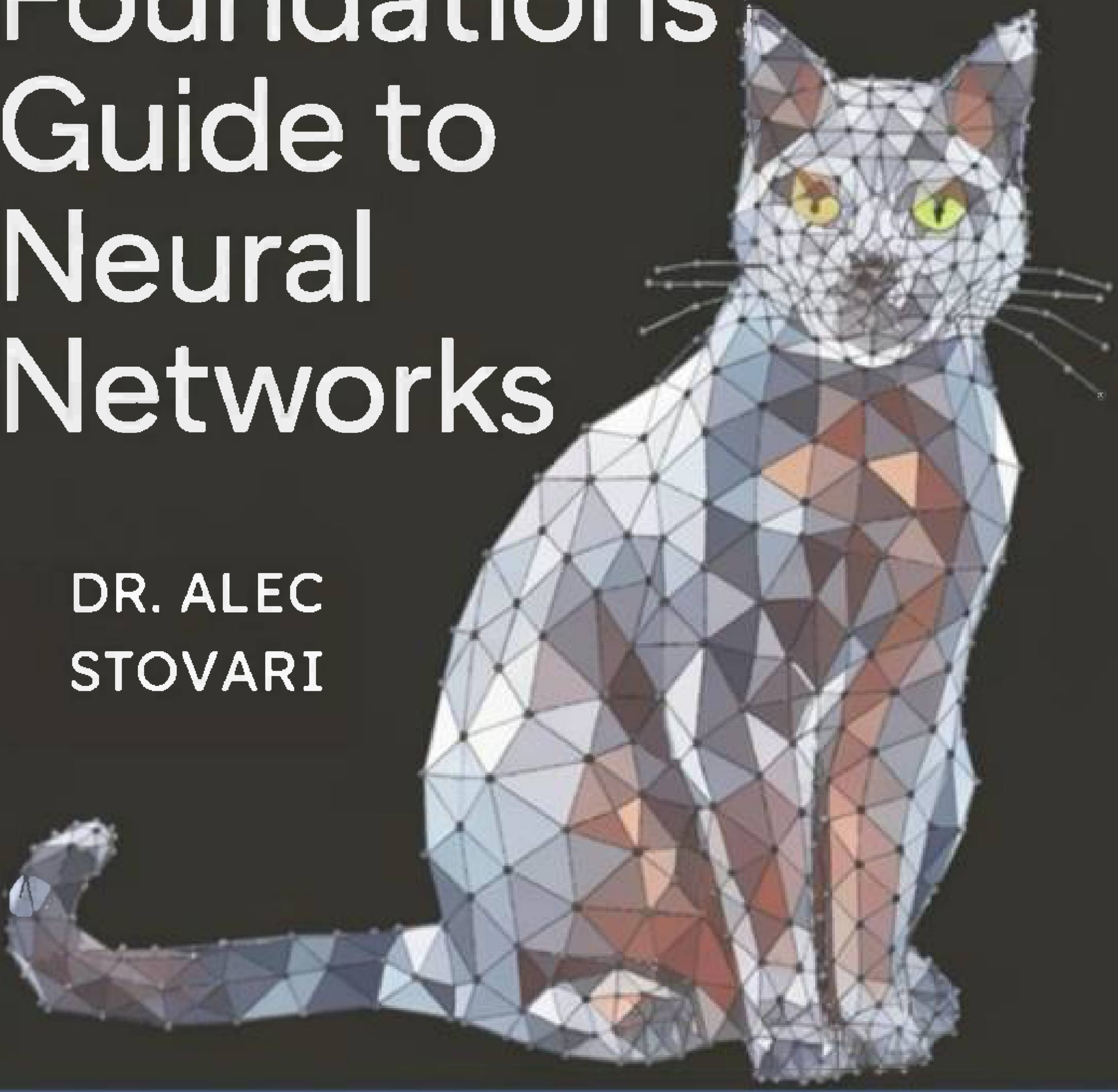
SONICT

Mathematical Foundations Guide to Neural Networks

DR. ALEC STOVARI



- Neural Network basics
- Backpropagation and optimization
- CNNs for image tasks
- RNNs for sequence data
- LSTMs for long-term dependencies
- Autoencoders for feature learning
- GANs for data generation
- Attention and Transformer models

Contents

Chapter 1

Matrix algebra and numerical methods

1.1 Recommended literature

[1] Grégoire Allaire et al. *Numerical linear algebra.* Vol. 55. 2008. Springer, 2008
[7] Rainer Kress. *Numerical analysis.* Vol. 181. Springer Science & Business Media, 2012
[13] L Richard. "Burden and J. Douglas Faires. Numerical analysis. Brooks". In: *Cole Publishing Company, Pacific Grove, California* 93950 (1997), p. 78

1.2 Norms

A **vector norm** is a function that assigns a non-negative length or size to each vector in a vector space. A vector norm $\|\cdot\|$ on $\mathbb{R}^n$ (or $\mathbb{C}^n$) satisfies the following properties for all vectors $\mathbf{u}, \mathbf{v}$ and scalar α:

1. **Non-negativity:** $\|\mathbf{u}\| \geq 0$ and $\|\mathbf{u}\| = 0$ if and only if $\mathbf{u} = \mathbf{0}$.
2. **Scalar multiplication:** $\|\alpha \mathbf{u}\| = |\alpha| \|\mathbf{u}\|$.
3. **Triangle inequality:** $\|\mathbf{u} + \mathbf{v}\| \leq \|\mathbf{u}\| + \|\mathbf{v}\|$.

Examples of vector norms:

1. **Manhattan norm (L_1 norm):**

$$\|\mathbf{u}\|_1 = \sum_{i=1}^{n} |u_i|$$

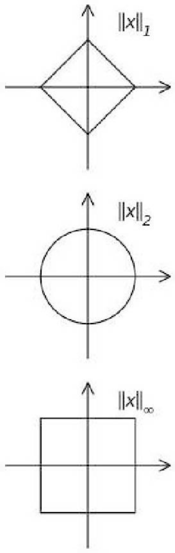
2. **Euclidean norm (L_2 norm):**

$$\|\mathbf{u}\|_2 = \left(\sum_{i=1}^{n} |u_i|^2 \right)^{1/2}$$

3. **Maximum norm (L_∞ norm):**

$$\|\mathbf{u}\|_\infty = \max_i |u_i|$$

Unit circle in vector norms:



A **matrix norm** is a natural extension of vector norms to matrices. For a matrix norm $\|\cdot\|$ on $\mathbb{R}^{m\times n}$, it satisfies the following properties for all matrices A, B and scalar α:

1. **Non-negativity:** $\|A\| \geq 0$ and $\|A\| = 0$ if and only if $A = 0$.
2. **Scalar multiplication:** $\|\alpha A\| = |\alpha| \|A\|$.
3. **Subadditivity (Triangle inequality):** $\|A + B\| \leq \|A\| + \|B\|$.
4. **Submultiplicativity:** $\|AB\| \leq \|A\| \|B\|$.

Examples of matrix norms:

1. **Frobenius norm:**
$$\|A\|_F = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2 \right)^{1/2}$$
2. **Induced (Operator) norm:** For a given vector norm $\| \cdot \|$, the induced matrix norm is defined as:
$$\|A\| = \sup_{\mathbf{x} \neq 0} \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|}$$
   (a) **Induced L_1 norm:**
$$\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{m} |a_{ij}|$$
   (b) **Induced L_2 norm (Spectral norm):**
$$\|A\|_2 = \sqrt{\lambda_{\max}(A^T A)}$$
   where $\lambda_{\max}$ denotes the largest eigenvalue.

   (c) **Induced L_∞ norm:**
$$\|A\|_\infty = \max_{1 \leq i \leq m} \sum_{j=1}^{n} |a_{ij}|$$

1.2.1 Applications of Norms in Numerical Mathematics

Norms are essential in numerical mathematics particularly in the context of error estimation and conditionality. For example, if $\mathbf{e} = \tilde{x} - x$ is the error vector (x - exact value, $\tilde{x}$ - estimate of x), its norm $\|\mathbf{e}\|$ gives a measure of the size of the error. We are calling it absolute error. In solving linear system or least square problems, for example, the norm of the residual $\|A\mathbf{x} - \mathbf{b}\|$ indicates how close the current solution $\mathbf{x}$ is to the actual solution. If the relative error $\frac{\|\mathbf{e}\|}{\|\mathbf{x}\|}$ is small, the computed result $\mathbf{x}$ has a high number of valid digits.

Conditionality

The **condition number** C_p is defined as

$$C_p = \frac{\left|\dfrac{\Delta y}{y}\right|}{\left|\dfrac{\Delta x}{x}\right|} = \frac{|\text{output relative error}|}{|\text{input relative error}|}.$$

If $C_p \approx 1$, the task is *well-conditioned*, whereas if $C_p > 100$, the task is *ill-conditioned.*

- The condition number of a matrix A with respect to a norm $\|\cdot\|$ is defined as:

$$\kappa(A) = \|A\|\|A^{-1}\|$$

- A high condition number indicates that the matrix is ill-conditioned, meaning small changes in the input can lead to large changes in the output, which is crucial for stability analysis in numerical algorithms.
- Norms help determine the sensitivity of the system $A\mathbf{x} = \mathbf{b}$. If A is ill-conditioned, small errors in $\mathbf{b}$ or in the elements of A can result in large errors in $\mathbf{x}$.

Example 1:

$$\begin{aligned} 101x + 10y &= 111 \\ 10x + y &= 11 \end{aligned} \quad \Rightarrow \quad \begin{aligned} x &= 1 \\ y &= 1 \end{aligned}$$

$$\begin{aligned} 101x + 10y &= 111 \\ 10x + y &= 11.1 \end{aligned} \quad \Rightarrow \quad \begin{aligned} x &= 0 \\ y &= 11.1 \end{aligned}$$

1.3 Systems of Linear Equations and Direct Methods

In linear algebra, solving systems of linear equations is a fundamental task. A system of linear equations can be represented in matrix form as:

$$A\mathbf{x} = \mathbf{b}$$

where A is a matrix of coefficients, $\mathbf{x}$ is a vector of unknowns, and $\mathbf{b}$ is a vector of constants. Several direct methods are used to find $\mathbf{x}$, including matrix inversion, Gaussian elimination, and matrix decomposition.

Matrix Inversion

One direct method is to invert the matrix A:

$$\hat{\mathbf{x}} = A^{-1}\mathbf{b}$$

This method is not always practical, especially for large matrices, due to the computational complexity of matrix inversion.

Gaussian Elimination

Gaussian elimination transforms the matrix equation $A\mathbf{x} = \mathbf{b}$ into an upper triangular form:

$$U\mathbf{x} = \tilde{\mathbf{b}}$$

where U is an upper triangular matrix. This transformation allows us to simplify the system for easier solution.

Matrix Decomposition

Matrix decomposition methods, such as LU decomposition, decompose A into a product of matrices that are easier to solve.

1.4 Gaussian Elimination without Row Replacement

Gaussian elimination without row replacement follows a straightforward approach to convert the matrix A into an upper triangular matrix U. This process involves the following steps:

1. **Forward Elimination:**
    - Start with the first row. Identify the first element of the row (pivot element).
    - Subtract multiples of the first row from the rows below it to create zeros in the first column below the pivot element.
    - Move to the second row and repeat the process to create zeros below the pivot element in the second column.
    - Continue this process for each subsequent row, always creating zeros below the pivot elements.

2. **Upper Triangular Form:**
    - By the end of the forward elimination process, the matrix A will be transformed into an upper triangular matrix U, where all the elements below the main diagonal are zeros.
3. **Matrix L:**
    - Simultaneously, a lower triangular matrix L is formed. The elements of L are the multipliers used in the forward elimination steps.
    - Specifically, for each elimination step $l > k$, the multiplier m used to eliminate the element A_{lk} is stored in L_{lk}.

Example 2, part 1:

Consider the system:

$$\begin{aligned} 2x_1 + 4x_2 - x_3 &= -5 \\ x_1 + x_2 - 3x_3 &= -9 \\ 4x_1 + x_2 + 2x_3 &= 9 \end{aligned}$$

Perform Gaussian elimination without row replacement and decompose A into L and U.

Solution Steps

1. **Initial augmented matrix:**

$$\left(\begin{array}{ccc|c} 2 & 4 & -1 & -5 \\ 1 & 1 & -3 & -9 \\ 4 & 1 & 2 & 9 \end{array}\right)$$

2. **First step of forward elimination:**

$$R_2 \leftarrow R_2 - \frac{1}{2}R_1$$

$$R_3 \leftarrow R_3 - 2R_1$$

Resulting in:

$$\left(\begin{array}{ccc|c} 2 & 4 & -1 & -5 \\ 0 & -1 & -2.5 & -6.5 \\ 0 & -7 & 4 & 19 \end{array}\right)$$

- The multipliers used are $\frac{1}{2}$ and 2, which are stored in L at positions L_{21} and L_{31}, respectively.

3. **Second step of forward elimination:**

$$R_3 \leftarrow R_3 - 7R_2$$

Resulting in:

$$\left(\begin{array}{ccc|c} 2 & 4 & -1 & -5 \\ 0 & -1 & -2.5 & -6.5 \\ 0 & 0 & 21.5 & 64.5 \end{array}\right)$$

- The multiplier used is 7, which is stored in L at position L_{32}.

4. **Resulting L and U matrices:**

$$L = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ 2 & 7 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 2 & 4 & -1 \\ 0 & -1 & -2.5 \\ 0 & 0 & 21.5 \end{pmatrix}$$

1.5 Gaussian Elimination with Partial Pivoting

Gaussian elimination with partial pivoting improves numerical stability by selecting the largest possible pivot element in each step. The procedure involves the following steps:

1. **Pivot Selection:**
   - In the first column, find the largest element in absolute value.
   - Swap the row containing this element with the first row.
   - This ensures the largest possible pivot element is used, reducing the risk of numerical instability.
2. **Forward Elimination:**
   - Subtract multiples of the first row from the rows below it to create zeros in the first column below the pivot element.
   - Move to the second row, select the largest element in the remaining submatrix, and swap rows if necessary.
   - Continue this process for each subsequent column and row.
3. **Upper Triangular Form:**
   - By the end of the forward elimination process, the matrix A will be transformed into an upper triangular matrix U.
4. **Permutation Matrix:**
   - Record the row swaps in a permutation matrix P.
5. **Matrix L:**
   - A lower triangular matrix L is simultaneously formed, where each entry L_{lk} (for $l > k$) represents the multiplier used to eliminate the element A_{lk} during forward elimination.

Example 2, part 2:

Solve the same system as in the previous section 1.4 using partial pivoting.

Solution Steps

1. **Initial augmented matrix:**

$$\left(\begin{array}{ccc|c} 2 & 4 & -1 & -5 \\ 1 & 1 & -3 & -9 \\ 4 & 1 & 2 & 9 \end{array}\right)$$

2. **Pivot selection and row swap:**

   - The largest element in the first column is 4.
   - Swap R_1 and R_3:

$$\left(\begin{array}{ccc|c} 4 & 1 & 2 & 9 \\ 1 & 1 & -3 & -9 \\ 2 & 4 & -1 & -5 \end{array}\right)$$

   - Record this swap in the permutation matrix P:

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

3. **First step of forward elimination:**

$$R_2 \leftarrow R_2 - \frac{1}{4}R_1$$

$$R_3 \leftarrow R_3 - \frac{1}{2}R_1$$

Resulting in:

$$\left(\begin{array}{ccc|c} 4 & 1 & 2 & 9 \\ 0 & 0.75 & -3.5 & -11.25 \\ 0 & 3.5 & -2 & -9.5 \end{array}\right)$$

- The multipliers used are $\frac{1}{4}$ and $\frac{1}{2}$, which are stored in L at positions L_{21} and L_{31}, respectively.

4. **Second step of pivot selection and row swap:**

   - The largest element in the second column (below the first row) is 3.5.
   - Swap R_2 and R_3:

$$\left(\begin{array}{ccc|c} 4 & 1 & 2 & 9 \\ 0 & 3.5 & -2 & -9.5 \\ 0 & 0.75 & -3.5 & -11.25 \end{array}\right)$$

   - Record this swap in the permutation matrix P:

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

5. **Second step of forward elimination:**

$$R_3 \leftarrow R_3 - \frac{3}{14} R_2$$

Resulting in:

$$\left(\begin{array}{ccc|c} 4 & 1 & 2 & 9 \\ 0 & 3.5 & -2 & -9.5 \\ 0 & 0 & -3.0714 & -9.2143 \end{array}\right)$$

- The multiplier used is $\frac{3}{14}$, which is stored in L at position L_{32}. Multipliers at positions L_{21} and L_{31} swap.

6. **Resulting L, U, and P matrices:**

$$L = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ \frac{1}{4} & \frac{3}{14} & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 4 & 1 & 2 \\ 0 & 3.5 & -2 \\ 0 & 0 & -3.0714 \end{pmatrix}, \quad P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

1.5.1 Advantages of Gaussian Elimination with Partial Pivoting

Gaussian elimination with partial pivoting offers several significant advantages, particularly in terms of numerical stability and accuracy. Below are the primary advantages of this method:

1. **Improved Numerical Stability**

   Partial pivoting helps mitigate numerical instability that can arise due to small pivot elements. When small pivot elements are used, the elimination process can amplify rounding errors, leading to inaccurate results. By always choosing the largest available pivot element in the current column, partial pivoting reduces the risk of such errors.

   Example: Consider a system of equations where the pivot element is very small:

$$\begin{aligned} 0.0001x_1 + x_2 &= 1 \\ x_1 + x_2 &= 2 \end{aligned}$$

   Without partial pivoting, using 0.0001 as a pivot can lead to significant rounding errors. Partial pivoting would swap the rows to use 1 as the pivot, avoiding these issues.

2. **Enhanced Accuracy**

   By ensuring that the largest available pivot element is used at each step, partial pivoting minimizes the relative errors in the computations. This leads to more accurate solutions, especially for large systems of equations or those with widely varying coefficients.

Example: For a system with coefficients of varying magnitudes:

$$\begin{aligned} 2x_1 + 3x_2 &= 5 \\ 1000x_1 + 1001x_2 &= 2000 \end{aligned}$$

Partial pivoting would ensure the larger coefficient 1000 is used first, improving the accuracy of the subsequent computations.

3. **Robustness in Handling Singular or Nearly Singular Matrices**

   Partial pivoting can help in identifying and dealing with singular or nearly singular matrices. In such cases, small pivot elements indicate potential singularity. By swapping rows to bring larger elements to the pivot position, partial pivoting can sometimes transform the matrix into a form where the solution is more apparent or confirm that the matrix is singular.

1.5.2 Full Pivoting

Full pivoting, also known as complete pivoting, is an extension of partial pivoting used in Gaussian elimination. In full pivoting, at each step of the elimination process, the algorithm searches for the largest absolute element in the entire submatrix (not just the column) to use as the pivot. This element is then swapped to the current pivot position by exchanging both rows and columns. Full pivoting provides even greater numerical stability than partial pivoting, as it minimizes the risk of rounding errors by consistently choosing the most significant available pivot element. However, this increased stability comes at the cost of additional computational complexity and time, as the algorithm must perform a more extensive search and additional row and column swaps.

1.6 Matrix Decompositions

1.6.1 LU Decomposition

The LU decomposition expresses a matrix A as the product of two matrices:

$$A = L \cdot U$$

where:

- L is a lower triangular matrix with 1s on the main diagonal.
- U is an upper triangular matrix obtained through Gaussian elimination.

To compute L, during Gaussian elimination, a multiple c of the k-th row is added to the l-th row (for $l > k$) to create a zero at position l, k. The value $-c$ is then placed in the l, k position of matrix L.

In the general case:

$$P \cdot A = L \cdot U$$

where P is a permutation matrix that accounts for any row exchanges in A.

1.6.1.1 Solving Systems of Linear Equations using LU decomposition

Given a system $Ax = b$ and using the LU decomposition $A = LU$, we have:

$$Ax = LUx = b$$

Let $y = Ux$. We then solve:

$$Ly = b$$

followed by solving:

$$Ux = y$$

Example 2, part 3:

We may finish solving system

$$\begin{aligned} 2x_1 + 4x_2 - x_3 &= -5 \\ x_1 + x_2 - 3x_3 &= -9 \\ 4x_1 + x_2 + 2x_3 &= 9 \end{aligned}$$

as we started in section 1.4.

Solve $Ly = b$:

$$\begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ 2 & 7 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} -5 \\ -9 \\ 9 \end{pmatrix}$$

$$y_1 = -5$$

$$\frac{1}{2}y_1 + y_2 = -9 \Rightarrow y_2 = -9 - \frac{1}{2}(-5) = -9 + 2.5 = -6.5$$

$$2y_1 + 7y_2 + y_3 = 9 \Rightarrow y_3 = 9 - 2(-5) - 7(-6.5) = 9 + 10 + 45.5 = 64.5$$

$$y = \begin{pmatrix} -5 \\ -6.5 \\ 64.5 \end{pmatrix}$$

Solve $Ux = y$:

$$\begin{pmatrix} 2 & 4 & -1 \\ 0 & -1 & -2.5 \\ 0 & 0 & 21.5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -5 \\ -6.5 \\ 64.5 \end{pmatrix}$$

$$x_3 = \frac{64.5}{21.5} = 3$$

$$-x_2 - 2.5 \cdot 3 = -6.5 \Rightarrow x_2 = 6.5 - 7.5 = -1$$

$$2x_1 + 4 \cdot (-1) - 1 \cdot 3 = -5 \Rightarrow 2x_1 - 4 - 3 = -5 \Rightarrow 2x_1 - 7 = -5 \Rightarrow 2x_1 = 2 \Rightarrow x_1 = 1$$

$$x = \begin{pmatrix} 1 \\ -1 \\ 3 \end{pmatrix}$$

Alternatively, if a permutation matrix P is involved:

$$PA = LU$$

then:

$$PAx = LUx = Pb$$

We solve:

$$Ly = Pb$$

followed by:

$$Ux = y$$

Example 2, part 4:

We may finish solving system

$$\begin{aligned} 2x_1 + 4x_2 - x_3 &= -5 \\ x_1 + x_2 - 3x_3 &= -9 \\ 4x_1 + x_2 + 2x_3 &= 9 \end{aligned}$$

as we started in section 1.5.

Solve $Ly = Pb$ for y:

$$\begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ \frac{1}{4} & \frac{3}{14} & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 9 \\ -5 \\ -9 \end{pmatrix}$$

$$y_1 = 9$$

$$\frac{1}{2}y_1 + y_2 = -5 \Rightarrow y_2 = -5 - \frac{1}{2}(9) = -5 - 4.5 = -9.5$$

$$\frac{1}{4}y_1 + \frac{3}{14}y_2 + y_3 = -9 \Rightarrow y_3 = -9 - \frac{1}{4}(9) - \frac{3}{14}(-9.5) = -9 - 2.25 + 2.0357 = -9.2143$$

$$y = \begin{pmatrix} 9 \\ -9.5 \\ -9.2143 \end{pmatrix}$$

Solve $Ux = y$ for x:

$$\begin{pmatrix} 4 & 1 & 2 \\ 0 & 3.5 & -2 \\ 0 & 0 & -3.0714 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 9 \\ -9.5 \\ -9.2143 \end{pmatrix}$$

$$x_3 = \frac{-9.2143}{-3.0714} = 3$$

$$3.5x_2 - 2(3) = -9.5 \Rightarrow x_2 = \frac{-9.5+6}{3.5} = -1$$

$$4x_1 + x_2 + 2x_3 = 9 \Rightarrow 4x_1 - 1 + 6 = 9 \Rightarrow 4x_1 + 5 = 9 \Rightarrow 4x_1 = 4 \Rightarrow x_1 = 1$$

$$x = \begin{pmatrix} 1 \\ -1 \\ 3 \end{pmatrix}$$

1.6.1.2 Calculating the Inverse Matrix using LU decomposition

To find the inverse of A using LU decomposition, we have:

$$A = LU$$

Thus:

$$A^{-1} = (LU)^{-1} = U^{-1}L^{-1}$$

With a permutation matrix P:

$$PA = LU$$

Therefore:

$$(PA)^{-1} = (LU)^{-1}$$
$$A^{-1}P^{-1} = U^{-1}L^{-1}$$

and so:

$$A^{-1} = U^{-1}L^{-1}P$$

1.6.2 QR Decomposition

For a given $m \times n$ matrix $\mathbf{A}$ with $m \geq n$:

$$\mathbf{A} = \mathbf{QR}$$

where:

- $\mathbf{Q}$ is an $m \times m$ orthogonal matrix ($\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$).
- $\mathbf{R}$ is an $m \times n$ upper triangular matrix.

QR Decomposition Algorithm using Gram-Schmidt Process

1. For each column vector $\mathbf{a}_i$ of $\mathbf{A}$, compute:

$$\mathbf{u}_i = \mathbf{a}_i - \sum_{j=1}^{i-1} \text{proj}_{\mathbf{u}_j}(\mathbf{a}_i)$$

where $\text{proj}_{\mathbf{u}_j}(\mathbf{a}_i) = \left(\frac{\mathbf{u}_j^T \mathbf{a}_i}{\mathbf{u}_j^T \mathbf{u}_j}\right) \mathbf{u}_j$.

2. Normalize $\mathbf{u}_i$ to get $\mathbf{q}_i$:

$$\mathbf{q}_i = \frac{\mathbf{u}_i}{\|\mathbf{u}_i\|}$$

3. Form $\mathbf{Q}$ from the orthonormal vectors .
4. To compute $\mathbf{R}$ you can use the fact that $\mathbf{Q}$ is orthogonal matrix and therefore you can simply compute its inverse $\mathbf{Q}^{-1} = \mathbf{Q}^T$. From that you can see that $\mathbf{R}$ consists of the coefficients of the projection.

Example 3: QR Decomposition of a 3x3 Matrix using Gram-Schmidt Process

Consider the matrix:

$$\mathbf{A} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & -1 \\ 1 & -1 & 2 \end{pmatrix}$$

1. **Compute the first column of Q**:

$$\mathbf{a}_1 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

$$\|\mathbf{a}_1\| = \sqrt{1^2 + 1^2 + 1^2} = \sqrt{3}$$

$$\mathbf{q}_1 = \frac{\mathbf{a}_1}{\|\mathbf{a}_1\|} = \frac{1}{\sqrt{3}} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \end{pmatrix}$$

2. **Compute the second column of Q**:

$$\mathbf{a}_2 = \begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix}$$

$$\text{proj}_{\mathbf{q}_1}(\mathbf{a}_2) = \left(\mathbf{q}_1^T \mathbf{a}_2\right) \mathbf{q}_1 = \left(\frac{1}{\sqrt{3}} \cdot 1 + \frac{1}{\sqrt{3}} \cdot 1 + \frac{1}{\sqrt{3}} \cdot (-1)\right) \mathbf{q}_1 = \left(\frac{1+1-1}{\sqrt{3}}\right) \mathbf{q}_1 = \frac{1}{\sqrt{3}} \mathbf{q}_1$$

$$\text{proj}_{\mathbf{q}_1}(\mathbf{a}_2) = \frac{1}{\sqrt{3}} \begin{pmatrix} \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \end{pmatrix} = \frac{1}{3} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{pmatrix}$$

$$\mathbf{u}_2 = \mathbf{a}_2 - \text{proj}_{\mathbf{q}_1}(\mathbf{a}_2) = \begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix} - \begin{pmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{pmatrix} = \begin{pmatrix} \frac{3}{3} - \frac{1}{3} \\ \frac{3}{3} - \frac{1}{3} \\ -\frac{3}{3} - \frac{1}{3} \end{pmatrix} = \begin{pmatrix} \frac{2}{3} \\ \frac{2}{3} \\ -\frac{4}{3} \end{pmatrix}$$

$$\|\mathbf{u}_2\| = \sqrt{\left(\frac{2}{3}\right)^2 + \left(\frac{2}{3}\right)^2 + \left(-\frac{4}{3}\right)^2} = \sqrt{\frac{4}{9} + \frac{4}{9} + \frac{16}{9}} = \sqrt{\frac{24}{9}} = \sqrt{\frac{8}{3}}$$

$$\mathbf{q}_2 = \frac{\mathbf{u}_2}{\|\mathbf{u}_2\|} = \frac{1}{\sqrt{\frac{8}{3}}} \begin{pmatrix} \frac{2}{3} \\ \frac{2}{3} \\ -\frac{4}{3} \end{pmatrix} = \sqrt{\frac{3}{8}} \begin{pmatrix} \frac{2}{3} \\ \frac{2}{3} \\ -\frac{4}{3} \end{pmatrix} = \begin{pmatrix} \frac{2}{\sqrt{8 \cdot 3}} \\ \frac{2}{\sqrt{8 \cdot 3}} \\ -\frac{4}{\sqrt{8 \cdot 3}} \end{pmatrix} = \begin{pmatrix} \frac{2}{2\sqrt{6}} \\ \frac{2}{2\sqrt{6}} \\ -\frac{4}{2\sqrt{6}} \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{6}} \\ \frac{1}{\sqrt{6}} \\ -\frac{2}{\sqrt{6}} \end{pmatrix}$$

3. **Compute the third column of Q**:

$$\mathbf{a}_3 = \begin{pmatrix} 0 \\ -1 \\ 2 \end{pmatrix}$$

$$\text{proj}_{\mathbf{q}_1}(\mathbf{a}_3) = \left(\mathbf{q}_1^T \mathbf{a}_3\right) \mathbf{q}_1 = \left(\frac{1}{\sqrt{3}} \cdot 0 + \frac{1}{\sqrt{3}} \cdot (-1) + \frac{1}{\sqrt{3}} \cdot 2\right) \mathbf{q}_1 = \left(0 - \frac{1}{\sqrt{3}} + \frac{2}{\sqrt{3}}\right) \mathbf{q}_1 = \frac{1}{\sqrt{3}} \mathbf{q}_1$$

$$\text{proj}_{\mathbf{q}_1}(\mathbf{a}_3) = \frac{1}{\sqrt{3}} \begin{pmatrix} \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \end{pmatrix} = \frac{1}{3} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{pmatrix}$$

Now compute the projection of $\mathbf{a}_3$ onto $\mathbf{q}_2$:

$$\text{proj}_{\mathbf{q}_2}(\mathbf{a}_3) = \left(\mathbf{q}_2^T \mathbf{a}_3\right) \mathbf{q}_2 = \left(\frac{1}{\sqrt{6}} \cdot 0 + \frac{1}{\sqrt{6}} \cdot (-1) + -\frac{2}{\sqrt{6}} \cdot 2\right) \mathbf{q}_2 = \left(0 - \frac{1}{\sqrt{6}} - \frac{4}{\sqrt{6}}\right) \mathbf{q}_2 = -\frac{5}{\sqrt{6}} \mathbf{q}_2$$

$$\text{proj}_{\mathbf{q}_2}(\mathbf{a}_3) = -\frac{5}{\sqrt{6}} \begin{pmatrix} \frac{1}{\sqrt{6}} \\ \frac{1}{\sqrt{6}} \\ -\frac{2}{\sqrt{6}} \end{pmatrix} = -\frac{5}{6} \begin{pmatrix} 1 \\ 1 \\ -2 \end{pmatrix} = \begin{pmatrix} -\frac{5}{6} \\ -\frac{5}{6} \\ \frac{10}{6} \end{pmatrix} = \begin{pmatrix} -\frac{5}{6} \\ -\frac{5}{6} \\ \frac{5}{3} \end{pmatrix}$$

$$\mathbf{u}_3 = \mathbf{a}_3 - \text{proj}_{\mathbf{q}_1}(\mathbf{a}_3) - \text{proj}_{\mathbf{q}_2}(\mathbf{a}_3) = \begin{pmatrix} -\frac{1}{3} \\ -\frac{4}{3} \\ \frac{5}{3} \end{pmatrix} - \begin{pmatrix} -\frac{5}{6} \\ -\frac{5}{6} \\ \frac{5}{3} \end{pmatrix} = \begin{pmatrix} -\frac{1}{3} + \frac{5}{6} \\ -\frac{4}{3} + \frac{5}{6} \\ \frac{5}{3} - \frac{5}{3} \end{pmatrix} = \begin{pmatrix} -\frac{2}{6} + \frac{5}{6} \\ -\frac{8}{6} + \frac{5}{6} \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{3}{6} \\ -\frac{3}{6} \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ 0 \end{pmatrix}$$

$$\|\mathbf{u}_3\| = \sqrt{\left(\frac{1}{2}\right)^2 + \left(-\frac{1}{2}\right)^2 + 0^2} = \sqrt{\frac{1}{4} + \frac{1}{4}} = \sqrt{\frac{2}{4}} = \sqrt{\frac{1}{2}}$$

$$\mathbf{q}_3 = \frac{\mathbf{u}_3}{\|\mathbf{u}_3\|} = \frac{1}{\sqrt{\frac{1}{2}}} \begin{pmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ 0 \end{pmatrix} = \sqrt{2} \begin{pmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \\ 0 \end{pmatrix}$$

4. **Construct Q and R**:

$$\mathbf{Q} = \begin{pmatrix} \mathbf{q}_1 & \mathbf{q}_2 & \mathbf{q}_3 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{3}} & -\frac{2}{\sqrt{6}} & 0 \end{pmatrix}$$

Now we compute $\mathbf{R} = \mathbf{Q}^T \mathbf{A}$:

5. **First row of R**:

$$\mathbf{R}_{11} = \mathbf{q}_1^T\mathbf{a}_1 = \frac{1}{\sqrt{3}} \cdot 1 + \frac{1}{\sqrt{3}} \cdot 1 + \frac{1}{\sqrt{3}} \cdot 1 = \sqrt{3}$$

$$\mathbf{R}_{12} = \mathbf{q}_1^T\mathbf{a}_2 = \frac{1}{\sqrt{3}} \cdot 1 + \frac{1}{\sqrt{3}} \cdot 1 + \frac{1}{\sqrt{3}} \cdot (-1) = \frac{1}{\sqrt{3}}$$

$$\mathbf{R}_{13} = \mathbf{q}_1^T\mathbf{a}_3 = \frac{1}{\sqrt{3}} \cdot 0 + \frac{1}{\sqrt{3}} \cdot (-1) + \frac{1}{\sqrt{3}} \cdot 2 = \frac{1}{\sqrt{3}}$$

6. **Second row of R**:

$$\mathbf{R}_{22} = \mathbf{q}_2^T\mathbf{a}_2 = \frac{1}{\sqrt{6}} \cdot 1 + \frac{1}{\sqrt{6}} \cdot 1 + -\frac{2}{\sqrt{6}} \cdot (-1) = \frac{4}{\sqrt{6}}$$

$$\mathbf{R}_{23} = \mathbf{q}_2^T\mathbf{a}_3 = \frac{1}{\sqrt{6}} \cdot 0 + \frac{1}{\sqrt{6}} \cdot (-1) + -\frac{2}{\sqrt{6}} \cdot 2 = -\frac{5}{\sqrt{6}}$$

7. **Third row of R**:

$$\mathbf{R}_{33} = \mathbf{q}_3^T\mathbf{a}_3 = \frac{1}{\sqrt{2}} \cdot 0 + -\frac{1}{\sqrt{2}} \cdot (-1) + 0 \cdot 2 = \frac{1}{\sqrt{2}}$$

So, **R** is:

$$\mathbf{R} = \begin{pmatrix} \sqrt{3} & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix}$$

The QR decomposition of **A** is:

$$\mathbf{A} = \mathbf{QR} = \begin{pmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{3}} & -\frac{2}{\sqrt{6}} & 0 \end{pmatrix} \begin{pmatrix} \sqrt{3} & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \\ 0 & \frac{4}{\sqrt{6}} & -\frac{5}{\sqrt{6}} \\ 0 & 0 & \frac{1}{\sqrt{2}} \end{pmatrix}$$

1.6.2.1 Numerical Stability of the Gram-Schmidt Process

The Gram-Schmidt process is a classical method for orthogonalizing a set of vectors, and it forms the basis for one approach to QR decomposition. However, it is well-known that the classical Gram-Schmidt process can suffer from numerical instability, especially when working with finite-precision arithmetic as is common in digital computers. To address the numerical instability of the Gram-Schmidt process, other approaches such as Householder Reflections and Givens Rotations are often used. These methods improve the numerical stability and accuracy of the QR decomposition.

Sources of Instability

Loss of Orthogonality: During the orthogonalization process, small errors can accumulate and cause the resulting vectors to lose their orthogonality. This is primarily due to the subtractions involved in the projection steps, which can lead to significant rounding errors.

Round-off Errors: In each step of the Gram-Schmidt process, the projection of one vector onto another involves several floating-point operations. The finite precision of these operations introduces round-off errors. These errors can grow exponentially, especially when the input matrix has columns that are nearly linearly dependent.

Condition Number: The condition number of the input matrix also affects the stability. Matrices with high condition numbers are more prone to numerical instability during the orthogonalization process.

1.6.2.2 Eigenvalue Computations using QR decomposition

The QR algorithm is one of the most efficient methods for computing eigenvalues and eigenvectors of a matrix. By iteratively applying QR decomposition and forming a sequence of matrices, the QR algorithm converges to a triangular matrix whose diagonal elements are the eigenvalues of the original matrix.

The QR algorithm works by leveraging the fact that orthogonal transformations preserve eigenvalues. At each iteration, the matrix is decomposed into an orthogonal matrix $\mathbf{Q}$ and an upper triangular matrix $\mathbf{R}$. Multiplying these factors in reverse order (i.e., $\mathbf{A}_{k+1} = \mathbf{R}_k\mathbf{Q}_k$) leads to a new matrix that is similar to the original matrix. This process gradually isolates the eigenvalues in the diagonal elements of the resulting upper triangular matrix.

Example 4: Computing Eigenvalues using the QR Algorithm

1. Initial Step: Compute the QR Decomposition of $\mathbf{A}$:

   $$\mathbf{A} = \mathbf{Q}_0\mathbf{R}_0$$

   where $\mathbf{Q}_0$ is an orthogonal matrix and $\mathbf{R}_0$ is an upper triangular matrix.

2. Form the Next Matrix:

   $$\mathbf{A}_1 = \mathbf{R}_0\mathbf{Q}_0$$

3. Iterate the Process: Repeat the QR decomposition on $\mathbf{A}_1$ to get:

   $$\mathbf{A}_1 = \mathbf{Q}_1\mathbf{R}_1 \quad \text{and} \quad \mathbf{A}_2 = \mathbf{R}_1\mathbf{Q}_1$$

   Continue this process for several iterations until $\mathbf{A}_k$ converges to an upper triangular matrix $\mathbf{T}$.

4. Eigenvalues: The diagonal elements of the final upper triangular matrix $\mathbf{T}$ are the eigenvalues of the original matrix $\mathbf{A}$.

Consider the matrix:

$$\mathbf{A} = \begin{pmatrix} 4 & 1 & -2 \\ 1 & 2 & 0 \\ -2 & 0 & 3 \end{pmatrix}.$$

For the given matrix $\mathbf{A}$, performing the QR algorithm will eventually yield:

$$\mathbf{T} = \begin{pmatrix} 5.7321 & 0 & 0 \\ 0 & 2.2679 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Thus, the eigenvalues of $\mathbf{A}$ are 5.7321, 2.2679, and 1.

1.6.2.3 Matrix inverses and determinants using QR decomposition

QR decomposition provides a numerically stable way to compute matrix inverses, determinants, and ranks. The orthogonal matrix $\mathbf{Q}$ preserves the norm, which helps in maintaining numerical stability and reducing the effects of round-off errors in computations.

Example 5: Computing the Inverse using QR Decomposition

Consider the matrix:

$$\mathbf{A} = \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}$$

1. Compute the QR Decomposition:

$$\mathbf{A} = \mathbf{QR}$$

   where:

$$\mathbf{Q} = \begin{pmatrix} \frac{2}{\sqrt{5}} & \frac{1}{\sqrt{5}} \\ \frac{1}{\sqrt{5}} & -\frac{2}{\sqrt{5}} \end{pmatrix} \quad \text{and} \quad \mathbf{R} = \begin{pmatrix} \sqrt{5} & \sqrt{5} \\ 0 & -\sqrt{5} \end{pmatrix}$$

2. Solve $\mathbf{QRA}^{-1} = \mathbf{I}$:

$$\mathbf{RA}^{-1} = \mathbf{Q}^T$$

3. Compute $\mathbf{A}^{-1}$ by Back-Substitution:

$$\mathbf{A}^{-1} = \mathbf{R}^{-1}\mathbf{Q}^T$$

   where:

$$\mathbf{R}^{-1} = \begin{pmatrix} \frac{1}{\sqrt{5}} & \frac{1}{\sqrt{5}} \\ 0 & -\frac{1}{\sqrt{5}} \end{pmatrix}$$

4. Result:

$$\mathbf{A}^{-1} = \begin{pmatrix} \frac{3}{5} & -\frac{1}{5} \\ -\frac{1}{5} & \frac{2}{5} \end{pmatrix}$$

Example 6: Computing the Determinant using QR Decomposition

Consider the same matrix $\mathbf{A}$:

$$\mathbf{A} = \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}$$

1. Compute the QR Decomposition:

$$\mathbf{A} = \mathbf{QR}$$

2. Compute the Determinant: Since $\mathbf{Q}$ is orthogonal, $|\mathbf{Q}| = \pm 1$. The sign is determined based on whether the transformation represented by Q includes a reflection, $|\mathbf{Q}| = -1$, or not $|\mathbf{Q}| = +1$. Therefore, the determinant of $\mathbf{A}$ is the product of the diagonal elements of $\mathbf{R}$:

$$\det(\mathbf{A}) = \det(\mathbf{Q}) \cdot \det(\mathbf{R}) = -1 \cdot \left(\sqrt{5} \cdot \left(-\sqrt{5}\right)\right) = 5.$$

1.6.3 Cholesky Decomposition

For a given symmetric positive-definite matrix $\mathbf{A}$:

$$\mathbf{A} = \mathbf{LL}^T$$

where:

- $\mathbf{L}$ is a lower triangular matrix with positive diagonal entries.
- $\mathbf{L}^T$ is the transpose of $\mathbf{L}$.

1.6.3.1 Cholesky Decomposition Algorithm

Let us start with $A \in \mathcal{R}^{3\times 3}$. Given:

$$\mathbf{L} = \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix}$$

Now, compute the product $\mathbf{L} \cdot \mathbf{L}^T$:

$$\mathbf{LL}^T = \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} L_{11} & L_{21} & L_{31} \\ 0 & L_{22} & L_{32} \\ 0 & 0 & L_{33} \end{pmatrix}$$

The resulting matrix $\mathbf{A} = \mathbf{LL}^T$ is:

$$\mathbf{A} = \begin{pmatrix} L_{11}^2 & L_{11}L_{21} & L_{11}L_{31} \\ L_{21}L_{11} & L_{21}^2 + L_{22}^2 & L_{21}L_{31} + L_{22}L_{32} \\ L_{31}L_{11} & L_{31}L_{21} + L_{32}L_{22} & L_{31}^2 + L_{32}^2 + L_{33}^2 \end{pmatrix}$$

Using this we now may express each element of A using elements of L. Generally you do it using the following formulae.

1. Initialize **L** as a zero matrix.
2. For each i from 1 to n:
    - For j from 1 to $i-1$:

$$L_{ij} = \frac{1}{L_{jj}} \left(A_{ij} - \sum_{k=1}^{j-1} L_{ik} L_{jk} \right)$$

    - Compute the diagonal entries:

$$L_{ii} = \sqrt{A_{ii} - \sum_{k=1}^{i-1} L_{ik}^2}$$

If **A** is positive-definite all elements of matrix L are real.

Example 7:

Consider the real matrix A that is not positive-definite:

$$A = \begin{pmatrix} 1 & 2 & -1 \\ 2 & 2 & 4 \\ -1 & 4 & 8 \end{pmatrix}$$

We want to find the Cholesky decomposition of A, i.e., $A = LL^T$, where L is a lower triangular matrix.

$$\begin{aligned}
L_{11} &= \sqrt{A_{11}} = \sqrt{1} = 1 \\
L_{21} &= \frac{A_{21}}{L_{11}} = \frac{2}{1} = 2 \\
L_{31} &= \frac{A_{31}}{L_{11}} = \frac{-1}{1} = -1 \\
L_{22} &= \sqrt{A_{22} - L_{21}^2} = \sqrt{2 - 2^2} = \sqrt{2-4} = \sqrt{-2} = i\sqrt{2} \\
L_{32} &= \frac{A_{32} - L_{31}L_{21}}{L_{22}} = \frac{4 - (-1 \cdot 2)}{i\sqrt{2}} = \frac{4+2}{i\sqrt{2}} = \frac{6}{i\sqrt{2}} = -i3\sqrt{2} \\
L_{33} &= \sqrt{A_{33} - L_{31}^2 - L_{32}^2} = \sqrt{8 - (-1)^2 - (-i3\sqrt{2})^2} = \sqrt{8 - 1 + 18} = 5
\end{aligned}$$

So, the matrix L is:

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & i\sqrt{2} & 0 \\ -1 & -i3\sqrt{2} & 5 \end{pmatrix}$$

1.6.4 Crout decomposition

Consider a general tridiagonal matrix A:

$$A = \begin{pmatrix} a_{11} & a_{12} & 0 & \cdots & 0 \\ a_{21} & a_{22} & a_{23} & \cdots & 0 \\ 0 & a_{32} & a_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & a_{n-1,n} \\ 0 & 0 & 0 & a_{n,n-1} & a_{nn} \end{pmatrix}$$

We aim to find matrices L and U such that $A = LU$, where L is a lower triangular matrix and U is an upper triangular matrix with unit diagonal elements.

The general forms of L and U are:

$$L = \begin{pmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \cdots & 0 \\ 0 & l_{32} & l_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & l_{n,n-1} & l_{nn} \end{pmatrix}$$

$$U = \begin{pmatrix} 1 & u_{12} & 0 & \cdots & 0 \\ 0 & 1 & u_{23} & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & u_{n-1,n} \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Matrix multiplication $A = LU$ is:

$$A = LU = \begin{pmatrix} l_{11} & l_{11}u_{12} & 0 & \cdots & 0 \\ l_{21} & l_{21}u_{12} + l_{22} & l_{22}u_{23} & \cdots & 0 \\ 0 & l_{32} & l_{32}u_{23} + l_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & l_{n-1,n}u_{n-1,n} \\ 0 & 0 & 0 & l_{n,n-1} & l_{n,n-1}u_{n-1,n} + l_{nn} \end{pmatrix}$$

Expressions for Elements of L and U

For $i = j$:

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}u_{ki}}$$

For $i > j$:

$$l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj}$$

For $i < j$:

$$u_{ij} = \frac{1}{l_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \right)$$

Doolittle decomposition is another method for decomposing a matrix into the product of a lower triangular matrix and an upper triangular matrix, but with a different arrangement. In Doolittle decomposition, the lower triangular matrix L has unit diagonal elements, while the upper triangular matrix U has non-unit diagonal elements. This is in contrast to Crout decomposition, where the upper triangular matrix U has unit diagonal elements, and the lower triangular matrix L can have non-unit diagonal elements.

Crout and Doolittle methods are particularly useful in numerical methods for solving systems of linear equations, which frequently arise in the numerical solution of differential equations using difference schemes. In these schemes, the continuous differential equation is discretized, transforming it into a system of algebraic linear equations. The resulting coefficient matrix, often tridiagonal, can be efficiently decomposed using Crout decomposition. This factorization facilitates the solution of the system through forward and backward substitution (similar to general LU decomposition introduced earlier in the chapter).

Example 8:

Consider the matrix A:

$$A = \begin{pmatrix} 2 & -1 & 0 & 0 \\ 1 & 4 & 1 & 0 \\ 0 & 1 & 3 & -2 \\ 0 & 0 & 2 & -3 \end{pmatrix}$$

We want to find matrices L and U such that $A = LU$, where L is a lower triangular matrix and U is an upper triangular matrix with unit diagonal elements.

The resulting matrices L and U are:

$$L = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 1 & \frac{9}{2} & 0 & 0 \\ 0 & 1 & \frac{25}{9} & 0 \\ 0 & 0 & 2 & -\frac{39}{25} \end{pmatrix}$$

$$U = \begin{pmatrix} 1 & -\frac{1}{2} & 0 & 0 \\ 0 & 1 & \frac{2}{9} & 0 \\ 0 & 0 & 1 & -\frac{18}{25} \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

1.6.5 Singular Value Decomposition (SVD)

The SVD of a matrix $\mathbf{A}$ is:

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T$$

where:

- $\mathbf{U}$ is an $m \times m$ orthogonal matrix (consists of left singular vectors of $\mathbf{A}$).
- $\boldsymbol{\Sigma}$ is an $m \times n$ diagonal matrix with non-negative real numbers on the diagonal, known as the singular values of $\mathbf{A}$.
- $\mathbf{V}^T$ is an $n \times n$ orthogonal matrix (consists of right singular vectors of $\mathbf{A}$).

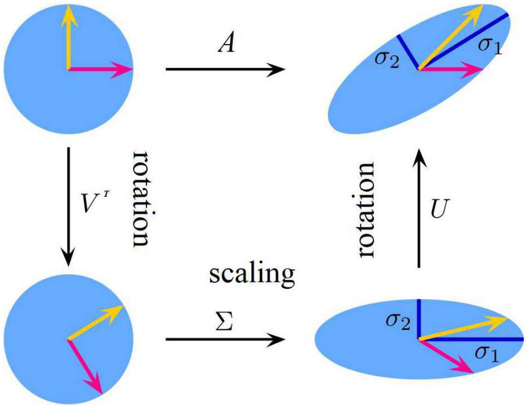


Fig. 1.1: Visual representation of singular value decomposition in 2D. It shows the deformation of the unit disc using action $\mathbf{A}$ and marking two canonical vectors (top). The action is decomposed into two rotations represented by $\mathbf{U}$ and $\mathbf{V}^T$, and scaling represented by $\boldsymbol{\Sigma}$. The image was adapted from [16].

Singular Values and Vectors - definition

A non-negative real number σ is a singular value for $\mathbf{A} \in \mathbb{R}^{m \times n}$ if and only if there exist unit-length vectors $\mathbf{u} \in \mathbb{R}^m$ and $\mathbf{v} \in \mathbb{R}^n$ such that:

$$\mathbf{A}\mathbf{v} = \sigma\mathbf{u}, \quad \mathbf{A}^T\mathbf{u} = \sigma\mathbf{v}$$

where $\mathbf{u}$ and $\mathbf{v}$ are called the left-singular and right-singular vectors for σ, respectively.

Singular Values - computation

Given a matrix $\mathbf{A}$ of size $m \times n$, the singular values of $\mathbf{A}$ are non-negative real values $\sigma_1, \sigma_2, \ldots, \sigma_r$, where r is the rank of the matrix. The singular values are defined as follows:

1. If $\mathbf{A}$ is a square matrix ($m = n$), the singular values are the positive square roots of the eigenvalues of the matrix $\mathbf{A}^T\mathbf{A}$.

2. If $\mathbf{A}$ is a rectangular matrix ($m \neq n$), the singular values are the positive square roots of the nonzero eigenvalues of both $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$.

Singular Vectors - computation

1. **Left Singular Vectors ($\mathbf{u}_i$)**: The left singular vectors are the eigenvectors of the matrix $\mathbf{A}\mathbf{A}^T$ corresponding to the non-zero eigenvalues.
2. **Right Singular Vectors ($\mathbf{v}_i$)**: The right singular vectors are the eigenvectors of the matrix $\mathbf{A}^T\mathbf{A}$ corresponding to the non-zero eigenvalues.

Example 9: Using SVD for Data Compression

1. **Perform SVD**: Given a data matrix $\mathbf{A}$ of size $m \times n$, compute the SVD: $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T$.
2. **Truncate Singular Values**: Retain only the first k singular values and their corresponding columns in $\mathbf{U}$ and rows in $\mathbf{V}$.
3. **Reconstruct Approximate Data**: Create an approximate matrix $\mathbf{A}'$ by using the truncated versions of $\mathbf{U}$, Σ, and $\mathbf{V}$: $\mathbf{A}' = \mathbf{U}_k\Sigma_k(\mathbf{V}_k)^T$.
4. **Reduced Storage**: The reduced matrix $\mathbf{A}'$ requires less storage and represents a compressed version of the original data.

As an example of this approach, assume you have 25 images of a human face with different facial expressions (with a resolution of 200x300 pixels), see Figure 1.2. Assume that your observations are observations of the overall images of a pixel (in total 60 000 observations). Your variables are different images in witch you are looking at the given pixel (in total 25 variables). Since the images are similar, pictures of the same face at approximately the same position), it makes sense to reduce number of observed variables to capture the crucial information in those images. For this purposes we may use SVD for Data Compression.

SVD and Principal Component Analysis (PCA) are closely related techniques used in data analysis and dimensionality reduction. PCA seeks to find the directions (principal components) that maximize the variance in a dataset. This is achieved by diagonalizing the covariance matrix of the data. SVD, on the other hand, decomposes the original data matrix into three matrices: the left singular vectors, the singular values, and the right singular vectors. When the data matrix is centered (i.e., the mean of each column is subtracted), the right singular vectors of SVD correspond to the principal components of PCA, and the singular values relate to the explained variance by each principal component. Thus, performing PCA on a dataset can be efficiently achieved using SVD, making it a powerful tool for extracting the most significant features from the data.

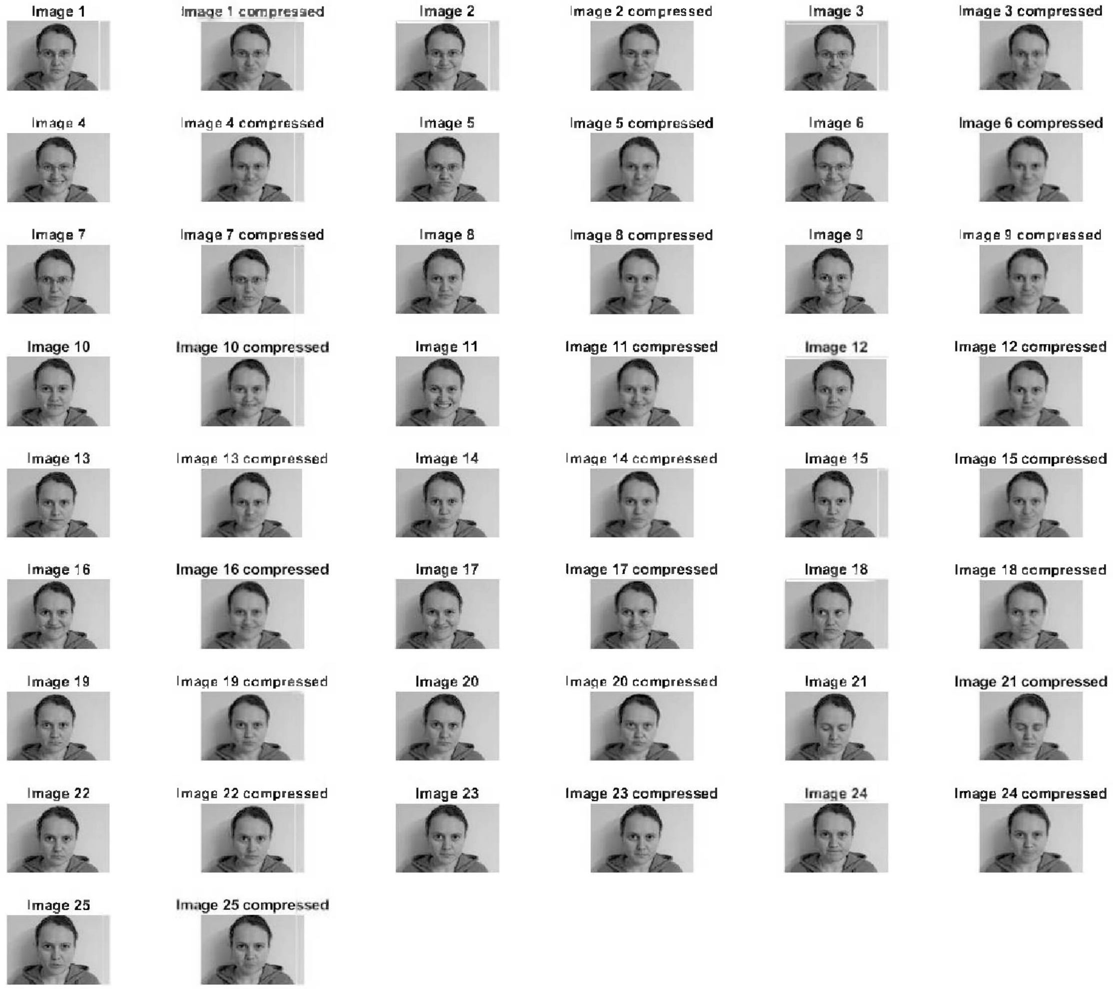


Fig. 1.2: 25 images of a human face with different facial expressions compressed using SVD decomposition.

Example 10: Iterative Closest Point

The Iterative Closest Point (ICP) algorithm is a popular method for aligning point clouds, which are collections of data points in a 3D space. ICP is widely used in fields such as robotics, computer vision, and computer graphics. Its goal is to find the optimal transformation (rotation and translation) that minimizes the distance between two sets of points.

How ICP Works

1. **Initialization**: Start with two point clouds: the source (or model) point cloud and the target (or reference) point cloud. The source point cloud is the one you want to align with the target point cloud.

2. **Matching**: For each point in the source point cloud, find the closest point in the target point cloud. This step pairs each point in the source cloud with its nearest neighbor in the target cloud.

3. **Transformation Estimation**: Compute the transformation (rotation and translation) that best aligns the matched points. This is usually done using methods like Singular Value Decomposition (SVD) to find the optimal transformation that minimizes the mean squared error between the matched points.
4. **Apply Transformation**: Apply the estimated transformation to the source point cloud.
5. **Repeat**: Repeat the matching, estimation, and application steps iteratively until convergence. Convergence is typically defined by a small change in the error metric or a fixed number of iterations.

Details of ICP implementation

Here we provide a potential implementation of the ICP algorithm for point sets $A, B \in \mathbb{R}^2$. This algorithm can be similarly extended to handle higher-dimensional data.

Step 1: Find Closest Points

For each point $\mathbf{a}_i$ in the source point set A, the algorithm finds the closest point $\mathbf{b}_i$ in the target point set B. This is done by calculating the Euclidean distance between each source point and all target points. The Euclidean distance between two points $\mathbf{a}_i = (a_{i1}, a_{i2})$ and $\mathbf{b}_j = (b_{j1}, b_{j2})$ is given by:

$$d(\mathbf{a}_i, \mathbf{b}_j) = \sqrt{(a_{i1} - b_{j1})^2 + (a_{i2} - b_{j2})^2}$$

The algorithm finds the point $\mathbf{b}_i$ in B that minimizes this distance for each $\mathbf{a}_i$ in A.

Step 2: Compute Centroids

The centroid of a point set is the average of all points in the set. For the source point set $A = \{\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_N\}$, the centroid $\mathbf{c}_A$ is calculated as:

$$\mathbf{c}_A = \frac{1}{N} \sum_{i=1}^{N} \mathbf{a}_i$$

Similarly, the centroid $\mathbf{c}_B$ of the corresponding points in the target set B is given by:

$$\mathbf{c}_B = \frac{1}{N} \sum_{i=1}^{N} \mathbf{b}_i$$

Step 3: Center the Points

Once the centroids $\mathbf{c}_A$ and $\mathbf{c}_B$ are computed, both the source and target points are translated to have their centroids at the origin. This is done by subtracting the centroids from each point in the respective sets:

$$\mathbf{a}_i' = \mathbf{a}_i - \mathbf{c}_A$$

$$\mathbf{b}_i' = \mathbf{b}_i - \mathbf{c}_B$$

These centered point sets A' and B' are then used for calculating the optimal rotation.

Step 4: Find the Optimal Rotation (SVD)

To find the optimal rotation matrix R, we first compute the covariance matrix H, which represents the relationship between the centered source points and the centered target points:

$$H = \sum_{i=1}^{N} \mathbf{a}_i'^T \mathbf{b}_i'$$

The next step is to perform Singular Value Decomposition (SVD) on the covariance matrix H. SVD decomposes H into three matrices U, Σ, and V such that:

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

The optimal rotation matrix R_{optimal} is then calculated as:

$$R_{\text{optimal}} = VU^T$$

To ensure that the rotation is proper (i.e., no reflection), if the determinant of R_{optimal} is negative, we adjust the sign of the last column of V before recomputing the rotation matrix.

Step 5: Compute the Optimal Translation

Once the optimal rotation matrix R_{optimal} is found, the translation vector t_{optimal} is calculated by aligning the centroids of the source and target point sets:

$$t_{\text{optimal}} = \mathbf{c}_B - R_{\text{optimal}}\mathbf{c}_A$$

Step 6: Apply the Transformation

The computed rotation R_{optimal} and translation t_{optimal} are applied to the source point set. Each point $\mathbf{a}_i$ in the source is transformed as:

$$\mathbf{a}_i^{\text{new}} = R_{\text{optimal}}\mathbf{a}_i + t_{\text{optimal}}$$

This updates the source point set to align more closely with the target.

Step 7: Check for Convergence

After applying the transformation, the algorithm computes the mean squared error (MSE) between the transformed source points and their corresponding closest points in the target set:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N} \|\mathbf{a}_i^{\text{new}} - \mathbf{b}_i\|^2$$

If the change in error between iterations is smaller than a predefined tolerance, the algorithm concludes that the transformation has converged and stops. Otherwise, the process repeats in the next iteration.

1.7 Least Squares Method

Theoretical Background

Consider the system of linear equations represented by $A \cdot x = b$. Often, this system is unsolvable in the exact sense. For any given vector x, we define the residue r_x as $r_x = b - A \cdot x$. The vector $\hat{x}$ is termed the solution in the sense of least squares if it satisfies $\|r_{\hat{x}}\| \leq \|r_x\|$ for any x.

To understand the solution better, we introduce some concepts related to the matrix A. The range space of the matrix A is denoted by $\mathcal{R}(A)$, and its orthogonal complement is $\mathcal{R}^{\perp}(A)$. Any vector b can be decomposed into $b = b_1 + b_2$, where $b_1 \in \mathcal{R}(A)$ and $b_2 \in \mathcal{R}^{\perp}(A)$. It follows that $A^T \cdot b_2 = \vec{0}$. The vector $\hat{x}$ that solves the least squares problem is then the solution of the system $A \cdot x = b_1$.

Normal Equations

To find the least squares solution, we start with the equation $A \cdot \hat{x} = b_1$. By multiplying both sides of this equation by A^T, we obtain the normal equations:

$$A^T \cdot A \cdot \hat{x} = A^T \cdot b_1.$$

$$A^T \cdot b_1 = A^T \cdot b_1 + A^T \cdot b_2 = A^T \cdot b.$$

Thus, $\hat{x}$ is the solution to the system $A^T \cdot A \cdot x = A^T \cdot b$. This solution is unique if the columns of A are linearly independent. In such a case, the solution is given by:

$$\hat{x} = (A^T A)^{-1} A^T b.$$

The least squares solution can also be expressed using the pseudoinverse. The pseudoinverse A^+ of A provides a generalization of the inverse for non-square or singular matrices. The pseudoinverse is defined as:

$$A^+ = (A^T A)^{-1} A^T$$

Thus, the least squares solution can be written as:

$$\hat{x} = A^{+} \cdot b$$

This approach works for any matrix A, whether it is square or rectangular, and provides the optimal solution in the least squares sense.

Function Approximation

The least squares method is also applicable in function approximation. Suppose we are given points $x_0, \ldots, x_n$ and corresponding function values $f_0, \ldots, f_n$. Additionally, assume we have a set of base functions $\Phi_0, \ldots, \Phi_m$. Our goal is to approximate the function using a linear combination of these base functions:

$$\Phi(x) = c_0\Phi_0(x) + \cdots + c_m\Phi_m(x).$$

We aim to find the parameters $\hat{c}_0, \ldots, \hat{c}_m$ that minimize the sum of squared differences between the given function values and the approximating function at the points x_k:

$$\sum_{k=0}^{n} \left[\Phi(x_k) - f_k\right]^2 .$$

To find these parameters in the sense of least squares, we set up the following system of equations:

$$\begin{aligned} c_0\Phi_0(x_0) + c_1\Phi_1(x_0) + \cdots + c_m\Phi_m(x_0) &= f_0, \\ c_0\Phi_0(x_1) + c_1\Phi_1(x_1) + \cdots + c_m\Phi_m(x_1) &= f_1, \\ &\vdots \\ c_0\Phi_0(x_n) + c_1\Phi_1(x_n) + \cdots + c_m\Phi_m(x_n) &= f_n. \end{aligned}$$

We can represent this system in matrix form as $A \cdot c = f$, where A is the matrix of the base functions evaluated at the given points, and f is the vector of function values:

$$A = \begin{pmatrix} \Phi_0(x_0) & \Phi_1(x_0) & \cdots & \Phi_m(x_0) \\ \Phi_0(x_1) & \Phi_1(x_1) & \cdots & \Phi_m(x_1) \\ \vdots & & & \vdots \\ \Phi_0(x_n) & \Phi_1(x_n) & \cdots & \Phi_m(x_n) \end{pmatrix}, \quad f = \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{pmatrix}.$$

The parameters $\hat{c} = (\hat{c}_0, \ldots, \hat{c}_m)^T$ are then given by the normal equations:

$$A^T \cdot A \cdot c = A^T \cdot f,$$

which simplifies to:

$$\hat{c} = \left(A^T \cdot A\right)^{-1} A^T \cdot f.$$

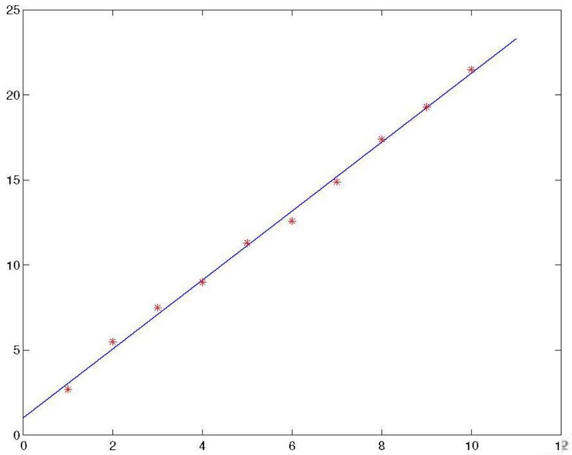


Fig. 1.3: Approximation of data by linear function using least square method.

Example 11: Linear Function Approximation

Consider an example where we have the following data points:

x_i	1	2	3	4	5	6	7	8	9	10
f_i	2.7	5.5	7.5	9.0	11.3	12.6	14.9	17.4	19.3	21.5

We seek a linear function to approximate this data. Using the base functions $\Phi_0(x) = 1$ and $\Phi_1(x) = x$, we construct the matrix A and the vector f:

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \\ 1 & 6 \\ 1 & 7 \\ 1 & 8 \\ 1 & 9 \\ 1 & 10 \end{pmatrix}, \quad f = \begin{pmatrix} 2.7 \\ 5.5 \\ 7.5 \\ 9.0 \\ 11.3 \\ 12.6 \\ 14.9 \\ 17.4 \\ 19.3 \\ 21.5 \end{pmatrix}.$$

Solving the normal equations, we find:

$$\hat{c} = \left(A^T \cdot A\right)^{-1} A^T \cdot f \approx \begin{pmatrix} 1.0267 \\ 2.0261 \end{pmatrix},$$

yielding the approximating function:

$$\Phi(x) = 1.0267 + 2.0261x.$$

Example 12: Model Height of Girls

The following table contains measurements of the height of students from a local primary, secondary, and high school, ranging in age from 5 to 18 years. Assume that the height increases linearly for girls between the ages of 5 and 12. For girls older than 12, the height remains constant. Use the least-squares method to estimate the exact dependence for the given data. What height does your model predict for:

(a) A 12-year-old girl?

(b) A 5-year-old girl?

Age (years)	15.5	17.8	16.5	7.8	5.5	9.0	6.2	12.0	10.5	8.3	14.0	13.5	18.0
Height (cm)	170	168	167	134	114	139	121	154	144	130	160	159	169

Table 1.1: Height and Age Data of Girls (sample of data), full data are stored in file `heigth_of_girls.txt`

Solution

We begin by extracting the relevant information from the table, namely the age X and height Y of the girls. We then use a piecewise linear model where the height increases linearly until the age of 12 and remains constant thereafter.

To construct the model, we define two functions:

$$f(x) = (x < 12) \cdot (x - 12)$$

representing the linear increase for ages below 12, and

$$g(x) = 1$$

representing the constant height for ages above 12.

Using these functions, we form a design matrix A and apply the least squares method to find the coefficients c that best fit the data. The resulting height function is:

$$h(x) = c_1 \cdot f(x) + c_2 \cdot g(x)$$

After performing the calculations, the model predicts the following heights:

- **For a 12-year-old girl:** The height is approximately **165.5820 cm**.
- **For a 5-year-old girl:** The height is approximately **107.6141 cm**.

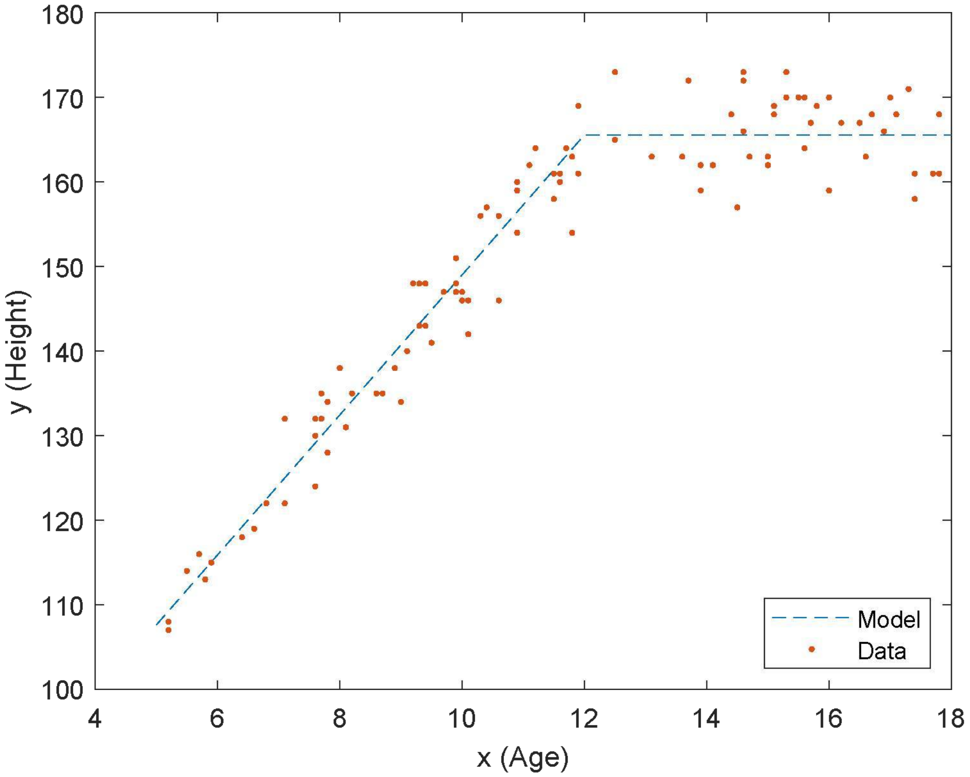


Fig. 1.4: Model of Height vs Age of Girls using least-square method.

Chapter 2

Interpolation

2.1 Recommended literature

[7] Rainer Kress. *Numerical analysis*. Vol. 181. Springer Science & Business Media, 2012
[13] L Richard. "Burden and J. Douglas Faires. Numerical analysis. Brooks". In: *Cole Publishing Company, Pacific Grove, California* 93950 (1997), p. 78

2.2 Introduction to Interpolation

Interpolation is a method used to estimate values of unknown function that fall within the range of a discrete set of known data points. It is a fundamental technique in numerical analysis, data fitting, and numerical modeling. The goal of interpolation is to construct new data points within the range of known data points.

Mathematically speaking, given points (knots) $x_0, \ldots, x_n$, where $x_i \neq x_j$ for $i \neq j$, and given function values (measurements) $f_0, \ldots, f_n$, where $f_i = f(x_i)$, we consider the function $\Phi(x) = a_0\Phi_0(x) + \cdots + a_n\Phi_n(x)$ that depends on the parameters $a_0, \ldots, a_n$. The problem of interpolation is to find the parameters $a_0, \ldots, a_n$ such that the conditions

$$\Phi(x_i) = f_i, \quad \text{for} \quad i = 0, 1, \ldots, n,$$

are fulfilled.

Examples include:

$$\Phi(x) = a_0 + a_1 x + \cdots + a_n x^n, \quad \text{a polynomial,}$$

and

$$\Phi(x) = a_0 + a_1 e^{ix} + \cdots + a_n e^{inx}, \quad \text{a trigonometric polynomial.}$$

2.3 Polynomial Interpolation

Polynomial interpolation involves approximating a given set of data points with a polynomial function. Given a set of $n + 1$ data points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$, polynomial interpolation finds a polynomial $P(x)$ of degree at most n such that $P(x_i) = y_i$ for all $i = 0, 1, \ldots, n$.

For given points $(x_i, f_i), i = 0, \ldots, n$, where $x_i \neq x_j$ for $i \neq j$, there exists a unique polynomial P of degree at most n such that

$$P(x_i) = f_i, \quad i = 0, \ldots, n.$$

Uniqueness: If $P_1(x_i) = P_2(x_i) = f_i, \quad i = 0, \ldots, n$, then $Q = P_1 - P_2$ is a polynomial of degree at most n and $Q(x_i) = 0, \quad i = 0, \ldots, n$, i.e., Q has $n+1$ roots so Q must be the zero polynomial.

Existence: To construct P, we construct the polynomials L_i:

- L_i is a polynomial of degree n,
- $L_i(x_j) = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j. \end{cases}$

Points $x_j, j \neq i$ are roots of L_i:

$$L_i(x) = A_i(x - x_0) \cdots (x - x_{i-1})(x - x_{i+1}) \cdots (x - x_n),$$

or

$$L_i(x) = A_i \pi_i(x), \quad \text{where} \quad \pi_i(x) = \prod_{j \neq i} (x - x_j).$$

Since $L_i(x_i) = 1$, we have

$$A_i = \frac{1}{\pi_i(x_i)}.$$

2.3.1 Lagrange Interpolation

The construction described in the previous paragraph is called the Lagrange interpolation. To recapitulate, the Lagrange interpolation polynomial is constructed as follows:

Define the Lagrange Basis Polynomials:

$$L_i(x) = \prod_{\substack{0 \leq j \leq n \\ j \neq i}} \frac{x - x_j}{x_i - x_j}$$

Each $L_i(x)$ is a polynomial of degree n that is equal to 1 at x_i and 0 at x_j for $j \neq i$.

Construct the Interpolating Polynomial:

$$P(x) = \sum_{i=0}^{n} y_i L_i(x)$$

This polynomial $P(x)$ is a linear combination of the basis polynomials $L_i(x)$ weighted by the data values y_i.

Effective Calculation of L_i

The calculation of one base polynomial L_i is $O(n^2)$, meaning that the direct calculation of the interpolation polynomial is $O(n^3)$.

To improve efficiency of the algorithm we may use Horner's scheme. It allows us to compute P in $O(n^2)$. The computation consists of the following steps:

1. Compute $\omega(x) = \prod_{j=0}^{n} (x - x_j)$, which is $O(n^2)$,
2. Compute $\pi_i(x) = \omega(x)/(x - x_i)$ using Horner's scheme, which is $O(n)$,
3. Compute $\pi(x_i)$ using Horner's scheme, which is $O(n)$,
4. Finally, compute the polynomial P in $O(n^2)$.

The Lagrange construction is useful when you need to create several interpolation polynomials for the same values of the independent variable x but different values of the dependent variable y. An example of such a scenario is measuring a human characteristic y during regular medical examinations (the timing of the examinations is given by the independent variable x) and finding the continuous dependence of such a characteristic using interpolation. The timing of the regular examinations is fixed, so the independent variable x remains the same for different patients, but the measured values y vary for each patient.

Example:

Construct Lagrange interpolation polynomial for the given data:

x_i	-1	0	1	3
f_i	-3	1	-1	1

Solution using direct construction:

$$\begin{aligned}
L_0(x) &= \frac{(x-0)(x-1)(x-3)}{(-1-0)(-1-1)(-1-3)} = -\frac{1}{8}x^3 + \frac{1}{2}x^2 - \frac{3}{8}x \\
L_1(x) &= \frac{(x+1)(x-1)(x-3)}{(0+1)(0-1)(0-3)} = \frac{1}{3}x^3 - x^2 - \frac{1}{3}x + 1 \\
L_2(x) &= \frac{(x+1)(x-0)(x-3)}{(1+1)(1-0)(1-3)} = -\frac{1}{4}x^3 + \frac{1}{2}x^2 + \frac{3}{4}x \\
L_3(x) &= \frac{(x+1)(x-0)(x-1)}{(3+1)(3-0)(3-1)} = \frac{1}{24}x^3 - \frac{1}{24}x \\
P(x) &= -3L_0(x) + L_1(x) - L_2(x) + L_3(x) = x^3 - 3x^2 + 1
\end{aligned}$$

Solution using Horner scheme:

$$\begin{aligned}
\omega(x) &= (x+1)(x-0)(x-1)(x-3) = x^4 - 3x^3 - x^2 + 3x \\
\pi_0(x) &= \omega(x) : (x+1) \\
\pi_1(x) &= \omega(x) : (x-0) \\
\pi_2(x) &= \omega(x) : (x-1) \\
\pi_3(x) &= \omega(x) : (x-3)
\end{aligned}$$

Using the Horner scheme for division $\omega(x) : (x+1)$ we construct $L_0(x)$, and other base polynomials can be computed similarly.

$$\begin{array}{c|ccccc}
\omega & 1 & -3 & -1 & 3 & 0 \\
\hline
-1 & \boxed{1} & \boxed{-4} & \boxed{3} & \boxed{0} & 0
\end{array}
\quad \Rightarrow \quad \pi_0(x) = x^3 - 4x^2 + 3x$$

Using Horner scheme we can evaluate $\pi_0(x_0) = \pi_0(-1)$: $\begin{array}{c|cccc} \pi(x) & 1 & -4 & 3 & 0 \\ \hline -1 & 1 & -5 & 8 & \boxed{-8} \end{array}$ $\Rightarrow$ $\pi_0(-1) = -8$

Finally, $L_0(x) = \frac{\pi_0(x)}{\pi_0(x_0)} = -\frac{1}{8}(x^3 - 4x^2 + 3x)$.

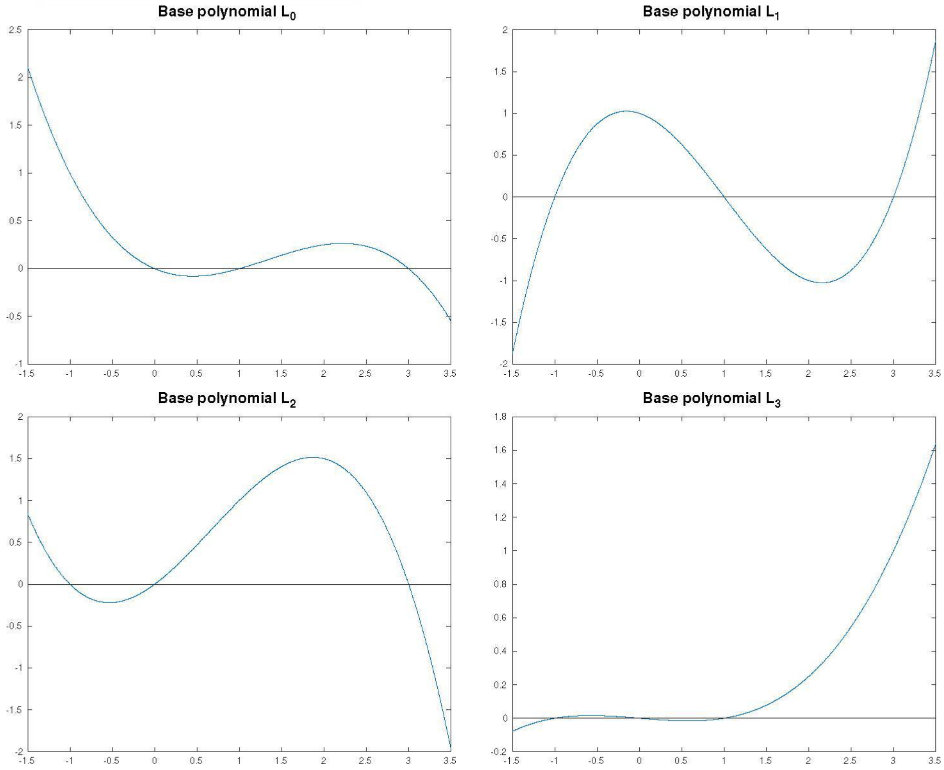


Fig. 2.1: Lagrange base polynomials

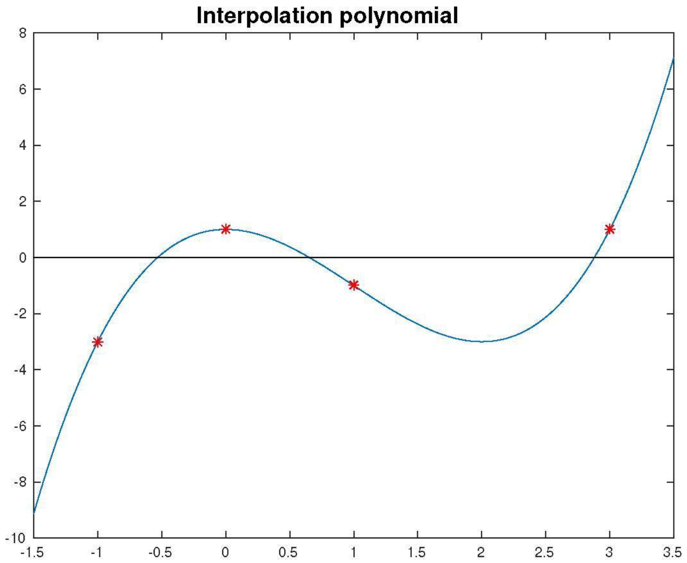


Fig. 2.2: Lagrange interpolation polynomial

2.3.2 Newton Interpolation

The Newton interpolation polynomial is constructed using divided differences. The steps are as follows:

Calculate the Divided Differences:

The first divided difference is:

$$f[x_i] = y_i$$

The second divided difference is:

$$f[x_i, x_j] = \frac{f[x_j] - f[x_i]}{x_j - x_i}$$

The general k-th divided difference is:

$$f[x_i, x_{i+1}, \ldots, x_{i+k}] = \frac{f[x_{i+1}, \ldots, x_{i+k}] - f[x_i, \ldots, x_{i+k-1}]}{x_{i+k} - x_i}$$

Construct the Newton Polynomial:

The Newton interpolation polynomial is:

$$P(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \cdots + a_n(x - x_0)(x - x_1) \cdots (x - x_{n-1})$$

where the coefficients a_i are the divided differences:

$$a_i = f[x_0, x_1, \ldots, x_i]$$

Table of divided differences:

The divided differences for Newton interpolation can be computed using a recursive tree structure. This structure systematically computes the coefficients needed for the Newton polynomial. Here's a detailed step-by-step process for constructing the divided differences tree.

x_i	f_i		$f[x_i, x_{i+1}]$		$f[x_i, x_{i+1}, x_{i+2}]$		...		
x_0	$\underline{f_0}$	$>$	$\underline{f[x_0, x_1]}$	$>$	$\underline{f[x_0, x_1, x_2]}$	$>$			
x_1	f_1	$>$	$f[x_1, x_2]$				$\cdots$	$>$	$\underline{f[x_0, \ldots, x_n]}$
x_2	f_2		$\vdots$		$\vdots$				
$\vdots$	$\vdots$	$>$	$f[x_{n-1}, x_n]$	$>$	$f[x_{n-2}, x_{n-1}, x_n]$	$>$			
x_n	f_n								

The first column contains the y-values of the data points. Each subsequent column in the tree is computed from the differences of the previous column, divided by the difference in the corresponding x-values. The tree is built level by level, starting from the first column (the y-values).

To proof $P(x_i) = f_i$, we start with the original polynomial:

$$\begin{aligned} P(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + \\ + \cdots + f[x_0, \ldots, x_n](x - x_0) \cdots (x - x_{n-1}) \end{aligned}$$

Now, we substitute $x = x_0$:

$$P(x_0) = f[x_0] = f_0$$

Now, we substitute $x = x_1$:

$$P(x_1) = f[x_0] + f[x_0, x_1](x_1 - x_0) = f_0 + \frac{f_1 - f_0}{x_1 - x_0}(x_1 - x_0) = f_1$$

Now, we substitute $x = x_2$:

$$\begin{aligned} P(x_2) &= f[x_0] + f[x_0, x_1](x_2 - x_0) + f[x_0, x_1, x_2](x_2 - x_0)(x_2 - x_1) = \\ &= f[x_0] + f[x_0, x_1](x_2 - x_0) + \frac{f[x_2, x_1] - f[x_0, x_1]}{x_2 - x_0}(x_2 - x_0)(x_2 - x_1) = \\ &= f[x_0] + f[x_0, x_1]x_2 - f[x_0, x_1]x_0 + f[x_1, x_2]x_2 + f[x_0, x_1]x_1 - f[x_1, x_2]x_1 - \\ &\quad - f[x_0, x_1]x_2 = P(x_1) + f[x_1, x_2](x_2 - x_1) = f_1 + \frac{f_2 - f_1}{x_2 - x_1}(x_2 - x_1) = f_2 \end{aligned}$$

Similarly, we can proceed with an arbitrary x_i.

The Newton construction is useful in scenarios where a new data point (x_{n+1}, y_{n+1}) is added (thus giving you $n + 2$ points) and you have already computed the interpolation for the original set of $n + 1$ points. This is particularly beneficial in situations where you need to increase the precision of the function estimation.

Example:

Construct Newton interpolation polynomial for the given data:

x_i	1	2	3	4
f_i	1	4	9	16

First Column:

x	$f[x]$
1	1
2	4
3	9
4	16

Second Column:

x	$f[x]$	$f[x_0, x_1]$
1	1	$\frac{4-1}{2-1} = 3$
2	4	$\frac{9-4}{3-2} = 5$
3	9	$\frac{16-9}{4-3} = 7$
4	16	

Third Column:

x	$f[x]$	$f[x_0, x_1]$	$f[x_0, x_1, x_2]$
1	1	3	$\frac{5-3}{3-1} = 1$
2	4	5	$\frac{7-5}{4-2} = 1$
3	9	7	
4	16		

Fourth Column:

x	$f[x]$	$f[x_0, x_1]$	$f[x_0, x_1, x_2]$	$f[x_0, x_1, x_2, x_3]$
1	1	3	1	$\frac{1-1}{4-1} = 0$
2	4	5	1	
3	9	7		
4	16			

So, the divided differences are:

$$\begin{aligned} f[x_0] &= 1 \\ f[x_0, x_1] &= 3 \\ f[x_0, x_1, x_2] &= 1 \\ f[x_0, x_1, x_2, x_3] &= 0 \end{aligned}$$

The Newton polynomial is:

$$\begin{aligned} P(x) &= 1 + 3(x-1) + 1(x-1)(x-2) + 0(x-1)(x-2)(x-3) \\ P(x) &= 1 + 3(x-1) + (x-1)(x-2) \\ P(x) &= 1 + 3x - 3 + x^2 - 3x + 2 \\ P(x) &= x^2 \end{aligned}$$

2.4 Error of Polynomial Interpolation

The error of polynomial interpolation for a function $f(x)$ is given by the interpolation error formula:

$$f(x) - P(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^{n} (x - x_i)$$

for some ξ in the interval $[a, b]$. The error depends on the distribution of interpolation points x_i.

2.4.1 Selecting Points to Minimize Error

To minimize interpolation error in the L_2 norm for a general continuous function, Chebyshev nodes are often used. Chebyshev nodes are given by:

$$x_i = \cos\left(\frac{2i+1}{2n+2}\pi\right), \quad i = 0, 1, \ldots, n$$

Benefits of Chebyshev Nodes

1. **Minimize the Maximum Error:** The distribution of Chebyshev nodes reduces the maximum value of the product

$$\prod_{i=0}^{n}(x - x_i)$$

over the interval $[-1, 1]$. This product, known as the interpolation polynomial's Lebesgue function, tends to have smaller maximum values when Chebyshev nodes are used compared to equidistant nodes. This effectively minimizes the interpolation error, particularly near the boundaries of the interval.

2. **Reduction of Runge Phenomenon:** The Runge phenomenon, which is the large oscillation that occurs when using equally spaced nodes for interpolation of certain functions, is significantly mitigated by the use of Chebyshev nodes. The clustering of Chebyshev nodes near the interval's endpoints reduces these oscillations.

3. **Uniform Distribution of Error:** Chebyshev nodes provide a near-optimal distribution of interpolation points that balances the error across the interval. The error tends to be more uniformly distributed, leading to a more accurate overall interpolation.

4. **Connection to Chebyshev Polynomials:** The Chebyshev polynomials themselves have the property of minimizing the maximum deviation from zero among all polynomials of the same degree with leading coefficient 1. This property translates to the interpolation error when using Chebyshev nodes.

2.5 Definitions of Linear, Bilinear, and Trilinear Interpolation

Linear Interpolation

Linear interpolation is a special case of polynomial interpolation, where the interpolating polynomial is of degree one. It is commonly used for piecewise interpolation, where a linear function is applied between consecutive data points. In this context, linear interpolation can also be viewed as a special case of splines, specifically a linear spline, which will be discussed later in this chapter.

Used for one-dimensional data. It approximates the value $f(x)$ within an interval $[x_0, x_1]$ by:

$$f(x) \approx \frac{(x - x_0)f(x_1) + (x_1 - x)f(x_0)}{x_1 - x_0}$$

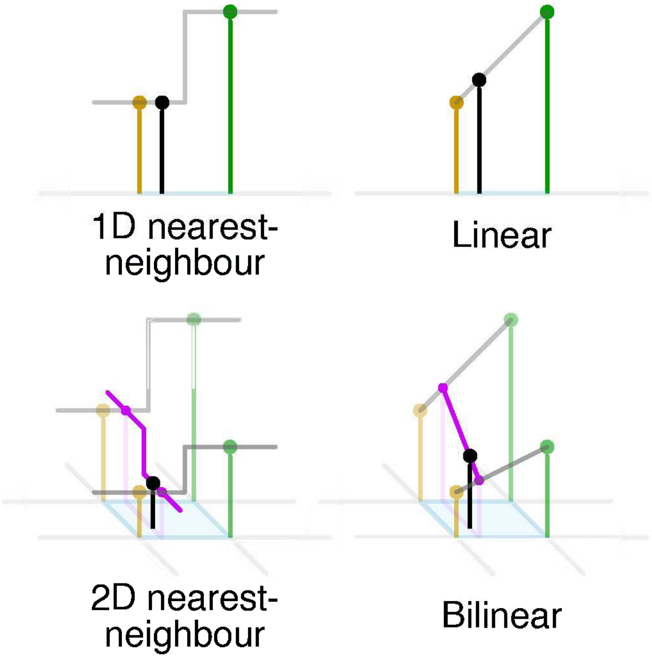


Fig. 2.3: Comparing nearest neighbor and linear interpolation in 1D and 2D. The figure was adapted from [15].

Bilinear Interpolation

Extends linear interpolation to two dimensions. Given points $(x_0, y_0), (x_1, y_1)$, it interpolates a value $f(x, y)$ based on linear interpolations along both dimensions.

$$f(x,y) \approx f(x_0,y_0)\frac{(x_1-x)(y_1-y)}{(x_1-x_0)(y_1-y_0)} + f(x_1,y_0)\frac{(x-x_0)(y_1-y)}{(x_1-x_0)(y_1-y_0)} + f(x_0,y_1)\frac{(x_1-x)(y-y_0)}{(x_1-x_0)(y_1-y_0)} + f(x_1,y_1)\frac{(x-x_0)(y-y_0)}{(x_1-x_0)(y_1-y_0)}$$

Trilinear Interpolation

Extends bilinear interpolation to three dimensions. It involves interpolating within a cube and uses linear interpolation first along one dimension, then another, and finally the third.

$$f(x,y,z) \approx \sum_{i=0}^{1}\sum_{j=0}^{1}\sum_{k=0}^{1} f(x_i,y_j,z_k)\frac{(x_{1-i}-x)(y_{1-j}-y)(z_{1-k}-z)}{(x_1-x_0)(y_1-y_0)(z_1-z_0)}$$

Each interpolation method serves different purposes and has its advantages based on the dimensionality and distribution of the data points.

2.6 Radial Basis Functions Interpolation

Radial basis functions (RBF) are a type of real-valued function whose value depends only on the distance from a central point. In mathematical terms, a radial basis function ϕ is defined as $\phi(\|x - c\|)$, where x is the input vector, c is the center of the function, and $\|\cdot\|$ denotes the Euclidean norm.

Common Radial Basis Functions

Several types of RBFs are commonly used in practice:

- **Gaussian**: $\phi(r) = e^{-\epsilon^2 r^2}$
- **Inverse Multiquadric**: $\phi(r) = \frac{1}{\sqrt{1+(\epsilon r)^2}}$
- **Thin Plate Spline**: $\phi(r) = r^2 \ln(r)$

Here, $r = \|x - c\|$ and ϵ is a shape parameter that can adjust the width of the basis function.

Advantages of RBF interpolation

- RBF interpolation can handle scattered and irregularly spaced data points, making it suitable for complex and non-uniform data distributions.
- Depending on the chosen RBF, the resulting interpolant can have higher-order continuity (e.g., Gaussian RBFs produce infinitely differentiable surfaces).

RBF Interpolation Process

Given a set of data points $\{(x_i, y_i)\}$ where $x_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}$, the goal is to find an interpolant function $s(x)$ such that $s(x_i) = y_i$ for all i. The RBF interpolant is typically expressed as:

$$s(x) = \sum_{i=1}^{N} \lambda_i \phi(\|x - x_i\|)$$

where λ_i are coefficients to be determined, ϕ is the chosen RBF.

Solving for Coefficients

To determine the coefficients λ_i one typically solves a linear system derived from the interpolation conditions. Specifically, the system of equations can be written as:

$$\sum_{j=1}^{N} \lambda_j \phi(\|x_i - x_j\|) = y_i \quad \text{for all } i = 1, \ldots, N$$

This linear system can be represented in matrix form and solved using standard numerical linear algebra techniques.

2.7 Spline Interpolation

Spline interpolation is a method used to construct a smooth function that passes through a given set of points. This technique is particularly useful in numerical analysis for approximating complex functions using piecewise polynomials. The key advantage of spline interpolation over polynomial interpolation is that it avoids the problem of oscillation between points, providing a smoother and more accurate fit.

Given a set of points, known as knots, $x_0, x_1, \ldots, x_n$ with $x_0 < x_1 < \ldots < x_n$, and corresponding function values $f_0, f_1, \ldots, f_n$, spline interpolation aims to find a function $S(x)$ that not only passes through these points but also has certain smoothness properties.

Parameters:

- r (degree): The degree of the piecewise polynomial segments.
- d (defect): The number of continuous derivatives the spline maintains.

A spline S of degree r with defect d is a piecewise polynomial of degree r that has continuous derivatives up to order $r - d$. The space of such splines is denoted by $\mathcal{S}_{r,d}$.

2.7.1 Linear Splines

In section 2.5 we naturally introduced piecewise-linear interpolation as a generalization of polynomial interpolation. Piecewise-linear interpolation also fits a definition of splines. Consider the space $\mathcal{S}_{1,1}$, which consists of piecewise linear continuous functions. A spline S in this space is uniquely determined by the function values $f_0, f_1, \ldots, f_n$. This results in a piecewise linear function that connects each pair of consecutive points with a straight line, ensuring continuity at each knot.

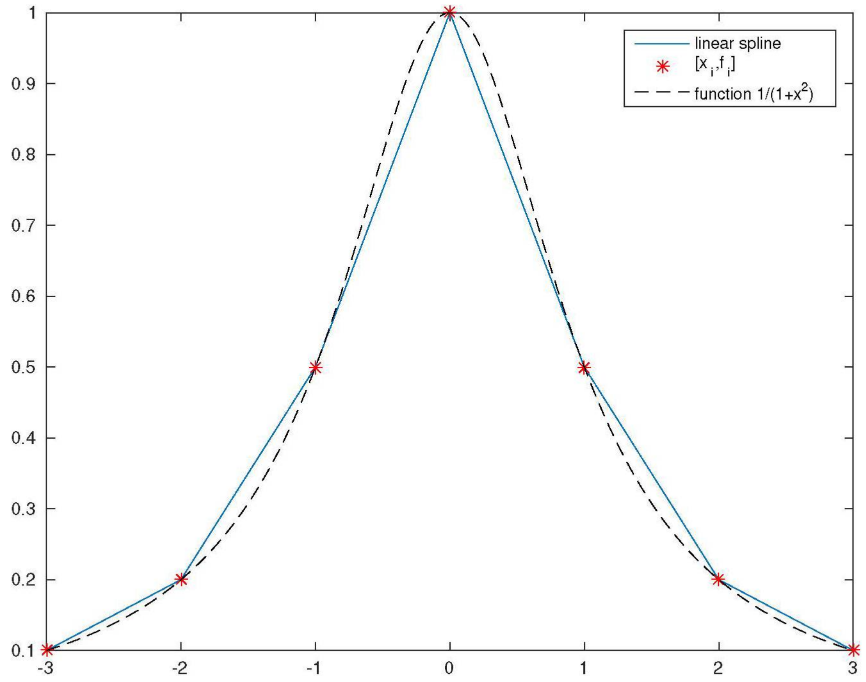


Fig. 2.4: Piecewise linear spline

2.7.2 Cubic Splines

For a more sophisticated approximation, consider $\mathcal{S}_{3,1}$, the space of piecewise cubic polynomials with continuous second derivatives. A cubic spline $S \in \mathcal{S}_{3,1}$ is defined on each subinterval $I_k = [x_k, x_{k+1}]$ by a cubic polynomial:

$$S(x) = S_k(x) = a_k + b_k(x - x_k) + c_k(x - x_k)^2 + d_k(x - x_k)^3$$

Determining the Parameters

The cubic spline across n subintervals requires $4n$ parameters (4 for each subinterval). However, these parameters are subject to several conditions ensuring the spline's smoothness and continuity:

- Continuity at each internal knot $x_1, x_2, \ldots, x_{n-1}$: $n - 1$ conditions.
- Continuity of the first derivative S' at each internal knot: $n - 1$ conditions.
- Continuity of the second derivative S'' at each internal knot: $n - 1$ conditions.
- The spline passing through each given point (x_k, f_k): $n + 1$ conditions.

Thus, the total number of conditions is $4n - 2$.

Boundary Conditions

To obtain a unique cubic spline, additional boundary conditions are required. Common choices include:

1. **Complete Cubic Spline**: Specifying the first derivatives at the endpoints, $S'(x_0)$ and $S'(x_n)$.
2. **Natural Cubic Spline**: Setting the second derivatives at the endpoints to zero, $S''(x_0) = S''(x_n) = 0$.
3. **Not-a-Knot Condition**: Ensuring the third derivative is continuous at the second and the last but one knots.
4. **Periodic Spline**: Requiring the function and its derivatives to be periodic, $S(x_0) = S(x_n)$, $S'(x_0) = S'(x_n)$, and $S''(x_0) = S''(x_n)$.

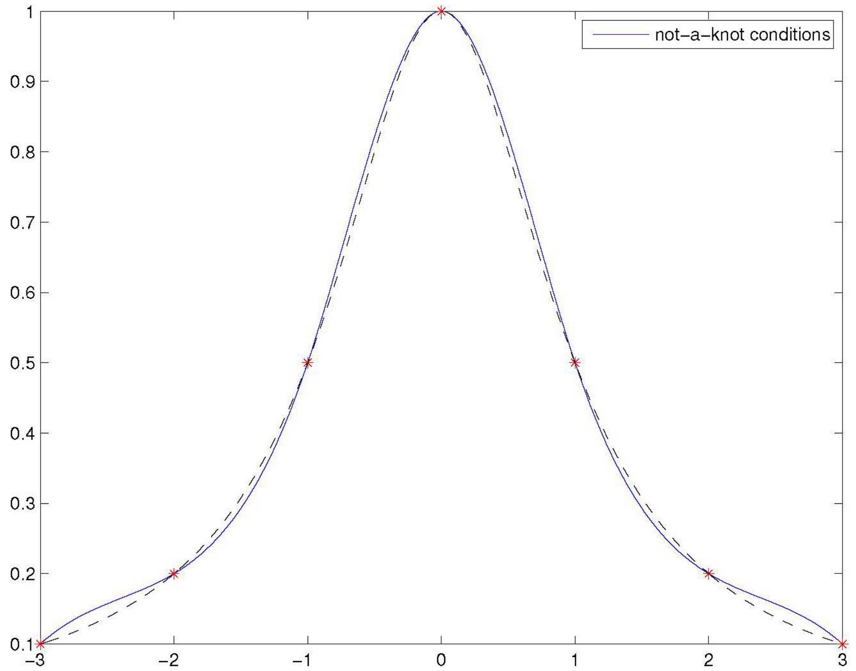


Fig. 2.5: Cubic interpolation spline with not-a-knot bondary conditions

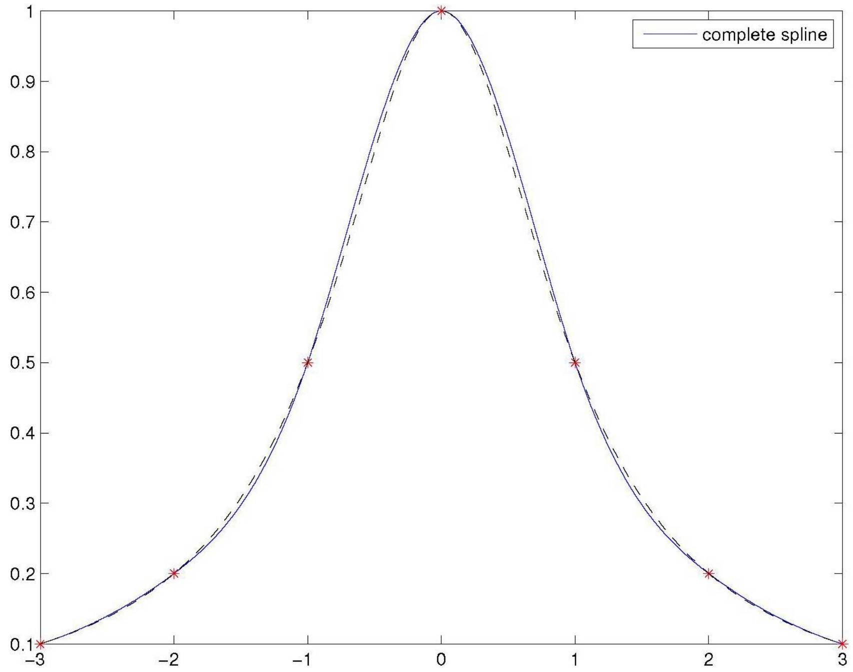


Fig. 2.6: Complete cubic interpolation spline

2.8 Examples

Example 1: Finding an intersection points of a curve with a plane

In this example, we demonstrate how to use interpolation to find the intersection points of a curve with a plane. The original curve is obtained using a numerical ordinary differential equation (ODE) solver. The coordinates of the points that lie on the curve in 3D space are stored in `data.mat`. Our goal is to approximate the part of the curve that intersects with a specified plane.

Procedure

We begin by loading the coordinates x, y, and z from `data.mat` and plotting the 3D curve. To visualize the plane, we create a mesh grid and plot the plane in the 3D space.

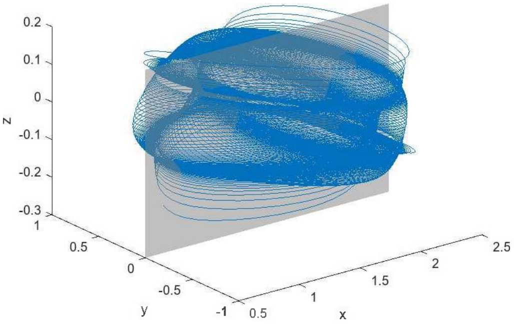
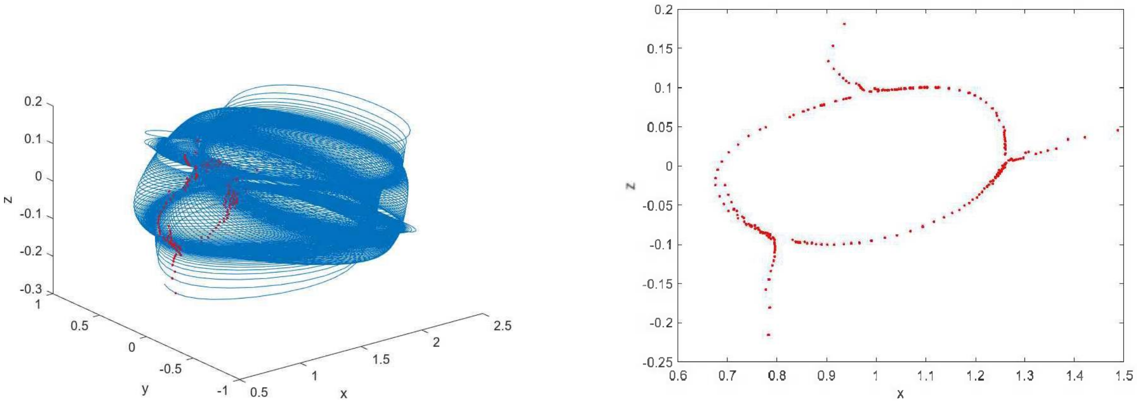


Fig. 2.7: The 3D curve

Next, we identify the points where the curve intersects the plane. Specifically, we look for points where the y-coordinate changes from positive to negative values, indicating an intersection with the plane.



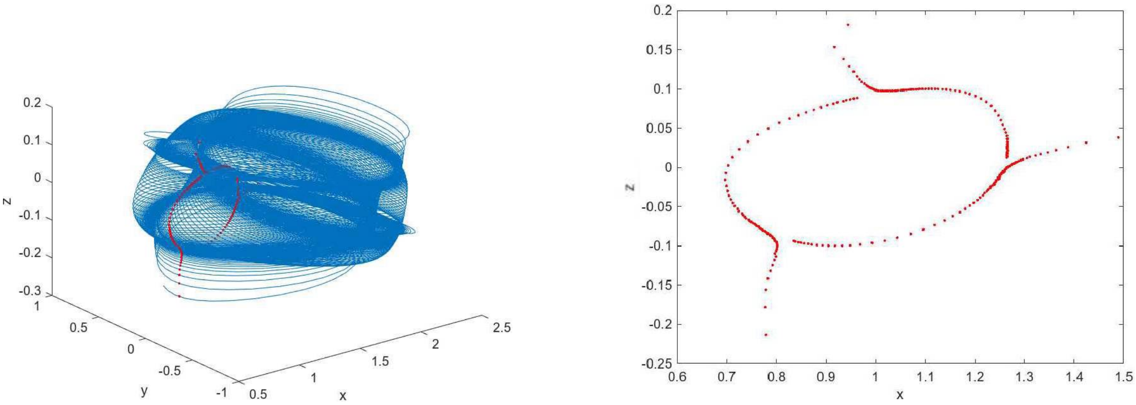
(a) The desirable points visualize on the 3D curve.

(b) The plain visualization of the desirable points.

Fig. 2.8: The 3D curve with identified desirable points.

To improve the accuracy of the intersection points, we perform linear interpolation. For each pair of points around the intersection, we estimate the exact coordinates of the intersection using the formula for linear interpolation.

Finally, we plot the resulting intersection points in the xz-plane to visualize the points of interest.



(a) The plain visualization of the improved desirable points.

(b) The plain visualization of the improved desirable points.

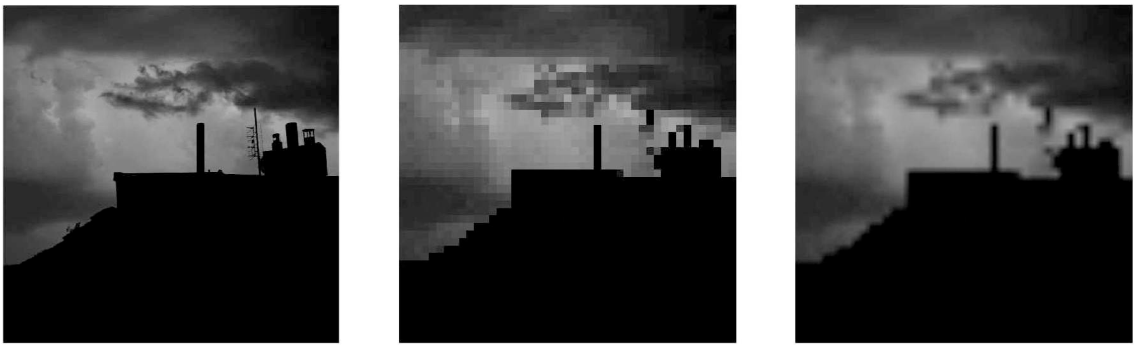
Fig. 2.9: The 3D curve with desirable points improved using linear interpolation.

This example illustrates the use of linear interpolation to find intersection points of a curve with a plane. By accurately estimating these points, we can better understand the behavior of the curve in relation to the plane.

Example 2: Bilinear interpolation to enhance a low-resolution figure

In the field of image processing, enhancing the resolution of images is a critical task, especially when dealing with images that are originally captured or stored at low resolutions. One common technique used for this purpose is bilinear interpolation. This method improves the visual quality of a low-resolution image by estimating the pixel values based on the linear interpolation of the nearest neighboring pixels.

In example, we demonstrate the process of using bilinear interpolation to enhance a low-resolution figure. We then compare the improved image with its high-resolution predecessor to evaluate the effectiveness of the interpolation technique. This comparison provides insights into how well the interpolation method preserves the details and quality of the original high-resolution image.



(a) high resolution

(b) reduced resolution

(c) interpolated figure

Fig. 2.10: Using bilinear interpolation, we enhanced a low-resolution figure and compared it with its high-resolution predecessor.

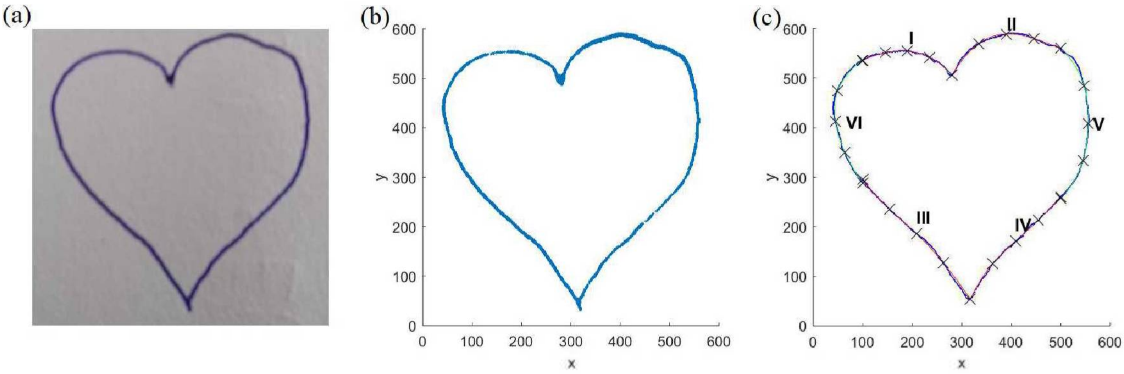


Fig. 2.11: Spline interpolation of implicit curve: (a) the original image, (b) extracted boundary of the heart, (c) interpolated image. In (c) blue dots marks the original boundary point, black crosses the data points selected for the interpolation, Greek numbers six different segments where the interpolation was performed, red line the resulting classical interpolation spline, green line the resulting inverse interpolation spline.

Example 3: Spline interpolation of implicit curve

The goal of this example is to extract an analytic curve that defines the bordering surface of an image from a photograph, see Figure 2.11. By segmenting the image data and applying spline interpolation techniques, we aim to accurately capture the implicit curve that represents the boundary.

Segmenting the data is essential for accurately analyzing an implicit curve within the image. An implicit curve is defined by an equation $f(x, y) = 0$, which means that y is not explicitly given as a function of x. To apply interpolation effectively, the image is divided into regions where the curve can be locally approximated by functions or where the relationship between x and y can be better managed. Segmenting the data is also necessary to manage regions where the curve's smoothness needs to be deliberately disrupted, ensuring that the interpolation reflects the actual characteristics of the border.

Classical Spline Interpolation

In regions where y can be treated as a function of x, classical spline interpolation is used. This technique helps in fitting a smooth curve through the data points. For each segment, unique x-coordinates are identified, and y-values are interpolated at specific knot points. A spline is then fitted through these knots, providing a smooth approximation of the curve.

Inverse Spline Interpolation

In regions where x can be treated as a function of y, inverse spline interpolation is applied. This is useful for segments where the implicit curve has a steep slope or vertical segments. Unique y-coordinates are identified, and x-values are interpolated at specific knot points. A spline is fitted through these knots, offering a smooth representation of the curve.

Example 4: RBF interpolation and surface reconstruction

This example demonstrates the use of Radial Basis Function (RBF) interpolation to reconstruct a smooth surface from scattered data points generated from a normal distribution. The code also compares the RBF interpolation results with those obtained from bilinear interpolation. The scattered data points represent the surface $z = \sin(\pi x) \cdot \cos(\pi y)$.

You should see that the results of the linear interpolation are not smooth as a key difference. Linear interpolation produces piecewise linear surfaces, which means that the interpolated surface is composed of flat triangles (in 2D) or tetrahedra (in higher dimensions) between the known data points. This can result in a surface that is not differentiable at the edges of these elements, leading to a lack of smoothness. In contrast, Radial Basis Function (RBF) interpolation uses smooth, continuous functions to model the surface. RBF interpolation can produce a surface that is infinitely differentiable, depending on the choice of the basis function, such as the Gaussian function. This results in a much smoother and more natural-looking surface, especially when the underlying data is expected to have smooth transitions.

Additionally, RBF interpolation is well-suited for extrapolation beyond the range of the original data points. The basis functions used in RBF interpolation have global support, meaning they can influence the interpolated values even at points far from the known data. This allows RBF interpolation to provide reasonable estimates for extrapolated values, whereas linear interpolation, being local, may not perform as well in extrapolation scenarios.

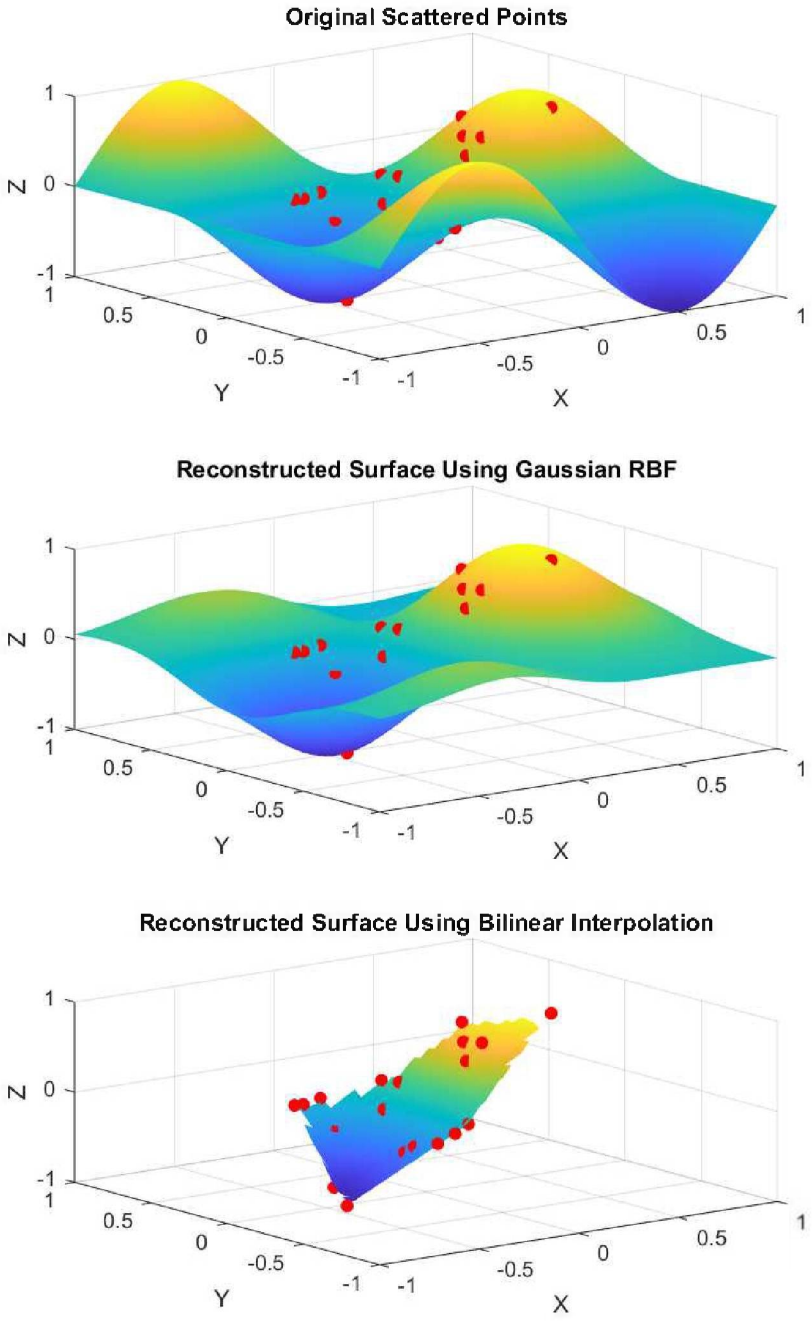


Fig. 2.12: RBF interpolation vs. bilinear interpolation: Sparse scattered data points X, Y generated from a normal distribution. Estimating surface $z = \sin(\pi x) \cdot \cos(\pi y)$.

Chapter 3

Iterations and stability review

3.1 Recommended literature

[4] Saber Elaydi. *An introduction to difference equations.* 3rd. ed. New York: Springer, 2005. ISBN: 03-872-3059-9
[9] Yuri A Kuznetsov. *Elements of Applied Bifurcation Theory.* 2nd ed. New York: Springer, 1998. ISBN: 03-879-8382-1

3.2 Difference equations

A general n-th order difference equation can be written as:

$$x_{k+n} = f(k, x_k, x_{k+1}, \dots, x_{k+n-1}) \tag{3.1}$$

where:

- x is a state variable,
- $x_0, \dots x_{n-1}$ are the initial iterations,
- f is a given function.

For example, a first-order difference equation can be expressed as:

$$x_{k+1} = f(k, x_k)$$

Equation 3.1 is called autonomous (or time-invariant) if

$$x_{k+n} = f(x_k, x_{k+1}, \dots, x_{k+n-1}),$$

otherwise it is called non-autonomous.

An easiest example of difference equation is a first-order linear difference equation. It can be written as:

$$x_{k+1} = a_k(k)\, x_k + b_k(k) \tag{3.2}$$

where:

- x_k is the sequence or function of interest,
- a_k and b_k are given functions of k, which can also be constants,
- if b_k is identically equal to zero the function is called homogeneous,
- if b_k is not identically equal to zero the function is called non-homogeneous.

Suppose the initial value is x_0. The general solution of equation (3.2) con be expressed as:

$$x_k = \left(\prod_{i=0}^{k-1} a_i\right) x_0 + \sum_{j=0}^{k-1} \left(b_j \prod_{i=j+1}^{k-1} a_i\right). \tag{3.3}$$

Examples

1. Prove formula (3.3) for constant $a_k(k) = a$ and arbitrary $b_k(k) \in \mathbb{R}$. Start with solution of homogeneous equation and then proceed with homogeneous equation. Use mathematical induction.
2. Find solution of $x(n+1) = 5x(n) + 7n$, $x(0) = 2$.
3. Suppose that a loan of $1,000,000$ CZK is to be amortized by equal monthly payments. If the interest rate is 5% compounded monthly, find the monthly payment required to pay off the loan in 30 years.
4. A space (three-dimensional) is divided by n planes, nonparallel, and no four planes having a point in common.
   (a) Write a difference equation that describes the number of regions created.
   (b) Find the number of these regions.

Solution

1. Consider the homogeneous linear difference equation:

$$x(n+1) = ax(n)$$

The solution to the homogeneous equation can be found by iterating from the initial condition $x(0)$:

$$x(1) = ax(0)$$
$$x(2) = ax(1) = a^2x(0)$$

$$\dots$$

$$x(n) = a^n x(0)$$

Now, consider the non-homogeneous linear difference equation:

$$x(n+1) = ax(n) + b(n)$$

Assume that the particular solution is of the form $x_p(n)$. The general solution to the non-homogeneous equation is:

$$x(n) = x_h(n) + x_p(n)$$

where $x_h(n)$ is the solution to the homogeneous equation and $x_p(n)$ is a particular solution to the non-homogeneous equation.

Use mathematical induction to prove that the solution is:

$$x(n) = a^n x(0) + \sum_{k=0}^{n-1} a^{n-1-k} b(k)$$

Base Case: For $n = 0$,

$$x(0) = a^0 x(0) + \sum_{k=0}^{-1} a^{-1-k} b(k) = x(0)$$

Inductive Step: Assume the formula holds for n:

$$x(n) = a^n x(0) + \sum_{k=0}^{n-1} a^{n-1-k} b(k)$$

For $n+1$:

$$x(n+1) = ax(n) + b(n)$$

Using the inductive hypothesis:

$$x(n+1) = a\left(a^n x(0) + \sum_{k=0}^{n-1} a^{n-1-k} b(k)\right) + b(n)$$

$$x(n+1) = a^{n+1} x(0) + \sum_{k=0}^{n-1} a^{n-k} b(k) + b(n)$$

$$x(n+1) = a^{n+1} x(0) + \sum_{k=0}^{n-1} a^{n-k} b(k) + a^0 b(n)$$

$$x(n+1) = a^{n+1} x(0) + \sum_{k=0}^{n} a^{n-k} b(k)$$

Thus, by induction, the formula is proved.

2. Find the solution of $x(n+1) = 5x(n) + 7n$, $x(0) = 2$.

   The homogeneous equation is:

$$x_h(n+1) = 5x_h(n)$$

   The solution to the homogeneous equation is:

$$x_h(n) = C \cdot 5^n$$

   To find a particular solution $x_p(n)$, we try a solution of the form $x_p(n) = An + B$. Substitute into the non-homogeneous equation:

$$A(n+1) + B = 5(An + B) + 7n$$
$$An + A + B = 5An + 5B + 7n$$
$$An + A + B = 5An + 5B + 7n$$

   Equating coefficients:

$$A = 7, \quad A + B = 5B$$
$$B = \frac{A}{4} = \frac{7}{4}$$

   Thus, the particular solution is:

$$x_p(n) = 7n + \frac{7}{4}$$

   The general solution is:

$$x(n) = x_h(n) + x_p(n) = C \cdot 5^n + 7n + \frac{7}{4}$$

   Using the initial condition $x(0) = 2$:

$$x(0) = C \cdot 5^0 + 7 \cdot 0 + \frac{7}{4} = 2$$
$$C + \frac{7}{4} = 2$$
$$C = 2 - \frac{7}{4} = \frac{1}{4}$$

   Therefore, the solution is:

$$x(n) = \frac{1}{4} \cdot 5^n + 7n + \frac{7}{4}$$

3. To derive the formula for the monthly payment M for a loan with principal P, interest rate r (expressed as a monthly rate), and a term of n months using a difference equation, we can start by defining the difference equation for the loan balance.

   Let B_k be the loan balance after k months. The difference equation describing the loan balance can be written as:

   $$B_{k+1} = (1 + r)B_k - M$$

   Where:

   - B_{k+1} is the loan balance after $k + 1$ months.
   - B_k is the loan balance after k months.
   - M is the monthly payment.
   - r is the monthly interest rate.

   At the beginning of the loan, the balance is equal to the principal: $B_0 = P$.

   We can solve this difference equation to find the value of M.

   (a) Substitute $k = 0$ to get the initial condition:

   $$B_1 = (1 + r)B_0 - M$$

   $$B_1 = (1 + r)P - M$$

   (b) Continue the iteration process:

   $$\begin{aligned}\text{For } k = 1 :&\\ B_2 &= (1 + r)B_1 - M\\ &= (1 + r)((1 + r)P - M) - M\\ &= (1 + r)^2P - (1 + r)M - M\end{aligned}$$

   $$\begin{aligned}\text{For } k = 2 :&\\ B_3 &= (1 + r)B_2 - M\\ &= (1 + r)((1 + r)^2P - (1 + r)M - M) - M\\ &= (1 + r)^3P - (1 + r)^2M - (1 + r)M - M\end{aligned}$$

   Continuing this process, for $k = n - 1$:

   $$B_n = (1 + r)^nP - M((1 + r)^{n-1} + (1 + r)^{n-2} + ... + 1)$$

   $$B_n = (1 + r)^nP - M((1 + r)^{n-1} + (1 + r)^{n-2} + ... + 1)$$

   (c) At the end of the loan term, $B_n = 0$:

   $$0 = (1 + r)^nP - M((1 + r)^{n-1} + (1 + r)^{n-2} + ... + 1)$$

(d) Now, we can solve for M:

$$M((1+r)^{n-1} + (1+r)^{n-2} + ... + 1) = (1+r)^n P$$
$$M\frac{(1-(1+r)^{-n})}{r} = P$$
$$M = P\frac{r(1+r)^n}{(1+r)^n - 1}$$

where in our case:

- $P = 1,000,000$ CZK is the principal loan amount.
- $r = \frac{0.05}{12} = \frac{1}{240}$ is the monthly interest rate.
- $n = 30 \times 12 = 360$ is the total number of payments.

Substituting the values:

$$M = 1,000,000 \cdot \frac{\frac{1}{240}(1+\frac{1}{240})^{360}}{(1+\frac{1}{240})^{360} - 1} \approx 1,000,000 \cdot \frac{0.0186}{3.4677} \approx 5,373.46$$

So the monthly payment required to pay off the loan in 30 years is approximately:

$$M \approx 5,373.46 \text{ CZK}$$

4. **(a) Difference Equation**

Let $R(n)$ be the number of regions created by n planes. The difference equation describing the number of regions is:

$$R(n+1) = R(n) + n$$

(b) Find the Number of Regions

The initial condition is $R(0) = 1$.

To solve the difference equation:

$$R(n+1) = R(n) + n$$

Substituting into both sides from $n = 0$ to $n = k$:

$$R(k) = R(0) + \sum_{n=0}^{k} n$$

Since $R(0) = 1$ and $\sum_{n=0}^{k} n = \frac{k(k+1)}{2}$:

$$R(k) = 1 + \frac{k(k+1)}{2}$$

Thus, the number of regions created by n planes is:

$$R(n) = 1 + \frac{n(n+1)}{2}$$

Example 5: Decay Rate of Uranium Isotopes

The **decay constant**, denoted by λ, is a parameter that characterizes the rate at which a radioactive substance undergoes decay. It is defined as the probability per unit time that a given atom will decay. The **half-life**, denoted by $t_{1/2}$, is the time required for half of the atoms in a sample of a radioactive substance to decay.

1. Formulate difference equation for radioactive decay using a decay constant λ.
2. Assume two isotopes of uranium and their half-life $t_{1/2}$. Namely:
   - **Uranium-238 (U-238)**: $t_{1/2}$ is approximately 4.468 billion years.
   - **Uranium-235 (U-235)**: $t_{1/2}$ is approximately 703.8 million years.

   What are the decay constants λ for them?

Difference Equation for Radioactive Decay

A difference equation describing a decay process can be formulated using the decay constant λ and the initial number of atoms N_0. For discrete time steps, the number of atoms N at time step t can be expressed as follows:

$$N(t+1) = N(t) - \lambda N(t)$$

However, to make this more precise and conventional, we often use a small time step Δt. The discrete form of the exponential decay law is:

$$N(t+\Delta t) = N(t)\,(1-\lambda\Delta t)$$

where:

- $N(t)$ is the number of atoms at time t.
- Δt is the discrete time step.
- λ is the decay constant.

If we start with an initial number of atoms N_0 at $t = 0$, the equation can be expressed as:

$$N(t+\Delta t) = N_0\,(1-\lambda\Delta t)^{\frac{t}{\Delta t}+1}$$

As $\Delta t \to 0$, this expression approaches the form of an exponential decay. Specifically:

$$\lim_{\Delta t \to 0} (1 - \lambda \Delta t)^{\frac{t}{\Delta t}} = e^{-\lambda t}$$

Therefore, the continuous form of the decay equation is:

$$N(t) = N_0 e^{-\lambda t}.$$

You can substitute $N(t) = \frac{N_0}{2}$ and $t = t_{1/2}$ solve for $t_{1/2}$.

$$\lambda = \frac{ln(2)}{t_{1/2}}.$$

- For U-238:
$$\lambda = \frac{\ln(2)}{4.468 \times 10^9 \text{ years}} \approx 1.55 \times 10^{-10} \text{ per year}$$
- For U-235:
$$\lambda = \frac{\ln(2)}{703.8 \times 10^6 \text{ years}} \approx 9.85 \times 10^{-10} \text{ per year.}$$

3.3 Equilibrium Definition

For a difference equation $x_{n+1} = f(n, x_n)$, an **equilibrium sequence** $\{x^*\}$ (fixed point) is a sequence that satisfies:

$$x^* = f(n, x^*).$$

3.4 Stability Definitions

For a difference equation $x_{n+1} = f(n, x_n)$:

1. **Stable Equilibrium Sequence**: An equilibrium sequence $\{x_n^*\}$ is stable if for every $\epsilon > 0$, there exists a $\delta > 0$ such that if $|x_0 - x_0^*| < \delta$, then $|x_n - x_n^*| < \epsilon$ for all $n \geq 0$.
$$\forall \epsilon > 0, \exists \delta > 0 \text{ such that if } |x_0 - x_0^*| < \delta, \text{ then } |x_n - x_n^*| < \epsilon \text{ for all } n \geq 0.$$
2. **Asymptotically Stable Equilibrium Sequence**: An equilibrium sequence $\{x_n^*\}$ is asymptotically stable if it is stable and $\lim_{n\to\infty} |x_n - x_n^*| = 0$. That is, there exists a $\delta > 0$ such that if $|x_0 - x_0^*| < \delta$, then $\lim_{n\to\infty} |x_n - x_n^*| = 0$.
$$\exists \delta > 0 \text{ such that if } |x_0 - x_0^*| < \delta, \text{ then } \lim_{n\to\infty} |x_n - x_n^*| = 0.$$
3. **Unstable Equilibrium Sequence**: An equilibrium sequence $\{x_n^*\}$ is unstable if it is not stable.

Examples

Example 6: Simple Linear Autonomous System

$$x_{n+1} = ax_n$$

Equilibrium

The equilibrium point x^* is found by solving:

$$x^* = ax^*$$

Thus, $x^* = 0$ is the equilibrium point.

Stability

Linearize around $x^* = 0$:

$$\varepsilon_{n+1} = a\varepsilon_n,$$

where $\varepsilon_n = x_n - x^*$.

- If $|a| < 1$, $x^* = 0$ is asymptotically stable.
- If $|a| > 1$, $x^* = 0$ is unstable.
- If $|a| = 1$, $x^* = 0$ is stable, but not asymptotically stable.

Example 7: Logistic map

The basic form of the logistic map is a first-order, nonlinear difference equation given by:

$$x_{n+1} = rx_n(1 - x_n)$$

where:

- x_n represents the population size at the n-th time step, normalized so that the carrying capacity of the environment is 1.
- r is a positive constant representing the intrinsic growth rate of the population.

The logistic map arises from the assumption that the population growth rate decreases linearly as the population size approaches the carrying capacity of the environment. The term $rx_n(1 - x_n)$ represents the net growth rate, where rx_n is the maximum per capita growth rate and $(1 - x_n)$ represents the fraction of resources available for growth.

Equilibrium

The equilibrium points x^* are found by solving:

$$x^* = rx^*(1 - x^*)$$

Thus, $x^* = 0$ and $x^* = 1 - \frac{1}{r}$ (for $r > 1$) are the equilibrium points.

Stability

Linearize around x^*:

$$\varepsilon_{n+1} = r(1 - 2x^*)\varepsilon_n$$

- For $x^* = 0$:

  $$\varepsilon_{n+1} = r\varepsilon_n$$

  - If $|r| < 1$, $x^* = 0$ is asymptotically stable.
  - If $|r| > 1$, $x^* = 0$ is unstable.

- For $x^* = 1 - \frac{1}{r}$:

  $$\varepsilon_{n+1} = (2 - r)\varepsilon_n$$

  - If $|2 - r| < 1$, $x^* = 1 - \frac{1}{r}$ is asymptotically stable.
  - If $|2 - r| > 1$, $x^* = 1 - \frac{1}{r}$ is unstable.

Example 8: Simple Non-Autonomous System

Consider a non-autonomous difference equation:

$$x_{n+1} = \frac{1}{2}x_n + \frac{\sin(n)}{n}$$

Equilibrium

To find the equilibrium sequence $\{x_n^*\}$, solve:

$$x^* = \frac{1}{2}x^* + \frac{\sin(n)}{n}$$

A correct sequence that approximates the equilibrium is:

$$x^* = \frac{2\sin(n)}{n}$$

Stability

Linearize around x^*:

$$\varepsilon_{n+1} = \frac{1}{2}\varepsilon_n$$

Since $\left|\frac{1}{2}\right| < 1$, the equilibrium sequence $x^* = \frac{2\sin(n)}{n}$ is asymptotically stable.

Example 9: Logistic grow with seasonal effect

Consider a non-autonomous system:

$$x_{n+1} = x_n\left(1 - \frac{x_n}{K(n)}\right) + a\cos(bn)$$

Equilibrium

Find the equilibrium sequence $\{x_n^*\}$ by solving:

$$x^* = x^*\left(1 - \frac{x^*}{K(n)}\right) + a\cos(bn)$$

This typically requires numerical methods for specific $K(n)$.

Stability

Linearize around x^*:

$$\varepsilon_{n+1} = \left(1 - \frac{2x^*}{K(n)}\right)\varepsilon_n$$

The stability depends on $\left|1 - \frac{2x^*}{K(n)}\right|$. If this term is less than 1 in magnitude for all n, the equilibrium sequence is stable. Otherwise, it is unstable.

3.5 Criterion for Asymptotic Stability

Consider an autonomous difference equation of the form:

$$x_{n+1} = f(x_n)$$

Let x^* be an equilibrium point, i.e., $x^* = f(x^*)$. The equilibrium point x^* is asymptotically stable if the magnitude of the derivative of f evaluated at x^* is less than 1:

$$|f'(x^*)| < 1.$$

3.6 Destabilization through Bifurcations

Fold (Saddle-Node) Bifurcation

A fold bifurcation occurs when a pair of fixed points (equilibria) of the difference equation $x_{n+1} = f(x_n)$ are created or destroyed as a parameter μ is varied. This typically happens when a stable and an unstable fixed point collide and annihilate each other.

Conditions for Fold Bifurcation

- At the bifurcation point, there is a fixed point x^* such that:

$$f(x^*, \mu) = x^*$$

- The first derivative of f with respect to x at x^* is:

$$f'_x(x^*, \mu) = 1$$

- Non-degeneracy and transversality conditions

$$f''_{xx}(x^*, \mu) \neq 0$$

$$f'_\alpha(x^*, \mu) \neq 0$$

At the fold bifurcation, the slope of the function f at the fixed point is 1, and the curvature is non-zero, leading to the merging and subsequent disappearance of two fixed points.

There exists two important non-generic bifurcations associated to $f'(x^*, \mu) = 1$. Those are:

Transcritical Bifurcation: At the transcritical bifurcation, the fixed points interchange their stability as the parameter μ passes through the bifurcation value.

- Fixed points condition: $f(x_1^*) = x_1^*$ and $f(x_2^*) = x_2^*$

- Derivative condition: $f'(x_1^*) = 1$ and $f'(x_2^*) = 1$

- Higher-order terms determine stability exchange.

Pitchfork Bifurcation: In the pitchfork bifurcation, the original fixed point becomes unstable and two new stable fixed points emerge.

- Fixed point condition: $f(x^*) = x^*$

- Derivative condition: $f'(x^*) = 1$

- Second derivative condition: $f''(x^*) = 0$

- Third derivative condition: $f'''(x^*) \neq 0$

Flip (Period-Doubling) Bifurcation

A flip bifurcation occurs when a fixed point of the difference equation $x_{n+1} = f(x_n)$ loses stability, and a periodic orbit of period 2 emerges. This is also known as a period-doubling bifurcation.

Conditions for Flip Bifurcation

- At the bifurcation point, there is a fixed point x^* such that:

$$f(x^*, \mu) = x^*$$

- The first derivative of f with respect to x at x^* is:

$$f'(x^*, \mu) = -1$$

- Non-degeneracy and transversality conditions:

$$\frac{1}{2}(f''_{xx}(x^*, \mu))^2 + \frac{1}{3}f'''_{xxx}(x^*, \mu) \neq 0$$

$$f''_{x\alpha}(x^*, \mu) \neq 0$$

At the flip bifurcation, the fixed point becomes unstable, and a new stable periodic orbit with double the period of the original fixed point emerges, indicated by the slope of the function f at the fixed point being -1.

3.7 Definition of a Cobweb Plot

A cobweb plot is a graphical representation used to visualize the behavior of a dynamic system described by an autonomous difference equation $x_{n+1} = f(x_n)$. It helps to illustrate the iterative process of the system's evolution over time.

The cobweb plot consists of two key components:

1. **Iterative Dynamics**: The plot shows the iteration of the difference equation, typically represented by a function $f(x)$, where x is the current state of the system. Starting from an initial value x_0, the plot illustrates the sequence of points (x_n, x_{n+1}) as the system evolves over discrete time steps.
2. **Identity Line**: The diagonal line $y = x$ represents the identity line, where the x-coordinate is equal to the y-coordinate. This line helps visualize the equilibrium points of the system, where $x_{n+1} = x_n$.

To create a cobweb plot:

1. Start with an initial value x_0 on the x-axis.
2. Plot the point $(x_0, f(x_0))$ on the graph, where $f(x_0)$ is the value of the difference equation evaluated at x_0.
3. Draw a vertical line from the point $(x_0, f(x_0))$ to the identity line.
4. At the intersection point, plot the point $(f(x_0), f(f(x_0)))$.
5. Repeat steps 3 and 4 for subsequent iterations to create a "cobweb" pattern.

The resulting plot visually demonstrates the behavior of the system, including convergence to equilibrium points, periodic orbits, or chaotic behavior, depending on the properties of the difference equation.

For asymptotically stable points the slope of f evaluated in the equilibrium x^* is larger than -1 and smaller than 1.

Example 10: Logistic map

Study dynamics of logistic map

$$x^* = rx^*(1 - x^*)$$

for $x \in [0, 1]$ and $r \in [0, 4]$. In the previous chapter we showed that logistic map has two equlibria. We also determined their asymptotic stability:

- For $x^* = 0$:
  - If $|r| < 1$, $x^* = 0$ is asymptotically stable.
  - If $|r| > 1$, $x^* = 0$ is unstable.
- For $x^* = 1 - \frac{1}{r}$:
  - If $|2 - r| < 1$, $x^* = 1 - \frac{1}{r}$ is asymptotically stable.
  - If $|2 - r| > 1$, $x^* = 1 - \frac{1}{r}$ is unstable.

1. What dynamical phenomenon occur in the logistic map while parameter r is crossing critical values $r = 1$ and $r = 3$?
2. Create a Cobweb Plot for logistic map. Use $x_0 = 1$ and $r \in \{0.9, 1.5, 2.5, 3.2, 3.46, 3.6, 4\}$.

Solution

- For $r < 1$: The fixed point $x^* = 0$ is stable, and the solution tends to zero.

$$x_{n+1} = rx_n(1 - x_n) \quad \text{with} \quad 0 < r < 1$$

- At $r = 1$: A transcritical bifurcation occurs; the fixed point $x^* = 0$ becomes unstable, and a new stable fixed point $x^* = 1 - \frac{1}{r}$ appears for $r > 1$.

$$x_{n+1} = rx_n(1 - x_n) \quad \text{with} \quad r = 1$$

- For $1 < r < 3$: The fixed point $x^* = 1 - \frac{1}{r}$ is stable, and the system converges to this fixed point.

$$x_{n+1} = rx_n(1 - x_n) \quad \text{with} \quad 1 < r < 3$$

- At $r = 3$: A period-doubling bifurcation occurs, and the fixed point $x^* = 1 - \frac{1}{r}$ becomes unstable. A stable period-2 orbit emerges for $r > 3$.

$$x_{n+1} = rx_n(1 - x_n) \quad \text{with} \quad r = 3$$

Solution of the part 2 is provided as Matlab live script `logistic_map.mlx`. Notice that $3 < r \leq 4$ the logistic map undergoes cascade of period doubling and that leads to deterministic chaos. You can find more about this phenomenon in [4, 9].

3.8 Linear Homogeneous Difference Equations with Constant Coefficients

A **linear homogeneous difference equation with constant coefficients** can be expressed as:

$$a_n x_{k+n} + a_{n-1} x_{k+n-1} + \cdots + a_1 x_{k+1} + a_0 x_k = 0$$

where $a_0, a_1, \ldots, a_n$ are constants, and $a_n \neq 0$.

General Solution

The solution to this equation involves the characteristic polynomial associated with it:

$$a_n \lambda^n + a_{n-1} \lambda^{n-1} + \cdots + a_1 \lambda + a_0 = 0$$

1. **Find the roots of the characteristic polynomial**: The roots of the characteristic polynomial can be real or complex and may have multiplicity. Let these roots be $\lambda_1, \lambda_2, \ldots, \lambda_k$, with corresponding multiplicities $m_1, m_2, \ldots, m_k$.

2. **Construct the general solution**:

   - For each distinct root λ_i with multiplicity m_i, the contribution to the general solution is:

   $$x_i(n) = (C_{i1} + C_{i2} n + \cdots + C_{im_i} n^{m_i - 1}) \lambda_i^n$$

   - For each distinct pair of complex roots $\lambda_{i,i+1} = \alpha \pm \beta i$ (where $\alpha, \beta \in \mathbb{R}$) we can apply Euler's formula, $e^{i\theta} = \cos(\theta) + i \sin(\theta)$:
   The complex roots $\alpha \pm \beta i$ can be written as:

   $$\alpha + \beta i = re^{i\phi}$$

   $$\alpha - \beta i = re^{-i\phi}$$

where $r = \sqrt{\alpha^2 + \beta^2}$ and $\phi = \tan^{-1}\left(\frac{\beta}{\alpha}\right)$. We can therefore apply Euler's formula to $\lambda_{i,i+1}$:

$$(\alpha + \beta i)^k = \alpha^k \left(\cos(\beta k) + i\sin(\beta k)\right)$$
$$(\alpha - \beta i)^k = \alpha^k \left(\cos(\beta k) - i\sin(\beta k)\right)$$

General solution of the difference equation is linear combination of all the contributions mentioned above where constants are determined by initial conditions.

Relation to Systems of Linear Equations

1. **Homogeneous linear difference equations can be converted to systems of first-order linear difference equations**. For example, the n-th order difference equation:
$$a_n x_{k+n} + a_{n-1}x_{k+n-1} + \cdots + a_1 x_{k+1} + a_0 x_k = 0$$
can be rewritten as a system of first-order difference equations:
$$\begin{cases} x_k^1 = x_k \\ x_{k+1}^1 = x_{k+1} = x_k^2 \\ x_{k+1}^2 = x_{k+2} = x_k^3 \\ x_{k+1}^3 = x_{k+3} = x_k^4 \\ \vdots \\ x_{k+1}^{n-1} = x_{k+n-1} = x_k^n \\ x_{k+1}^n = x_{k+n} = -\frac{a_{n-1}}{a_n}x_k^n - \frac{a_{n-2}}{a_n}x_k^{n-1} - \cdots - \frac{a_1}{a_n}x_k^2 - \frac{a_0}{a_n}x_k^1 \end{cases}$$

2. **Matrix representation**: This system can be written in matrix form:
$$\mathbf{X}_{k+1} = B\mathbf{X}_k$$
where $\mathbf{X}_k = \begin{pmatrix} x_k \\ x_{k+1} \\ \vdots \\ x_{k+n-1} \end{pmatrix}$ and B is a matrix composed of the coefficients from the system above.

3. **Solution via eigenvalues and eigenvectors**: The general solution to this system is determined by the eigenvalues and eigenvectors of the matrix B, which correspond to the roots of the characteristic polynomial of the original difference equation.

The structure and solutions of homogeneous linear difference equations with constant coefficients are deeply tied to the linear algebraic properties of the associated coefficient matrices. This interplay allows for methods from linear algebra to be applied to difference equations and vice versa.

3.9 Leslie Model

Definition

The Leslie model is a discrete, age-structured population projection model used in demography and ecology to describe the growth and structure of a population over time. It uses a Leslie matrix, which incorporates age-specific fertility and survival rates, to project the population distribution across different age classes.

Let $\mathbf{n}(t)$ be a vector representing the population size at time t, where each element $n_i(t)$ represents the number of individuals in the i-th age class at time t. The population at the next time step, $\mathbf{n}(t+1)$, is given by:

$$\mathbf{n}(t+1) = L\mathbf{n}(t)$$

where L is the Leslie matrix, defined as:

$$L = \begin{pmatrix} f_1 & f_2 & \cdots & f_{m-1} & f_m \\ s_1 & 0 & \cdots & 0 & 0 \\ 0 & s_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & s_{m-1} & 0 \end{pmatrix}$$

Here, f_i represents the fertility rate of the i-th age class, and s_i represents the survival rate from the i-th age class to the $(i+1)$-th age class.

Perron-Frobenius Theorem

A nonnegative matrix is a matrix in which all the entries are greater than or equal to zero. Formally, an $n \times n$ matrix $A = [a_{ij}]$ is nonnegative if:

$$a_{ij} \geq 0 \quad \text{for all } 1 \leq i, j \leq n.$$

A nonnegative $n \times n$ matrix A is irreducible if, for every pair of indices i and j, there exists a positive integer k such that the (i, j)-th entry of A^k (the matrix A raised to the power k) is positive:

$$(A^k)_{ij} > 0 \quad \text{for some } k > 0.$$

For a non-negative, irreducible matrix A, the Perron-Frobenius theorem states:

1. There exists a unique largest positive eigenvalue λ_{max} such that $\lambda_{\text{max}} > 0$.
2. The eigenvalue λ_{max} has a corresponding positive eigenvector.
3. The spectral radius (the largest absolute value of the eigenvalues) is equal to λ_{max}.

Applying Perron-Frobenius to the Leslie Model

The Leslie matrix L is non-negative and often irreducible. By applying the Perron-Frobenius theorem, we derive the following properties:

Population Growth Rate

The dominant eigenvalue $\lambda_{\max}$ of the Leslie matrix L determines the long-term growth rate of the population: If $\lambda_{\max} > 1$, the population grows exponentially. If $\lambda_{\max} < 1$, the population declines. If $\lambda_{\max} = 1$, the population remains constant.

Stable Age Distribution

The eigenvector $\mathbf{v}_{\max}$ corresponding to $\lambda_{\max}$ represents the stable age distribution:

$$L\mathbf{v}_{\max} = \lambda_{\max}\mathbf{v}_{\max}$$

As time progresses, the population vector $\mathbf{n}(t)$ converges to a direction proportional to $\mathbf{v}_{\max}$, regardless of the initial population distribution $\mathbf{n}(0)$.

Asymptotic Behavior

Given an initial population vector $\mathbf{n}(0)$, the population at time t is:

$$\mathbf{n}(t) = L^t\mathbf{n}(0)$$

Using spectral decomposition, L can be written in terms of its eigenvalues and eigenvectors. As $t \to \infty$, the term involving $\lambda_{\max}$ dominates:

$$\mathbf{n}(t) \approx \lambda_{\max}^t\mathbf{v}_{\max}$$

Thus, the long-term behavior of the population is determined by the dominant eigenvalue and the corresponding eigenvector of the Leslie matrix. To know more about this application in contex of the history of studying population dynamics see [2].

Example: Study the following paper [3] Larry B. Crowder et al. "Predicting the Impact of Turtle Excluder Devices on Loggerhead Sea Turtle Populations". In: *Ecological Applications* 4.3 (1994), pp. 437–445. ISSN: 1051-0761. DOI: 10.2307/1941948.

3.10 Markov Chains

A Markov chain is a stochastic process that undergoes transitions from one state to another on a state space. It is a memoryless process, meaning the probability of transitioning to the next state depends only on the current state and not on the previous states.

For a discrete-time Markov chain, we can describe the system's behavior over time using a transition matrix. The general solution of difference equations is closely related to the long-term behavior of Markov chains.

Discrete-Time Markov Chain

Let $\{X_t\}$ be a discrete-time Markov chain with a finite state space $S = \{1, 2, \ldots, n\}$. The transition probability matrix P is an $n \times n$ matrix where each element p_{ij} represents the probability of transitioning from state i to state j in one time step:

$$P = [p_{ij}] \quad \text{where} \quad p_{ij} = \mathbb{P}(X_{t+1} = j \mid X_t = i)$$

State Vector

Let $\mathbf{v}(t)$ be the state vector at time t, where each element $v_i(t)$ represents the probability of being in state i at time t:

$$\mathbf{v}(t) = \begin{pmatrix} v_1(t) \\ v_2(t) \\ \vdots \\ v_n(t) \end{pmatrix}$$

The evolution of the state vector over time can be described by the difference equation:

$$\mathbf{v}(t+1) = P\mathbf{v}(t)$$

Long-Term Behavior

As $t \to \infty$, the behavior of $\mathbf{v}(t)$ is dominated by the eigenvalue of P with the largest absolute value (which is 1 for a Markov chain with a stationary distribution). Let $\lambda_1 = 1$ be the largest eigenvalue and $\mathbf{u}_1$ be the corresponding eigenvector (which corresponds to the stationary distribution):

$$\mathbf{v}(t) \approx \mathbf{u}_1 \quad \text{as } t \to \infty$$

This implies that the state vector converges to the stationary distribution $\mathbf{u}_1$, providing the long-term behavior of the Markov chain.

Example 11: User Navigation on a Website

Scenario: A website has three main pages: S_1: Home Page, S_2: Products Page, S_3: Checkout Page. The website administrator wants to model the behavior of users as they navigate between these pages. Based on user data, they determine the following transition probabilities:

- From the Home Page, users move to the Products Page with a probability of 0.5 and stay on the Home Page with a probability of 0.5.
- From the Products Page, users move to the Checkout Page with a probability of 0.1, return to the Home Page with a probability of 0.2, and stay on the Products Page with a probability of 0.7.
- From the Checkout Page, users return to the Home Page with a probability of 0.1, return to the Products Page with a probability of 0.3, and stay on the Checkout Page with a probability of 0.6.

1. Write down transition probability matrix.
2. Find the steady-state distribution representing long-term behavior of the model.
3. Assume user starts at Home Page. Write down the probability distribution after 10 steps.

Solution: These probabilities can be represented in the transition matrix P:

$$P = \begin{pmatrix} 0.5 & 0.2 & 0.1 \\ 0.5 & 0.7 & 0.3 \\ 0 & 0.1 & 0.6 \end{pmatrix}$$

Steady-State Distribution: By analyzing the eigenvectors and eigenvalues, particularly the dominant eigenvalue ($\lambda_1 = 1$), the website can determine the steady-state distribution of users across the pages. The normalized eigenvector for $\lambda_1 = 1$ indicates that, over time, approximately 33.33% of users will be on each page.

Short-Term Predictions: Using P^{10}, the website can find the expected distribution of users starting from the Home Page.

The result of $(P^T)^{10} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$ is approximately:

$$\begin{pmatrix} 0.26505226 \\ 0.58872298 \\ 0.14622476 \end{pmatrix}.$$

Example 12: One-Dimensional Random Walk

Suppose we have a random walk on the integers $\{0, 1, 2, \ldots, n\}$, where at each time step, the walker moves to an adjacent state with equal probability. We can describe the transition probabilities as follows:

- If the walker is at state i (where $1 \leq i \leq n-1$), there is a probability of 0.5 of moving to state $i-1$ and a probability of 0.5 of moving to state $i+1$.

- If the walker is at the boundary states (0 or n), the walker stays in the same state.

1. Write down the transition probability matrix for $n = 4$.
2. Assume walker starts at state 1. Write down a probability distribution of walker's position after two steps.
3. Write down a probability distribution of walker's position as the number of steps tends to infinity.

Solution: The transition probability matrix P for this random walk can be constructed as follows. Suppose we have a random walk on $\{0, 1, 2, 3, 4\}$. The transition matrix P is:

$$P = \begin{pmatrix} 1 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & 0 & 0 \\ 0 & 0.5 & 0 & 0.5 & 0 \\ 0 & 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 1 \end{pmatrix}$$

In general, for a random walk on $\{0, 1, 2, \ldots, n\}$, the transition matrix P will have the following form:

$$P = \begin{pmatrix} 1 & 0.5 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0.5 & \cdots & 0 & 0 & 0 \\ 0 & 0.5 & 0 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 0.5 & 0 \\ 0 & 0 & 0 & \cdots & 0.5 & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0.5 & 1 \end{pmatrix}$$

Consider the initial state vector $\mathbf{v}(0)$ where the walker starts at state 1:

$$\mathbf{v}(0) = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Using the transition matrix P for $\{0, 1, 2, 3\}$, the state vector at time $t = 1$ is:

$$\mathbf{v}(2) = P^2\mathbf{v}(0) = \begin{pmatrix} 1 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & 0 & 0 \\ 0 & 0.5 & 0 & 0.5 & 0 \\ 0 & 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 1 \end{pmatrix}^2 \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0.5 \\ 0.25 \\ 0 \\ 0.25 \\ 0 \end{pmatrix}$$

Long-Term Behavior: For matrix P, both the algebraic and geometric multiplicity of eigenvalue 1 is 2. Corresponding eigenvectors are $\mathbf{v}_1 = (1, 0, 0, 0, 0)^T$ and $\mathbf{v}_2 = (0, 0, 0, 0, 1)^T$. As $t \to \infty$, for the random walk on a finite state space with absorbing states (like 0 and n), the long-term behavior is that the walker will eventually be absorbed at one of these states, the exact probability ratio depends on $\mathbf{v}(0)$.

3.11 Power Method

The power method is an iterative algorithm used to find the dominant eigenvalue and its corresponding eigenvector of a matrix A.

Given a matrix A and an initial vector $\mathbf{x}_0$, the method iteratively computes:

$$\mathbf{x}_{k+1} = \frac{A\mathbf{x}_k}{\|A\mathbf{x}_k\|}.$$

Eigenvalue Estimation: Once the vector has converged to an eigenvector $\mathbf{v}_{\text{max}}$, the dominant eigenvalue can be estimated as:

$$\lambda_{\text{max}} \approx \frac{\mathbf{v}_{\text{max}}^T A \mathbf{v}_{\text{max}}}{\mathbf{v}_{\text{max}}^T \mathbf{v}_{\text{max}}}$$

When we represent the system of difference equations in matrix form, the behavior of the system over iterations is governed by the matrix A. Specifically, consider the system:

$$\mathbf{x}_{k+1} = A\mathbf{x}_k$$

where $\mathbf{x}_k$ is the state vector at step k. The general solution to this system can be expressed in terms of the eigenvalues and eigenvectors of A.

Let λ_i be the eigenvalues of A and $\mathbf{v}_i$ be the corresponding eigenvectors. If A is diagonalizable, we can write:

$$A = V\Lambda V^{-1}$$

where V is the matrix of eigenvectors and Λ is the diagonal matrix of eigenvalues. The solution can then be expressed as:

$$\mathbf{x}_k = A^k\mathbf{x}_0 = (V\Lambda V^{-1})^k\mathbf{x}_0 = V\Lambda^k V^{-1}\mathbf{x}_0$$

As $k \to \infty$, the behavior of $\mathbf{x}_k$ is dominated by the eigenvalue λ_{max} with the largest magnitude (dominant eigenvalue). The corresponding eigenvector $\mathbf{v}_{\text{max}}$ gives the mode shape of the system.

- **Stability**: The magnitude $|\lambda_{\text{max}}|$ determines the stability of the system. If $|\lambda_{\text{max}}| < 1$, then $\mathbf{x}_k \to 0$ as $k \to \infty$, indicating stability. If $|\lambda_{\text{max}}| > 1$, $\mathbf{x}_k$ grows without bound, indicating instability.
- **Mode Shape**: The eigenvector $\mathbf{v}_{\text{max}}$ associated with λ_{max} describes how the components of $\mathbf{x}_k$ evolve relative to each other as $k \to \infty$. Specifically, as $k \to \infty$, the state vector $\mathbf{x}_k$ aligns with $\mathbf{v}_{\text{max}}$.

3.12 Autonomous Systems of Nonlinear Difference Equations

An **autonomous system of nonlinear difference equations** refers to a system of equations where the next state of the dependent variables is determined by the current state, independent of an external variable such as time. Mathematically, it can be expressed as:

$$\mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}_k)$$

where $\mathbf{x}_k \in \mathbb{R}^n$ is the vector of dependent variables at the discrete time step k, and $\mathbf{f}(\mathbf{x}_k)$ is a vector-valued function representing the system dynamics.

3.12.1 Equilibrium Points

An **equilibrium point** (or fixed point) of an autonomous difference equation system is a point $\mathbf{x}^* \in \mathbb{R}^n$ where the system remains at rest if it starts there. Mathematically, $\mathbf{x}^*$ is an equilibrium point if:

$$\mathbf{x}^* = \mathbf{f}(\mathbf{x}^*)$$

At this point, the state of the system does not change from one time step to the next.

3.12.2 Linearization of Nonlinear Difference Systems

To analyze the stability of equilibria in nonlinear difference systems, linearization is a common approach. Given an equilibrium point $\mathbf{x}^*$, the nonlinear system can be approximated by a linear system in the vicinity of $\mathbf{x}^*$.

The linearization process involves computing the Jacobi matrix $\mathbf{J}$ of $\mathbf{f}$ at $\mathbf{x}^*$:

$$\mathbf{J} = \left.\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}^*}$$

The linearized system around the equilibrium point $\mathbf{x}^*$ is then given by:

$$\mathbf{y}_{k+1} = \mathbf{J}\mathbf{y}_k$$

where $\mathbf{y}_k = \mathbf{x}_k - \mathbf{x}^*$ represents the perturbation from the equilibrium.

3.12.3 Criteria for Stability of Equilibria

The stability of the equilibrium point $\mathbf{x}^*$ in the linearized system can be analyzed by examining the eigenvalues of the Jacobi matrix $\mathbf{J}$. Notice that we can apply definitions of stability introduced in section 3.4.

- **Stable Equilibrium (Asymptotically Stable)**: If all eigenvalues of $\mathbf{J}$ have magnitudes less than one (i.e., $|\lambda_i| < 1$ for all i), the equilibrium $\mathbf{x}^*$ is asymptotically stable. Perturbations from $\mathbf{x}^*$ will decay exponentially over time, and the system will return to the equilibrium state.
- **Unstable Equilibrium**: If at least one eigenvalue of $\mathbf{J}$ has a magnitude greater than one (i.e., $|\lambda_i| > 1$ for some i), the equilibrium $\mathbf{x}^*$ is unstable. Perturbations from $\mathbf{x}^*$ will grow exponentially over time, and the system will move away from the equilibrium state.
- **Marginally Stable (or Lyapunov Stable) Equilibrium**: If all eigenvalues of $\mathbf{J}$ have magnitudes less than or equal to one (i.e., $|\lambda_i| \leq 1$ for all i) and at least one eigenvalue has a magnitude exactly equal to one (i.e., $|\lambda_i| = 1$ for some i), further analysis is required. The system may be stable (in the sense of Lyapunov) if perturbations neither grow nor decay, but this stability is not asymptotic. This scenario often necessitates a more detailed nonlinear analysis to determine stability.

In section 3.6 we introduced one-parameter bifurcation of an equilibrium. The same one-parameter bifurcation can be studied in systems of nonlinear difference equations though technique that is called Central Manifold Theorem, see [9]. Additionally, there exists another one-parameter bifurcation associated with a pair of complex conjugate eigenvalues of the Jacobi matrix $\mathbf{J}$ called Neimark-Sacker (NS) bifurcation. NS bifurcation leads to the emergence of quasi-periodic behavior or invariant closed curves (cycles) in the phase space, see [9].

Chapter 4

Fixed Point Iterative Methods

4.1 Recommended literature

[4] Saber Elaydi. *An introduction to difference equations.* 3rd. ed. New York: Springer, 2005. ISBN: 03-872-3059-9
[10] John H Mathews, Kurtis D Fink, et al. *Numerical methods using MATLAB.* vol. 4. Pearson prentice hall Upper Saddle River, NJ, 2004
[8] Václav Kučera. *Numerical Methods for Nonlinear Equations.* Univerzita Karlova, Matematicko-fyzikální fakulta, Matematická sekce, 2022, URL: https://www.karlin.mff.cuni.cz/~kucera/Numerical_Methods_for_Nonlinear_Equations.pdf
[14] Josef Stoer et al. *Introduction to numerical analysis.* Vol. 1993. Springer, 1980

4.2 Fixed Point Iterative Methods in the Context of Difference Equations

4.2.1 Problem Definition

Consider a nonlinear equation of the form:

$$f(x) = 0$$

To solve this equation using fixed point iteration, we rewrite it as:

$$x = g(x)$$

Here, $g(x)$ is a function that transforms the problem into finding a fixed point of g. That is, a point x^* such that $g(x^*) = x^*$. The fixed point iteration generates a sequence $\{x_n\}$ given by:

$$x_{n+1} = g(x_n)$$

Example 1:

Find solution of $f(x) = 0$ on $[0, 1]$. Does the suggested iterative function $g(x)$ work?

- $f(x) = x - 2^{x-2} \implies g(x) = 2^{x-2}, x_0 = 1$
- $f(x) = x - 2^{-x} \implies g(x) = 2^{-x}, x_0 = 1$
- $f(x) = 3x^2 - x - 1 \implies g(x) = 3x^2 - 1, x_0 = 1$

Solution:

- For $f(x) = x - 2^{x-2}$ with $g(x) = 2^{x-2}$ and $x_0 = 1$:
  - The function $g(x) = 2^{x-2}$ appears to converge to a fixed point upon iteration starting from $x_0 = 1$, but a detailed convergence analysis is needed to confirm.
- For $f(x) = x - 2^{-x}$ with $g(x) = 2^{-x}$ and $x_0 = 1$:
  - The function $g(x) = 2^{-x}$ also appears to converge to a fixed point upon iteration starting from $x_0 = 1$.
- For $f(x) = 3x^2 - x - 1$ with $g(x) = 3x^2 - 1$ and $x_0 = 1$:
  - The function $g(x) = 3x^2 - 1$ diverges, indicating it does not work as an iterative method for the given interval.

4.2.2 Convergence and Stability

For the iterative method to converge to the fixed point x^*, we analyze the stability of the difference equation $x_{n+1} = g(x_n)$. This analysis involves examining the derivative $g'(x)$ at the fixed point. Let us recall concept of asymptotic stability, that was discussed in section 3.4. Mathematically, asymptotic stability is captured by the condition:

$$|g'(x^*)| < 1$$

for differentiable map $g(x)$. This condition ensures that $g(x)$ is a contraction mapping near the fixed point, meaning it pulls points closer together with each iteration.

- **Intuitive Explanation**: If $|g'(x^*)| < 1$, a small perturbation ϵ from the fixed point x^* will result in a smaller perturbation in the next iteration. This shrinking effect ensures that successive iterates get closer to the fixed point, leading to convergence.
- **Mathematical Explanation**:

  $$x_{n+1} \approx g(x^*) + g'(x^*)(x_n - x^*) = x^* + g'(x^*)(x_n - x^*)$$

  Denoting the error $e_n = x_n - x^*$:

  $$e_{n+1} \approx g'(x^*)e_n$$

  For $|g'(x^*)| < 1$, e_n converges to 0 as n increases, indicating convergence to x^*.

Lipschitz Condition

The Lipschitz condition is a more general criterion ensuring that the function $g(x)$ does not distort distances too much. Specifically, $g(x)$ is Lipschitz continuous if there exists a constant L such that:

$$|g(x) - g(y)| \leq L|x - y| \quad \text{for all } x, y.$$

When $L < 1$, $g(x)$ is a contraction mapping, guaranteeing that the fixed point iteration will converge to a unique fixed point. This can be linked to the derivative condition for differentiable functions:

$$|g'(x)| \leq L < 1.$$

Fixed point method on $[a, b]$

If $g([a, b]) \mapsto [a, b]$ satisfies the Lipschitz condition with $L < 1$ on interval $[a, b]$, then:

- **Uniqueness**: There is a unique fixed point in the interval $[a, b]$.
- **Convergence**: The iterative sequence $\{x_n\}$ will converge to this fixed point from any starting point within the interval $x_0 \in [a, b]$.

Example 2: Applying Fixed Point Iteration

Let us apply these concepts to an example:

Solve $f(x) = x^2 - 3x + 2 = 0$ using fixed point iteration.

1. **Rewrite the Equation**:
$$x = g(x) = \frac{x^2 + 2}{3}$$
2. **Check the Convergence Condition**:
$$g'(x) = \frac{2x}{3}$$
For convergence, we need $|g'(x)| < 1$. Evaluate this condition near the fixed point(s):
$$|g'(x)| = \left|\frac{2x}{3}\right|$$
For $|g'(x)| < 1$:
$$\left|\frac{2x}{3}\right| < 1 \implies |x| < \frac{3}{2}$$
3. **Verify Interval Mapping**:
To ensure an interval $[-\frac{3}{2}, \frac{3}{2}]$ is such that g maps $[-\frac{3}{2}, \frac{3}{2}]$ into itself. We need to verify that:
$$-\frac{3}{2} \leq g\left(-\frac{3}{2}\right) \leq \frac{3}{2}$$
Evaluating at the endpoints:
$$g\left(-\frac{3}{2}\right) = \frac{\left(-\frac{3}{2}\right)^2 + 2}{3} \approx 1.42$$

$$g\left(\frac{3}{2}\right) = \frac{\left(\frac{3}{2}\right)^2 + 2}{3} \approx 1.42$$

The minimum value at $x = 0$:

$$g(0) = \frac{0^2 + 2}{3} = \frac{2}{3} \approx 0.67$$

The maximum value at $x = \pm\frac{3}{2}$:

$$g\left(\pm\frac{3}{2}\right) \approx 1.42$$

Since all these values lie within $[-\frac{3}{2}, \frac{3}{2}]$, $g(x)$ maps the interval $[-\frac{3}{2}, \frac{3}{2}]$ into itself.

4. **Iterative Scheme**: Start with an initial guess x_0 within the interval $\left(-\frac{3}{2}, \frac{3}{2}\right)$ and iterate:

$$x_{n+1} = \frac{x_n^2 + 2}{3}$$

By ensuring $g(x)$ satisfies the Lipschitz condition with $L < 1$ and that the derivative $|g'(x)| < 1$ near the fixed point, we can guarantee the convergence of the fixed point iteration method.

Error Estimation using Lipschitz Constant

Let x^* be the fixed point such that $x^* = g(x^*)$. The error at the n-th iteration is defined as:

$$e_n = x_n - x^*$$

From the Lipschitz condition, we have:

$$|g(x) - g(y)| \leq L|x - y|$$

Applying this to the error:

$$|e_{n+1}| = |x_{n+1} - x^*| = |g(x_n) - g(x^*)| \leq L|x_n - x^*| = L|e_n|$$

This recursive relation shows that the error at each step is at most L times the error at the previous step. Therefore, we can write:

$$|e_{n+1}| \leq L|e_n|$$

We can use the estimate above to find an approximation of error of the method:

$$m > n \geq 1 : |x_m - x_n| = |x_m - x_{m-1} + x_{m-1} - x_{m-2} + \cdots + x_{n+1} - x_n|$$

$$\leq |x_m - x_{m-1}| + |x_{m-1} - x_{m-2}| + \cdots + |x_{n+1} - x_n|$$

$$\leq L^{m-1}|x_1 - x_0| + L^{m-2}|x_1 - x_0| + \cdots + L^n|x_1 - x_0|$$

$$= L^n|e_0| \left(L^{m-1-n} + L^{m-2-n} + \cdots + L + 1\right)$$

$$|x^* - x_n| = \lim_{m\to\infty} |x_m - x_n| \leq \lim_{m\to\infty} L^n|e_0| \sum_{i=0}^{m-1-n} L^i$$

$$= L^n|e_0| \sum_{i=0}^{\infty} L^i$$

$$= \frac{L^n}{1-L}|e_0|$$

$$|x_n - x^*| \leq \frac{L^n}{1-L}|e_0|$$

For a differentiable function g studied on interval $[a, b]$, the error bound can be expressed as:

$$|e_n| \leq \frac{|g'(\xi)|^n}{1-|g'(\xi)|}|e_0|$$

where $\xi \in [a, b]$ and for all $x \in [a, b]$ $|g'(\xi)| > |g'(x)|$. $|g'(\xi)|$ plays the role of the Lipschitz constant L. Meaning if $|g'(x)| \approx 0$ on $[a, b]$ the error is immediately negligible, and if $|g'(x)| \approx 1$ the error is shrinking slower.

Example 3:

Look at those three iterative functions $g(x) = 2^{-x}$, $g(x) = 2^{-x^2}$, $g(x) = 2^{-x^3}$. Look specifically at the slope of $g(x)$ at a fixed point and see how quick is the convergence of the fixed-point iterative method.

Solution:

1. $g(x) = 2^{-x}$
   - Fixed Point: $x^* \approx 0.6412$
   - Derivative at Fixed Point: $g'(x^*) \approx -0.4444$

2. $g(x) = 2^{-x^2}$
   - Fixed Point: $x^* \approx 0.7071$
   - Derivative at Fixed Point: $g'(x^*) \approx -0.6931$
3. $g(x) = 2^{-x^3}$
   - Fixed Point: $x^* \approx 0.7465$
   - Derivative at Fixed Point: $g'(x^*) \approx -0.8685$

Analysis of Convergence

The speed of convergence of the fixed-point iterative method is influenced by the magnitude of the derivative at the fixed point. The closer $|g'(x^*)|$ is to 0, the faster the convergence. Here are the results:

- For $g(x) = 2^{-x}$:
  $$|g'(0.6412)| \approx 0.4444$$
  This indicates relatively faster convergence.
- For $g(x) = 2^{-x^2}$:
  $$|g'(0.7071)| \approx 0.6931$$
  This indicates moderate convergence.
- For $g(x) = 2^{-x^3}$:
  $$|g'(0.7481)| \approx 0.8706$$
  This indicates slower convergence.

4.3 Newton method

4.3.1 Motivation

In Example 1, section 4.2.1, we saw that probably the easiest way to get iterative function g that allows use to solve $f(x) = 0$ is a $g(x) = x - f(x)$. But the fixed-point method with $g(x) = x - f(x)$ may not always be convergent. Therefore let us modify $g(x)$. Assume the iterative function g in the following form $g(x) = x - \frac{f(x)}{K}$ on interval I. Our goal is to select K so that for some L: (i) $|g'(x)| < 1 \; \forall x \in I$ to ensure that the method is convergent, and (ii) $|g'(x)|$ is close to zero to speed up the convergence. Consider a computation of $\sqrt{a}$ using $f(x) = x^2 - a \quad \Rightarrow \quad g(x) = x - \frac{x^2-a}{K}$, for example (see Example 4). You will find that K needs to be close to $f'(x^*)$. This ensures that $g'(x) = 1 - \frac{f'(x)}{K}$ is close to 0 near x^*. If we work on this ideas further we can use an integrative function g in the following form:

$$g(x) = x - \frac{f(x)}{h(x)}$$

$$g'(x) = 1 - \frac{f'(x)h(x) - f(x)h'(x)}{h^2(x)}$$

For $h(x) = f'(x)$, assuming that $f''(x)$ exists:

$$g'(x) = \frac{f(x)f''(x)}{(f'(x))^2}.$$

Generally, since $f(x)$ is close to zero, $g'(x)$ is close to zero. But that is not always the case. When searching, for example, for multiple roots, not only $f(x)$ is close to zero near the root, but also $f'(x)$, so $g'(x)$ may not be close to zero.

4.3.2 Newton Method for a Single Nonlinear Equation

Given a function $f(x)$, the Newton method is given by formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

where:

- x_n is the current approximation,
- $f(x_n)$ is the function value at x_n,
- $f'(x_n)$ is the derivative of the function at x_n.

The process is repeated until the difference between successive approximations is smaller than a given tolerance.

Geometric Description of Newton Method

Newton method finds successively better approximations to the root of a function by leveraging the tangent line at the current approximation. Therefore Newton method is somethimes called Tangent method.

1. **Starting Point**: Begin with an initial guess x_0.
2. **Tangent Line**: At x_0, compute the function value $f(x_0)$ and the derivative $f'(x_0)$. The tangent line to the curve at this point is given by:

$$y = f(x_0) + f'(x_0)(x - x_0)$$

3. **Intersection with x-axis**: The next approximation x_1 is found where this tangent line intersects the x-axis. Set $y = 0$ and solve for x:

$$0 = f(x_0) + f'(x_0)(x_1 - x_0)$$

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

4. **Iterate**: Repeat the process with x_1 to find x_2, and so on, until the approximations converge to the root.

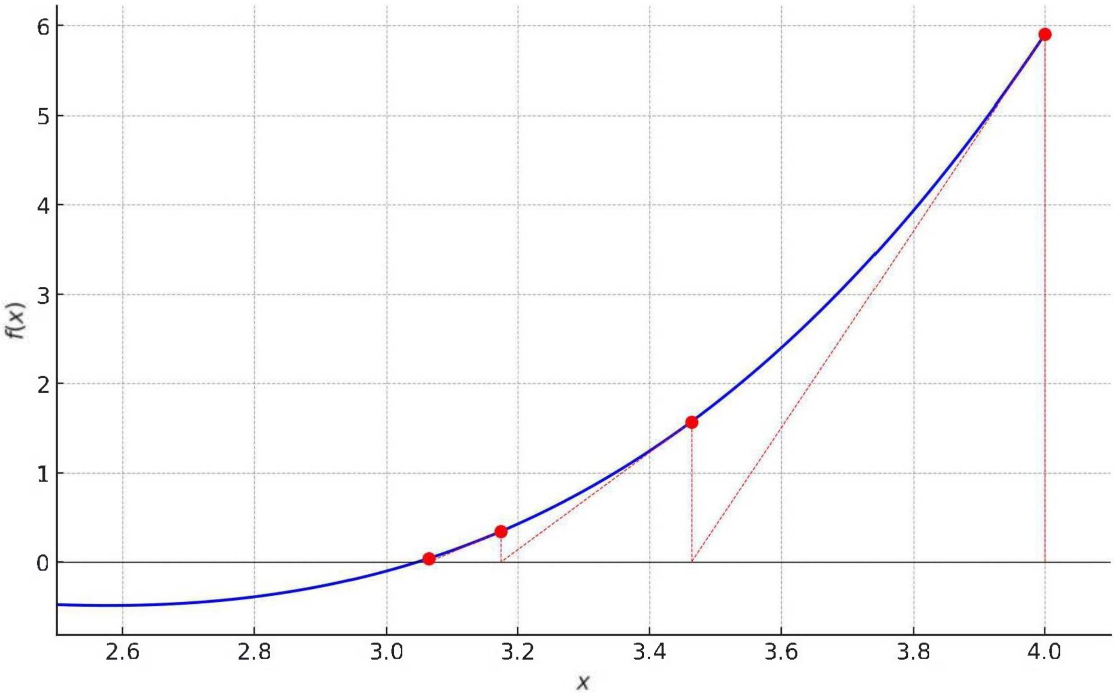


Fig. 4.1: Graphical visualization of the Newton method.

4.3.3 Newton Method for Systems of Nonlinear Equations

For a system of n nonlinear equations, $\mathbf{F}(\mathbf{x}) = \mathbf{0}$, where $\mathbf{F}$ is a vector-valued function and $\mathbf{x}$ is a vector of n variables, Newton method is derived using the first-order Taylor expansion of each equation around a current guess $\mathbf{x}^{(k)}$.

Consider the system of nonlinear equations:

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} f_1(x_1, x_2, \ldots, x_n) \\ f_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ f_n(x_1, x_2, \ldots, x_n) \end{bmatrix} = \mathbf{0}.$$

Each function $f_i(\mathbf{x})$ can be approximated using a first-order Taylor expansion around the current iterate $\mathbf{x}^{(k)}$. For each $i = 1, \ldots, n$, we have:

$$f_i(\mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k)}) \approx f_i(\mathbf{x}^{(k)}) + \sum_{j=1}^{n} \frac{\partial f_i}{\partial x_j}(\mathbf{x}^{(k)})\Delta x_j^{(k)}.$$

Explicitly, the system of equations becomes:

$$\begin{aligned}
f_1(\mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k)}) &\approx f_1(\mathbf{x}^{(k)}) + \frac{\partial f_1}{\partial x_1}(\mathbf{x}^{(k)})\Delta x_1^{(k)} + \cdots + \frac{\partial f_1}{\partial x_n}(\mathbf{x}^{(k)})\Delta x_n^{(k)},\\
f_2(\mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k)}) &\approx f_2(\mathbf{x}^{(k)}) + \frac{\partial f_2}{\partial x_1}(\mathbf{x}^{(k)})\Delta x_1^{(k)} + \cdots + \frac{\partial f_2}{\partial x_n}(\mathbf{x}^{(k)})\Delta x_n^{(k)},\\
&\vdots\\
f_n(\mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k)}) &\approx f_n(\mathbf{x}^{(k)}) + \frac{\partial f_n}{\partial x_1}(\mathbf{x}^{(k)})\Delta x_1^{(k)} + \cdots + \frac{\partial f_n}{\partial x_n}(\mathbf{x}^{(k)})\Delta x_n^{(k)}.
\end{aligned}$$

In matrix form, the system can be written using the Jacobian matrix $J(\mathbf{x}^{(k)})$ as:

$$\mathbf{F}(\mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k)}) \approx \mathbf{F}(\mathbf{x}^{(k)}) + J(\mathbf{x}^{(k)})\Delta\mathbf{x}^{(k)},$$

where the Jacobian matrix $J(\mathbf{x})$ is:

$$J(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}.$$

To find the next iterate $\mathbf{x}^{(k+1)}$, we set $\mathbf{F}(\mathbf{x}^{(k+1)}) = \mathbf{0}$. Using the Taylor expansion:

$$\mathbf{0} \approx \mathbf{F}(\mathbf{x}^{(k)}) + J(\mathbf{x}^{(k)})\Delta\mathbf{x}^{(k)}.$$

Rearranging, we get the linear system:

$$J(\mathbf{x}^{(k)})\Delta\mathbf{x}^{(k)} = -\mathbf{F}(\mathbf{x}^{(k)}),$$

which is solved for the correction $\Delta\mathbf{x}^{(k)}$.

Steps of the Newton Method for Systems

1. **Initial Guess**: Start with an initial guess $\mathbf{x}_0$.
2. **Linear System Solution**: At each iteration k, solve the linear system:

   $$J(\mathbf{x}_k)\Delta\mathbf{x}_k = -\mathbf{F}(\mathbf{x}_k)$$

   for the correction $\Delta\mathbf{x}_k$.
3. **Update the Solution**: Update the solution:

   $$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}_k$$

4. **Convergence Check**: The process is repeated until $\|\Delta\mathbf{x}^{(k)}\|$ becomes sufficiently small, ensuring convergence to the solution $\hat{\mathbf{x}}$.

Geometric description of the method

The Jacobian matrix $J(\mathbf{x})$ is constructed using the gradients of the functions f_i. Each row of the Jacobian matrix is the gradient of the corresponding function f_i:

$$J(\mathbf{x}) = \begin{bmatrix} \nabla f_1(\mathbf{x}) \\ \nabla f_2(\mathbf{x}) \\ \vdots \\ \nabla f_n(\mathbf{x}) \end{bmatrix}$$

The gradient of a function f_i is a vector of its partial derivatives with respect to each variable x_j:

$$\nabla f_i(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_i}{\partial x_1} & \frac{\partial f_i}{\partial x_2} & \cdots & \frac{\partial f_i}{\partial x_n} \end{bmatrix}$$

The gradients in the Jacobian matrix provide a linear approximation of the functions f_i around the current guess $\mathbf{x}^{(k)}$. This approximation helps in solving the system of linear equations for the update step. Graphically the solution of the linear system in update step is intersection of tangent planes to graph to each function f_i and plane $f(\mathbf{x}) = 0$.

Example 5: Newton Method for the System of Nonlinear Equations

We aim to solve the following system of nonlinear equations using Newton method:

$$\begin{aligned} x_2^2 - x_1 + 1 &= 0 \\ x_1^2 + x_2^2 - 2x_1 &= 0 \end{aligned}$$

Step 1: Define the System of Equations and the Jacobian Matrix

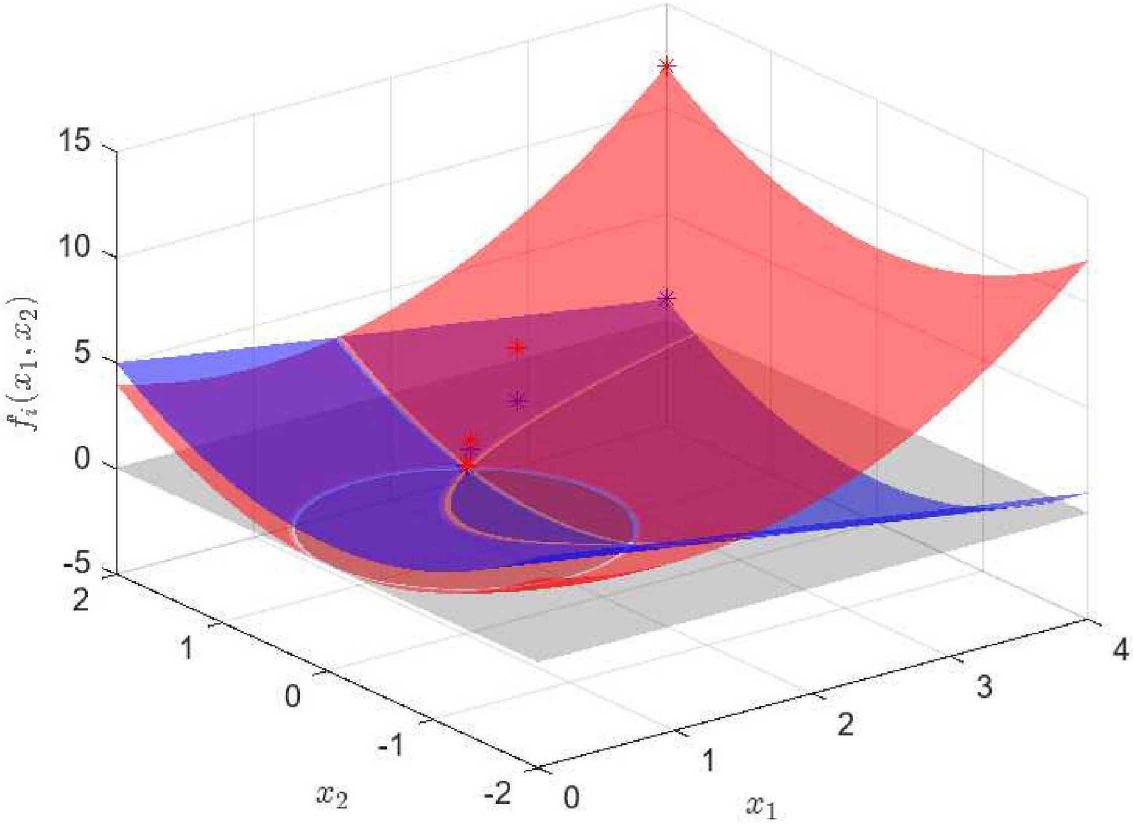
We represent the system in vector form as follows:

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{bmatrix} = \begin{bmatrix} x_2^2 - x_1 + 1 \\ x_1^2 + x_2^2 - 2x_1 \end{bmatrix}$$

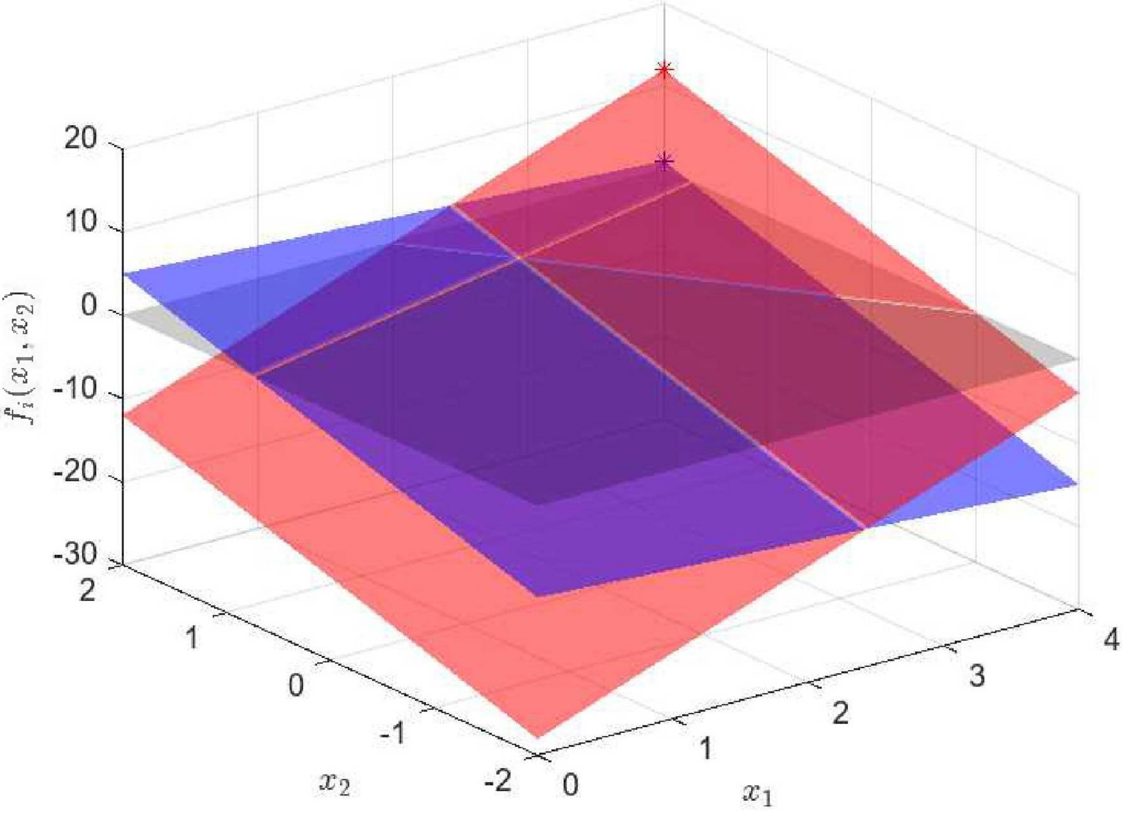
Newton method for systems of nonlinear equations requires the Jacobian matrix, which is composed of the partial derivatives of the functions $f_1(x_1, x_2)$ and $f_2(x_1, x_2)$. The Jacobian matrix $J(\mathbf{x})$ for this system is given by:

$$J(\mathbf{x}) = \begin{bmatrix} -\frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} -1 & 2x_2 \\ 2x_1 - 2 & 2x_2 \end{bmatrix}$$

This matrix will be used in each iteration to compute the Newton step, $\Delta\mathbf{x}^{(k)}$.



(a) Visualization of iterations.



(b) Tangent planes for each functions f_i in $\mathbf{x}_0$ and their intersection with plane $f(\mathbf{x}) = 0$.

Fig. 4.2: Newton method for systems of equations, see Example 5

Step 2: Iterative Process Using Newton Method

First Iteration:

We start with an initial guess $\mathbf{x}^{(0)} = [4, 2]^\top$.

1. **Evaluate the function $\mathbf{F}(\mathbf{x}^{(0)})$**:

$$\mathbf{F}(\mathbf{x}^{(0)}) = \begin{bmatrix} 2^2 - 4 + 1 \\ 4^2 + 2^2 - 2 \cdot 4 \end{bmatrix} = \begin{bmatrix} 1 \\ 12 \end{bmatrix}$$

2. **Evaluate the Jacobian $J(\mathbf{x}^{(0)})$** at $\mathbf{x}^{(0)} = [4, 2]^\top$:

$$J(\mathbf{x}^{(0)}) = \begin{bmatrix} -1 & 2 \cdot 2 \\ 2 \cdot 4 - 2 & 2 \cdot 2 \end{bmatrix} = \begin{bmatrix} -1 & 4 \\ 6 & 4 \end{bmatrix}$$

3. **Solve for $\Delta\mathbf{x}^{(0)}$** by solving the linear system:

$$J(\mathbf{x}^{(0)})\Delta\mathbf{x}^{(0)} = -\mathbf{F}(\mathbf{x}^{(0)})$$

$$\begin{bmatrix} -1 & 4 \\ 6 & 4 \end{bmatrix} \begin{bmatrix} \Delta x_1^{(0)} \\ \Delta x_2^{(0)} \end{bmatrix} = \begin{bmatrix} -1 \\ -12 \end{bmatrix}$$

Solving this system results in:

$$\Delta x_1^{(0)} = -1.5714, \quad \Delta x_2^{(0)} = -0.6429$$

4. **Update the solution**:

$$\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \Delta\mathbf{x}^{(0)} = \begin{bmatrix} 4 \\ 2 \end{bmatrix} + \begin{bmatrix} -1.5714 \\ -0.6429 \end{bmatrix} = \begin{bmatrix} 2.4286 \\ 1.3571 \end{bmatrix}$$

The new approximation for the solution after the first iteration is $\mathbf{x}^{(1)} = [2.4286, 1.3571]^\top$.

By repeating this iterative process, updating the solution at each step, the Newton method will converge towards a solution. In this case, after several iterations, the values of $\mathbf{x}$ will tend to:

$$\mathbf{x}^* = \begin{bmatrix} 1.6180 \\ 0.7862 \end{bmatrix}$$

This is an approximation of the root of the system of nonlinear equations. As the number of iterations increases, the solution becomes more accurate, with the values of x_1 and x_2 converging to 1.6180 and 0.7862, respectively. Figure 4.2 visualize the iterative process.

4.3.4 Numerical Continuation Using One Parameter

Numerical continuation is a method used to track the solutions of a system of equations as a parameter varies. It can be seen as a predictor-corrector method because it employs a two-step iterative process to approximate the solution curve of a system. In the prediction step, an initial guess of the new solution is made using extrapolation techniques, often based on the tangent or higher-order derivatives of the solution curve. This guess is typically quite rough but provides a starting point close to the actual solution. In the correction step, this initial guess is refined by applying Newton method or other root-finding algorithms to ensure that it satisfies the underlying equations more precisely. By alternating between these prediction and correction steps, numerical continuation efficiently tracks the solution path through parameter values.

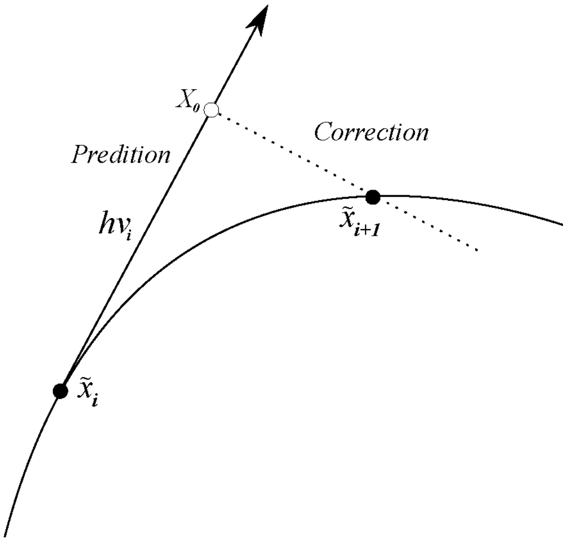


Fig. 4.3: Graphical visualization of the numerical continuation process.

Numerical continuation is just one example of a common principal of numerical mathematics call the predictor-corrector method. The motivation behind this method lies in its ability to balance efficiency and accuracy. The prediction step offers a quick, computationally inexpensive estimate of the solution. This step provides a provisional solution that, while not exact, is sufficiently close to the true solution to make the next step more effective. The correction step then refines this estimate, using more precise methods such as Newton. This iterative refinement ensures that the final solution is accurate, conforming closely to the desired tolerance. By combining these two steps, predictor-corrector methods harness the strengths of both: the speed of simple and the accuracy of more complex. Another field of numerical mathematics where this principal is particularly valuable is solving ordinary differential equations (ODEs).

4.3.4.1 Natural Continuation

In natural continuation, one parameter λ in the system $\mathbf{F}(\mathbf{x}, \lambda) = \mathbf{0}$ is varied, and the system is solved iteratively for each value of λ.

Steps of the Natural Continuation

1. Start with a known solution $(\mathbf{x}_0, \lambda_0)$.
2. Increment the parameter λ by a small step $\Delta\lambda$.
3. Use Newton method to solve $\mathbf{F}(\mathbf{x}, \lambda + \Delta\lambda) = \mathbf{0}$ starting from the previous solution $\mathbf{x}_{\text{prev}}$.

4.3.4.2 Pseudo-arclength Continuation

Pseudo-arclength continuation allows continuation past limit points, cusp point, or more generally singular points by parameterizing the solution path with an arc-length parameter s. A limit point is identified at the folding point of a curve. At this point, the curve changes direction, indicating a sharp transition or singularity in the behavior of the function. A cusp point on a curve is a point where the curve has a sharp point, i.e., where the curve does not have a well-defined tangent.

Steps of the Pseudo-arclength Continuation

1. Define an augmented system that includes the original equations and an additional constraint to fix the arc length.
2. Solve the augmented system using Newton method:
$$\begin{cases} \mathbf{F}(\mathbf{x}, \lambda) = \mathbf{0} \\ \mathbf{v}^T(\mathbf{x} - \mathbf{x}_{\text{prev}}, \lambda - \lambda_{\text{prev}}) - \Delta s = 0 \end{cases}$$
   where $\mathbf{v}$ is a tangent vector, and Δs is the step in arc length.

Example 6

The success of continuation methods depends on the nature of the equations and the singularities encountered:

1. For $F(x, \lambda) = 3x^2 + \lambda$, natural continuation fails at the fold $(0, 0)$, whereas pseudo-arclength continuation succeeds.
2. For $F(x, \lambda) = x^3 + 4\lambda x + \lambda^2$, both natural and pseudo-arclength continuation struggle at the branch point $(0, 0)$.

To avoid problems described above we may evaluate functions on a grid (usually mathematical software do so while producing implicit plots) and find contour at zero-level set. This method may be computationally inefficient specifically it the evaluations of desirable function is costly. On the other hand both singularities and branch points may cause difficulties while performing continuation.

4.3.5 Gauss-Newton Method

The Gauss-Newton method is used to solve nonlinear least squares problems, particularly when dealing with over-determined systems. For a system of equations $\mathbf{F}(\mathbf{x})$ where $\mathbf{F}$ represents residuals, the objective is to minimize $||\mathbf{F}(\mathbf{x})||^2$.

1. Start with an initial guess $\mathbf{x}_0$.
2. At each iteration, solve the linear least squares problem:

   $$J(\mathbf{x}_n)^T J(\mathbf{x}_n)\Delta\mathbf{x}_n = -J(\mathbf{x}_n)^T \mathbf{F}(\mathbf{x}_n)$$

   where $J(\mathbf{x}_n)$ is the Jacobian matrix of $\mathbf{F}$ at $\mathbf{x}_n$.
3. Update the solution: $\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta\mathbf{x}_n$.
4. Repeat until convergence.

We can express an integration of the Gauss-Newton method using the left pseudoinverse of $\mathbf{J}(\mathbf{x}_k)$:

$$\Delta\mathbf{x}_k = -\mathbf{J}(\mathbf{x}_k)^+\mathbf{r}(\mathbf{x}_k)$$

Here, $\mathbf{J}(\mathbf{x}_k)^+$. denotes the Moore-Penrose pseudoinverse of $\mathbf{J}(\mathbf{x}_k)$. For the left pseudoinverse specifically:

$$\mathbf{J}(\mathbf{x}_k)^+ = (\mathbf{J}(\mathbf{x}_k)^T\mathbf{J}(\mathbf{x}_k))^{-1}\mathbf{J}(\mathbf{x}_k)^T.$$

Example 7: Logistic Growth Model for human population

We have historical world population data and we want to fit the logistic growth model to this data to estimate the parameters K, y_0, and r. The logistic growth model is given by:

$$y(t) = \frac{K}{1 + \frac{K - y_0}{y_0} e^{-rt}}$$

where $y(t)$ is the population at time t, K is the carrying capacity, y_0 is the initial population size, r is the growth rate. Let us assume we have the following world population data (in billions):

Year	Population
1950	2.53
1960	3.03
1970	3.70
1980	4.45
1990	5.32
2000	6.12
2010	6.92
2020	7.80

Solution

The residuals r_i are the differences between the observed data points and the model predictions:

$$r_i = y_i - \frac{K}{1 + \frac{K - y_0}{y_0} e^{-rt_i}}$$

Compute the Jacobian matrix $\mathbf{J}$ of the residuals with respect to the parameters K, y_0, and r: $\mathbf{J}_{ij} = \frac{\partial r_i}{\partial \theta_j}$, where $\theta = [K, y_0, r]^T$. Start with an initial guess for the parameters, say $K_0 = 10$, $y_0 = 2.5$, and $r_0 = 0.02$. Update the parameters using the Gauss-Newton iteration:

$$\theta_{k+1} = \theta_k - (\mathbf{J}(\theta_k)^T \mathbf{J}(\theta_k))^{-1} \mathbf{J}(\theta_k)^T \mathbf{r}(\theta_k).$$

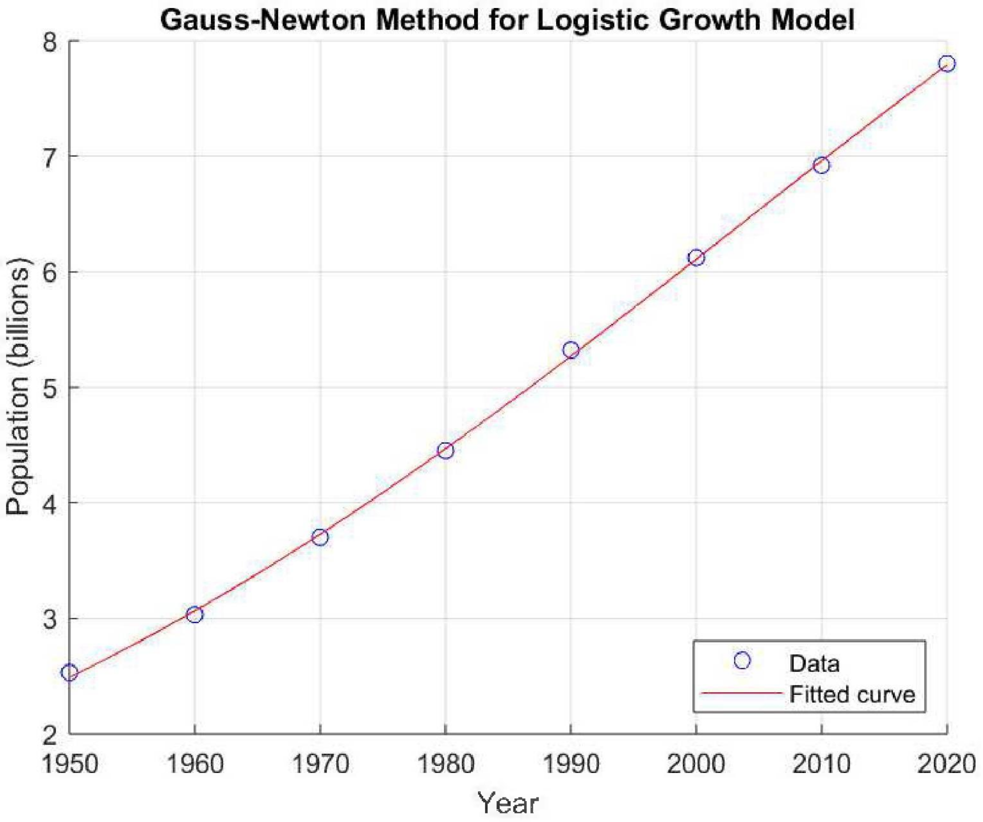


Fig. 4.4: Logistic grow of the human population. The estimated parameters are $K = 12.8087$, $y_0 = 2.488$, $r = 0.026599$.

4.4 Methods derived from Newton method

4.4.1 Secant Method

The Secant Method is a root-finding algorithm that, unlike Newton Method, does not require the derivative of the function. Instead, it uses a finite difference approximation to the derivative. Given two initial approximations x_0 and x_1, the iteration is:

$$x_{n+1} = x_n - f(x_n) \cdot \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

The Secant Method approximates the derivative $f'(x)$ as:

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

4.4.2 Regula Falsi Method

The Regula Falsi Method (or False Position Method) is another iterative method to find roots. It combines the ideas of the bisection method and the secant method. It guarantees that the root remains bracketed within an interval $[a, b]$. The iteration is defined by:

$$x_{n+1} = b - f(b) \cdot \frac{b-a}{f(b) - f(a)}$$

Here, a and b are such that $f(a)$ and $f(b)$ have opposite signs (i.e., $f(a)f(b) < 0$), ensuring a root exists in the interval $[a, b]$. In each iteration we change a or b to x_{n+1} and preserve property $f(a)f(b) < 0$.

The Regula Falsi (or False Position) method shares similarities with both the Secant Method and the Bisection Method.

Connection to the Secant Method

The Regula Falsi Method can be seen as a modified version of the Secant Method where the interval $[a, b]$ is dynamically updated to ensure that the root is always bracketed. Specifically:

- If $f(x_{n+1})$ and $f(b)$ have opposite signs, set $a = x_{n+1}$ and keep b unchanged.
- Otherwise, set $b = x_{n+1}$ and keep a unchanged.

This ensures that the root always lies between a and b, improving the robustness of the method compared to the Secant Method.

Connection to the Bisection Method

The key connection lies in how the interval is updated to ensure the root remains bracketed.

Bisection Method

The Bisection Method is a simple root-finding algorithm that relies on the Intermediate Value Theorem. Given a continuous function $f(x)$ on an interval $[a, b]$ where $f(a)f(b) < 0$, the method iteratively halves the interval until it converges to the root. The iteration is:

1. Compute the midpoint: $c = \frac{a+b}{2}$.
2. Evaluate $f(c)$.

3. Update the interval:
   - If $f(a)f(c) < 0$, set $b = c$.
   - Otherwise, set $a = c$.

The Regula Falsi method uses a linear interpolation, potentially finding the root faster while still ensuring the root is bracketed, improving on the Bisection Method's interval halving approach.

4.4.3 Quasi-Newton Methods for Multiple Variables

Quasi-Newton methods approximate the Jacobian matrix (see section 4.3.3) rather than computing it exactly. From that perspective Secant method can be seen as a special example of Quasi-Newton for single variable. One of the popular Quasi-Newton methods for multiple variables is Broyden method.

Broyden Method

Broyden method updates an approximate Jacobian $\mathbf{B}_n$ at each iteration. The iteration for Broyden method is:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{B}_n^{-1}\mathbf{F}(\mathbf{x}_n)$$

After updating $\mathbf{x}$, the Jacobian approximation $\mathbf{B}$ is updated using the formula:

$$\mathbf{B}_{n+1} = \mathbf{B}_n + \frac{(\Delta\mathbf{F}_n - \mathbf{B}_n\Delta\mathbf{x}_n)\Delta\mathbf{x}_n^T}{\Delta\mathbf{x}_n^T\Delta\mathbf{x}_n}$$

where: $\Delta\mathbf{x}_n = \mathbf{x}_{n+1} - \mathbf{x}_n$, $\Delta\mathbf{F}_n = \mathbf{F}(\mathbf{x}_{n+1}) - \mathbf{F}(\mathbf{x}_n)$

Explanation of the formula for $\mathbf{B}_{n+1}$

- Difference in Function Values: $\Delta\mathbf{F}_k$ represents how much the function values have changed after taking the step $\Delta\mathbf{x}_k$.
- Current Approximation Error: $\mathbf{B}_k\Delta\mathbf{x}_k$ is the current Jacobian's estimate of the function change. The difference $\Delta\mathbf{F}_k - \mathbf{B}_k\Delta\mathbf{x}_k$ indicates the error in the Jacobian's prediction.
- Rank-One Update: The term $\frac{(\Delta\mathbf{F}_k - \mathbf{B}_k\Delta\mathbf{x}_k)\Delta\mathbf{x}_k^T}{\Delta\mathbf{x}_k^T\Delta\mathbf{x}_k}$ represents a rank-one correction to the approximate Jacobian $\mathbf{B}_k$. This correction adjusts the Jacobian to better match the observed change in the function values.

- Normalization: The denominator $\Delta\mathbf{x}_k^T\Delta\mathbf{x}_k$ normalizes the update to ensure it is appropriately scaled.

Algorithm Steps

1. **Initialization**:
   - Choose an initial guess $\mathbf{x}_0$.
   - Compute $\mathbf{F}(\mathbf{x}_0)$ and an initial approximation $\mathbf{B}_0$ to the Jacobian matrix.
2. **Iteration**:
   - Solve $\mathbf{B}_n\Delta\mathbf{x}_n = -\mathbf{F}(\mathbf{x}_n)$ for $\Delta\mathbf{x}_n$.
   - Update $\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta\mathbf{x}_n$.
   - Compute $\mathbf{F}(\mathbf{x}_{n+1})$.
   - Update the Jacobian approximation $\mathbf{B}_{n+1}$ using the formula above.
   - Check for convergence (e.g., if $\|\mathbf{F}(\mathbf{x}_{n+1})\|$ is sufficiently small).

4.5 Rate of Convergence

The rate of convergence describes how quickly a sequence approaches its limit as the number of terms increases. It is a measure of the speed at which the terms of the sequence get closer to the limit. All the definition bellow all formulated for single nonlinear equation, equivalently we may proceed for systems of nonlinear equations, while instead of absolute values we are going to apply an arbitrary vector norms.

Definition: Given a sequence (x_n) that converges to a limit L, the rate of convergence provides a quantitative description of how fast x_n approaches L as $n \to \infty$.

The rate of convergence is often classified into different categories, depending on the relationship between the sequence and its limit. Here are some common types:

Sublinear Convergence: A sequence (x_n) is said to converge sublinearly to a limit L if:

$$\lim_{n\to\infty} \frac{|x_{n+1} - L|}{|x_n - L|} = 1$$

This means that the sequence converges to L more slowly than any linear rate. The ratio of successive errors approaches 1 as n goes to infinity, indicating very slow convergence.

Linear Convergence: If there exists a constant C (with $0 < C < 1$) such that:

$$|x_{n+1} - L| \le C|x_n - L|$$

for sufficiently large n, then the sequence (x_n) is said to converge linearly to L. Linear convergence implies that the error $|x_n - L|$ decreases by a constant proportion in each iteration.

Superlinear Convergence: If the sequence (x_n) converges faster than linearly but not as fast as quadratic convergence, it is said to converge superlinearly. Formally, if:

$$\lim_{n\to\infty} \frac{|x_{n+1} - L|}{|x_n - L|} = 0$$

then the sequence converges superlinearly.

Quadratic Convergence: If there exists a constant C (with $C > 0$) such that:

$$|x_{n+1} - L| \leq C|x_n - L|^2$$

for sufficiently large n, then the sequence (x_n) is said to converge quadratically to L. Quadratic convergence implies that the error $|x_n - L|$ squares in each iteration, leading to very rapid convergence.

Convergence with Order p: A sequence (x_n) converges to L with order p (where $p > 1$) if there exists a constant $C > 0$ such that:

$$|x_{n+1} - L| \leq C|x_n - L|^p$$

for sufficiently large n. This means that the sequence converges faster than quadratically if $p > 2$.

Theorem: Let $g : \mathbb{R} \to \mathbb{R}$ be a continuously differentiable function, and let x^* be a fixed point of g (i.e., $g(x^*) = x^*$). If the sequence $\{x_n\}$ is defined by the iteration $x_{n+1} = g(x_n)$, then:

1. If $|g'(x^*)| < 1$, the iteration converges to x^* with a linear rate of convergence.
2. If $g'(x^*) = 0$ and $|g''(x^*)| \neq 0$, the iteration converges to x^* with a quadratic rate of convergence.
3. More generally, if $g^{(k)}(x^*) = 0$ for $k = 1, 2, \ldots, p-1$ and $g^{(p)}(x^*) \neq 0$, the iteration converges to x^* with a rate of convergence p.

Proof: The proof involves analyzing the behavior of the iterates near the fixed point x^*.

Linear Convergence: If $|g'(x^*)| < 1$, then there exists a neighborhood around x^* such that $|g'(x)| < 1$ for all x in that neighborhood. By the mean value theorem, for x_n close to x^*:

$$|x_{n+1} - x^*| = |g(x_n) - g(x^*)| = |g'(c_n)(x_n - x^*)| < |x_n - x^*|$$

where c_n is some point between x_n and x^*. This implies linear convergence.

Quadratic Convergence If $g'(x^*) = 0$ and $g''(x^*) \neq 0$, we use a second-order Taylor expansion of g around x^*:

$$g(x) = x^* + \frac{1}{2}g''(x^*)(x - x^*)^2 + O((x - x^*)^3)$$

Thus,

$$x_{n+1} - x^* = g(x_n) - x^* = \frac{1}{2}g''(x^*)(x_n - x^*)^2 + O((x_n - x^*)^3)$$

Hence, $|x_{n+1} - x^*| \approx C|x_n - x^*|^2$, indicating quadratic convergence. The genral formula can be proven similarly. □

The following table gives an overview of rates of convergence for commonly used methods. To achieve maximal rates as are written in the table, other conditions may need to be fulfilled, but in practical applications are not usually checked.

Method	**Rate of Convergence**
Newton Method	Quadratic (2)
Secant Method	Superlinear (≈ 1.618)
Bisection Method	Linear (1)
Regula-Falsi Method	Linear (1)
Broyden Method	Superlinear (≈ 1.5)

Example 8

Find a root of $x^3 - 10 = 0$ using (a) Regula-falsi (R-F) method on interval $[1, 3]$, (b) Newton method with $x_0 = 4$. For both methods generate sequences e_k. Plot a dependence e_{k+1} on e_k and give it into the context of convergence rates of iterative methods.

Regula-Falsi method - Iteration Formula

$$x_0 = a, x_1 = b, x_{k+1} = x_k - \frac{f(x_k)(x_k - x_s)}{f(x_k) - f(x_s)} \text{ and } f(x_k)f(x_s) < 0 \text{ for } s \in \{k, k-1\}$$

Iterations

$$\begin{aligned}
&\text{Iteration 1:} \quad x_0 = 1, \quad x_1 = 3 \\
&\text{Iteration 2:} \quad x_2 = 3 - \frac{f(3)(3-1)}{f(3) - f(1)} = 3 - \frac{17 \cdot 2}{17 - (-9)} = 3 - \frac{34}{26} \approx 1.6923077 \\
&\text{Iteration 3:} \quad x_3 = 1.6923077 - \frac{f(1.6923077)(1.6923077 - 3)}{f(1.6923077) - f(3)} \approx 1.9965072 \\
&\qquad\qquad\quad \vdots
\end{aligned}$$

Newton Method - Iteration Formula

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

Iterations

$$\begin{aligned} \text{Iteration 1:} \quad & x_1 = 4 - \frac{4^3 - 10}{3 \cdot 4^2} = 4 - \frac{54}{48} \approx 2.875000 \\ \text{Iteration 2:} \quad & x_2 = 2.875 - \frac{2.875^3 - 10}{3 \cdot 2.875^2} \approx 2.319943 \\ & \vdots \end{aligned}$$

Error Sequence Let $x^* = \sqrt[3]{10} \approx 2.1544$. The error sequence $e_k = |x_k - x^*|$ is computed for each iteration.

k	x_k		e_k	
	R-F	Newton	R-F	Newton
0	1.000000	4.000000	1.154435	1.845565
1	3.000000	2.875000	0.845565	0.720565
2	1.692308	2.319943	0.462127	0.165509
3	1.996507	2.165962	0.157928	0.011527
4	2.104111	2.154496	0.050324	0.000061
5	2.167187	2.154435	0.012753	0.000000
6	2.154133	2.154435	0.000301	0.000000
7	2.154433	2.154435	0.000002	0.000000

Convergence Rate Analysis

- **Regula-Falsi Method:** This method typically exhibits linear convergence. The plot of e_{k+1} vs. e_k should approximate a line with a slope less than 1.
- **Newton Method:** This method typically exhibits quadratic convergence. The plot of e_{k+1} vs. e_k should approximate a quadratic function.

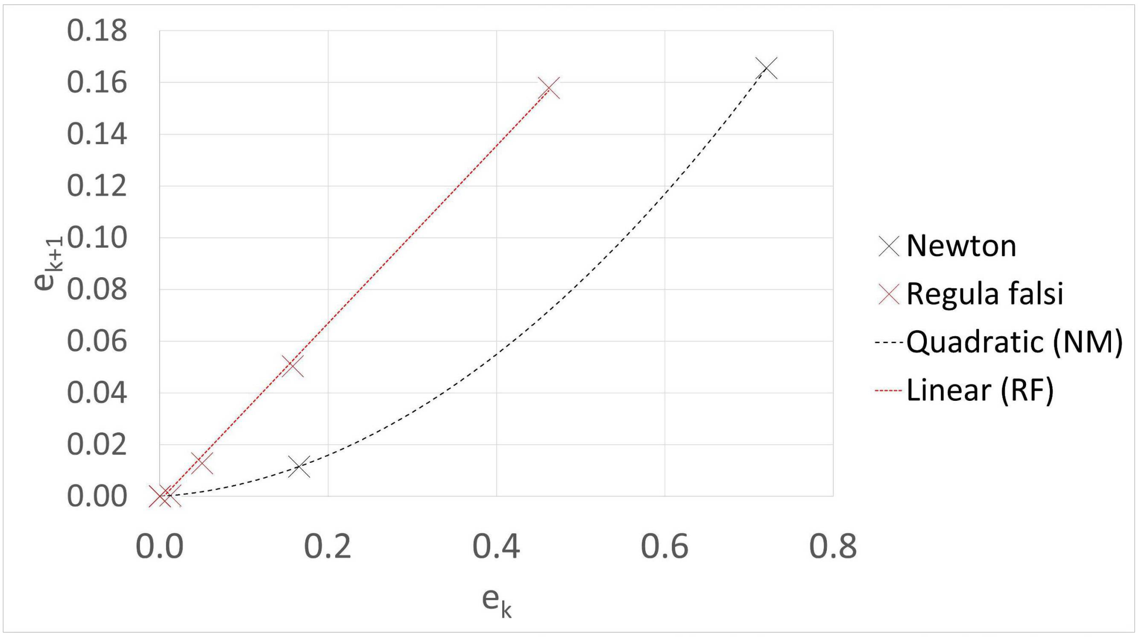


Fig. 4.5: Dependence of e_{k+1} on e_k and fits of corresponding curves (linear function for Regula-falsi method, Quadratic function for Newton method) using least square method.

Example 9

Find a non-trivial fixed point for logistic map for $a = 1.5, 2.5, 1, 3$ using fixed point iterative method. Plot a dependence e_{k+1} on e_k and give it into the context of convergence rates of iterative methods. Plot a dependence of $\frac{|e_{k+1}|}{|e_k|}$ on k and give it into the context of convergence rates of iterative methods.

Solution:

For the logistic map $x_{n+1} = ax_n(1 - x_n)$:

For $a = 1.5$ and $a = 2.5$:

- The sequences generated by the fixed point iteration method show linear convergence.
- The plots of $|e_{k+1}|$ vs. e_k display a linear relationship with a slope less than 1, indicating that the error decreases steadily at a linear rate.

For $a = 1$:

- There exists only trivial fixed point ($x^* = 0$), and the sequences converge to zero.
- The convergence is sublinear, as seen from the plot of $\frac{|e_{k+1}|}{|e_k|}$, which tends to 1, indicating a slower rate of error reduction compared to linear convergence.

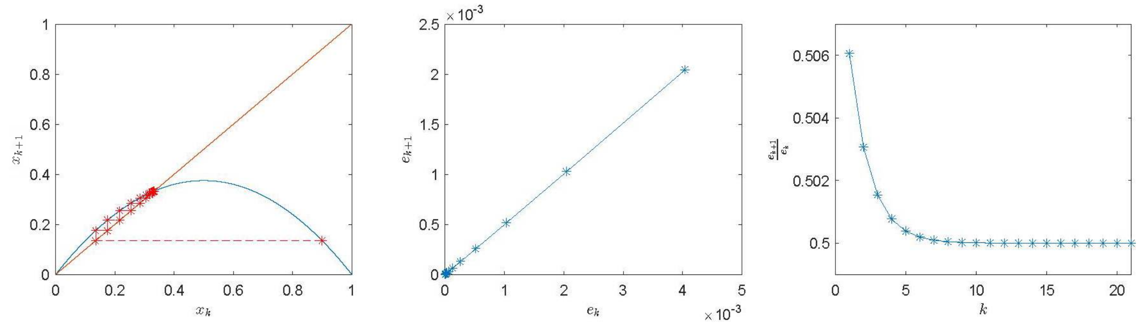


Fig. 4.6: Fixed point for logistic map for $a = 1.5$, terminating condition $|x_k - x^*| < 10^{-9}$.

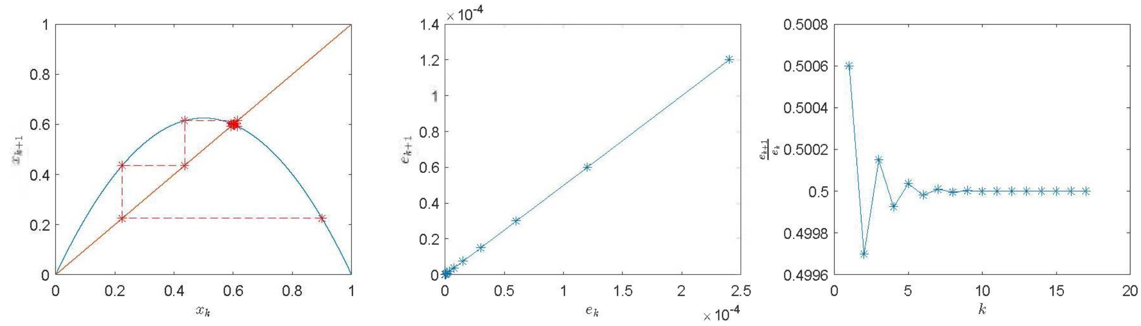


Fig. 4.7: Fixed point for logistic map for $a = 2.5$, terminating condition $|x_k - x^*| < 10^{-9}$.

For $a = 3$:

- The sequences show sublinear convergence.
- The plot of $\frac{|e_{k+1}|}{|e_k|}$ tends to 1, indicating a slower convergence rate, characterized by a gradual reduction in error over iterations.

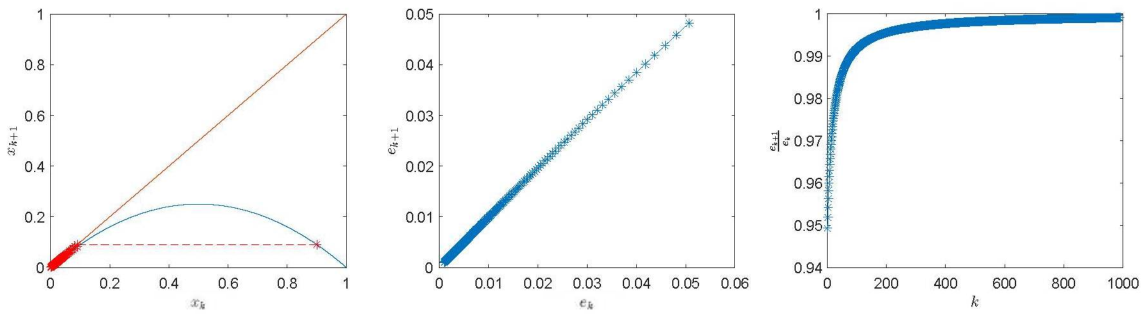


Fig. 4.8: Fixed point for logistic map for $a = 1$, terminating condition $|x_k - x^*| < 10^{-9}$.

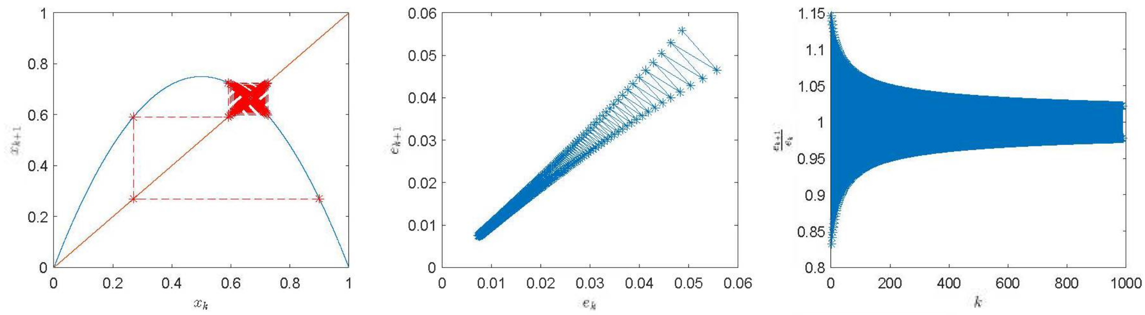


Fig. 4.9: Fixed point for logistic map for $a = 3$, terminating condition $|x_k - x^*| < 10^{-9}$.

4.5.1 Proof of Quadratic Convergence of Newton Method

Given a function $f(x)$ with a root x^*, Newton method is defined by:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

We want to show that the convergence rate of Newton method is quadratic, i.e.,

$$|x_{k+1} - x^*| \leq C|x_k - x^*|^2$$

for some constant C and for all k sufficiently large.

1. **Assumptions and Setup:**

   Assume that:

   - $f(x^*) = 0$ (i.e., x^* is a root of f).
   - f is twice continuously differentiable in a neighborhood around x^*.
   - $f'(x^*) \neq 0$.

2. **Taylor Series Expansion:**

   Consider the Taylor series expansion of f around x^*:

   $$f(x_k) = f(x^*) + f'(x^*)(x_k - x^*) + \frac{f''(\xi_k)}{2}(x_k - x^*)^2$$

   for some ξ_k between x_k and x^*.

   Since $f(x^*) = 0$, the expansion simplifies to:

   $$f(x_k) = f'(x^*)(x_k - x^*) + \frac{f''(\xi_k)}{2}(x_k - x^*)^2$$

3. **Newton Iteration:**

   Applying Newton method:

   $$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

   Substitute the Taylor expansion of $f(x_k)$:

   $$x_{k+1} = x_k - \frac{f'(x^*)(x_k - x^*) + \frac{f''(\xi_k)}{2}(x_k - x^*)^2}{f'(x_k)}$$

4. **Approximate $f'(x_k)$:**

   Since $f'(x)$ is continuous and x_k is close to x^*, we can approximate:

   $$f'(x_k) \approx f'(x^*)$$

   Therefore:

   $$x_{k+1} \approx x_k - \frac{f'(x^*)(x_k - x^*) + \frac{f''(\xi_k)}{2}(x_k - x^*)^2}{f'(x^*)}$$

5. **Simplify:**

   Simplifying the expression:

   $$x_{k+1} \approx x_k - (x_k - x^*) - \frac{\frac{f''(\xi_k)}{2}(x_k - x^*)^2}{f'(x^*)}$$

   $$x_{k+1} \approx x_k - x_k + x^* - \frac{f''(\xi_k)}{2f'(x^*)}(x_k - x^*)^2$$

   $$x_{k+1} \approx x^* - \frac{f''(\xi_k)}{2f'(x^*)}(x_k - x^*)^2$$

6. **Error Term:**

   Let $e_k = x_k - x^*$ be the error at the k-th iteration. Then:

   $$e_{k+1} = x_{k+1} - x^* \approx -\frac{f''(\xi_k)}{2f'(x^*)}e_k^2$$

   Hence:

   $$|e_{k+1}| \approx \left|\frac{f''(\xi_k)}{2f'(x^*)}\right| |e_k|^2 \quad \square$$

4.5.2 Basins of Attraction and criteria of convergence

Definition

A basin of attraction for a fixed point in an iterative method is the set of initial points that lead to convergence to that specific fixed point under the iteration. For Newton method, the basin of attraction of a root α of $\mathbf{F}(\mathbf{x})$ is the set:

$$B(\alpha) = \{\mathbf{x}_0 \in \mathbb{R}^n \mid \lim_{n\to\infty} \mathbf{x}_n = \alpha\}$$

where $\mathbf{x}_n$ is the sequence generated by Newton method starting from $\mathbf{x}_0$.

Example 10: Fractal Nature of Basins of Attraction

The basins of attraction for Newton method can exhibit fractal structures. This fractal nature arises from the complex dynamics of the iterative process, especially in cases where there are multiple roots. Small changes in the initial guess can lead to convergence to different roots, creating a highly intricate boundary between basins.

Basin of Attraction for $x^5 - 1$

Consider the polynomial $f(x) = x^5 - 1$. The roots of this polynomial are the complex fifth roots of unity, and they are given by:

$$\alpha_k = e^{2\pi ik/5} \quad \text{for} \quad k = 0, 1, 2, 3, 4$$

where i is the imaginary unit. Explicitly, these roots are:

$$\begin{aligned} \alpha_0 &= 1, \\ \alpha_1 &= e^{2\pi i/5}, \\ \alpha_2 &= e^{4\pi i/5}, \\ \alpha_3 &= e^{6\pi i/5}, \\ \alpha_4 &= e^{8\pi i/5}. \end{aligned}$$

Each of these roots has its own basin of attraction under Newton method. The basins of attraction for the polynomial $x^5 - 1$ are regions in the complex plane such that any initial guess within a given region will converge to a particular root. Let us visualize the basin of attraction of root $\alpha_0 = 1$, see Figure 4.10.

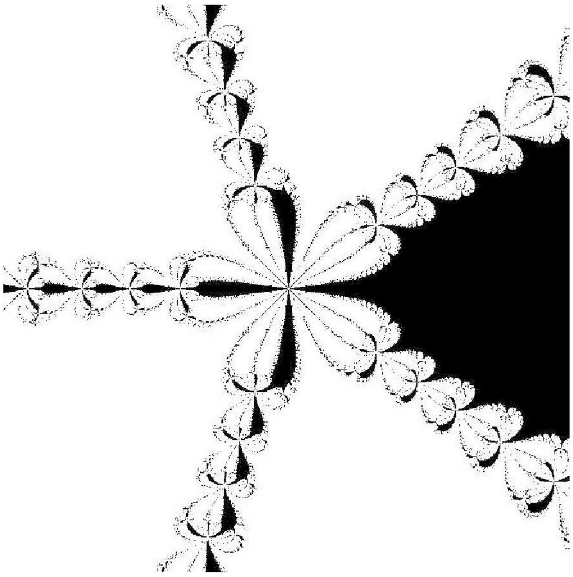


Fig. 4.10: Basin of attraction of $\alpha_0 = 1$ as a root of $f(x) = x^5 - 1$ using a sequence given by Newton method.

Fourier conditions

For function f of a single variable x we may examine convergence using the following conditions.

1. Let f be continuous up to the second derivative in $[a, b]$, $f(a) \cdot f(b) \leq 0$.
2. f' and f'' doesn't change its sign in $[a, b]$, and $\forall x \in [a, b] : f'(x) \neq 0$

Let's choose $x_0 \in \{a, b\}$ such that $f(x_0) \cdot f'' \geq 0$. Then the sequence generated by Newton method converges monotonously to $\hat{x}$. The convergence is quadratic.

It is important to note that these conditions are sufficient but not necessary. There can be cases where Newton method converges quadratically even if these conditions are not fully satisfied.

4.5.3 Big O and Little o Notation

1. Big O Notation (O): This notation is used to describe an upper bound on the growth rate of a function. For a function $f(x)$, we say $f(x) = O(g(x))$ as $x \to \infty$ if there exist constants C and M such that
$$|f(x)| \leq C|g(x)| \quad \text{for all } x > M.$$
2. Little o Notation (o): This notation is used to describe a function that grows much slower than another function. We say $f(x) = o(g(x))$ as $x \to \infty$ if for every constant $\epsilon > 0$, there exists a constant M such that
$$|f(x)| \leq \epsilon|g(x)| \quad \text{for all } x > M.$$

Connection to Convergence Rates

1. Linear Convergence: If an iterative method has a linear convergence rate, the error in the next iteration is proportional to the error in the current iteration. In Big O notation, we write:
$$e_{k+1} = O(e_k).$$
This indicates that the error decreases linearly with each iteration.
2. Superlinear Convergence: If an iterative method has a superlinear convergence rate, the error decreases faster than linear but not necessarily quadratically. Using Little o notation:
$$e_{k+1} = o(e_k).$$
This indicates that the ratio $\frac{e_{k+1}}{e_k} \to 0$ as $k \to \infty$, meaning the error reduction accelerates.
3. Quadratic Convergence: If an iterative method has a quadratic convergence rate, the error in the next iteration is proportional to the square of the current error. In Big O notation, we write:
$$e_{k+1} = O(e_k^2).$$
This indicates that the error decreases quadratically, resulting in a very rapid convergence.

4.5.4 Convergence Acceleration and Steffensen Method

Convergence acceleration techniques aim to enhance the rate of convergence of an iterative process. In numerical analysis, these techniques are vital for improving the efficiency of methods that exhibit slow convergence. The objective is to transform a sequence with linear or sub-linear convergence into one with faster convergence, ideally quadratic or higher-order convergence.

Steffensen method is an iterative technique used to find the roots of a function, providing an accelerated convergence. The method is based on Aitken Δ^2 process and is defined as follows:

Given an initial guess x_0, Steffensen iteration is:

$$\tilde{x}_{n+1} = x_n - \frac{(g(x_n) - x_n)^2}{g(g(x_n)) - 2g(x_n) + x_n}$$

This method approximates the root of $f(x) = 0$ by accelerating convergence of iteration process $x_{n+1} = g(x_n)$ and typically achieves quadratic convergence under suitable conditions.

Example 11: Multiple Roots and Linear Convergence of Newton Method

Consider the case where $f(x)$ has a multiple root. A multiple root occurs when $f(\alpha) = 0$ and $f'(\alpha) = 0$, but $f''(\alpha) \neq 0$. For example, the function $f(x) = (x - \alpha)^2$ has a double root at $x = \alpha$.

When applying Newton method to $f(x)$ with a multiple root, the iteration formula is:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

However, in the presence of multiple roots, Newton method converges only linearly. This is because the derivative $f'(x)$ is small near the root, slowing down the convergence.

To achieve quadratic convergence in such cases, we can employ one of the following strategies:

1. **Modification of Newton Method for Multiple Roots:**

For a function $f(x)$ with a root of multiplicity m, modify Newton method as:

$$x_{n+1} = x_n - m\frac{f(x_n)}{f'(x_n)}$$

This modification accounts for the multiplicity m and restores quadratic convergence.

2. **Using a New Function $u(x) = \frac{f(x)}{f'(x)}$:**

Define a new function $u(x)$:

$$u(x) = \frac{f(x)}{f'(x)}$$

Applying Newton method to $u(x)$ helps address the issue of multiple roots, as it transforms the original problem into one where the roots are simple, enhancing the convergence rate.

3. **Steffensen Method:**

Steffensen method can also be used to accelerate the convergence:

$$\tilde{x}_{n+1} = x_n - \frac{(g(x_n) - x_n)^2}{g(g(x_n)) - 2g(x_n) + x_n},$$

where

$$g(x) = x - \frac{f(x)}{f'(x)}.$$

By applying Steffensen iteration, we achieve quadratic convergence without requiring the explicit knowledge of the multiplicity of the root.

Example 12: Steffensen method

Let us consider the sequence generated from the iterative method applied to the function $g(x) = \cos(x)$. The goal is to find the root $\hat{x}$, which is approximately 0.739085.

The data from the iterative process is as follows:

IT	x_n	$x_n - \hat{x}$	error ratio
0	1	0.260915	NaN
1	0.540302	-0.198783	-0.761868
2	0.857553	0.118468	-0.595968
3	0.654290	-0.084795	-0.715653
4	0.793462	0.054377	-0.641429
5	0.701369	-0.037716	-0.693704
6	0.763959	0.024874	-0.659556
7	0.722103	-0.016982	-0.682986
8	0.750418	0.011333	-0.667516
9	0.731404	-0.007681	-0.678091
10	0.744237	0.005152	-0.671030
11	0.735604	-0.003481	-0.675431
12	0.741426	0.002341	-0.672350

Table 4.1: Iterative process for $g(x) = \cos(x)$

To apply the Steffenson process, we use the formula:

$$\tilde{x}_{n+1} = x_n - \frac{(g(x_n) - x_n)^2}{g(g(x_n)) - 2g(x_n) + x_n}$$

1. **Initial Iteration:**

$x_0 = 1$, $x_1 = 0.540302$, $x_2 = 0.857553$

2. **Compute Aitken Accelerated Value:**

$$\tilde{x}_0 = 1 - \frac{(0.540302 - 1)^2}{0.857553 - 2 \cdot 0.540302 + 1}$$
$$\tilde{x}_0 = 0.728010$$

3. Compute two new iterations using the fixed point method and use Aitken accelerated value again. Repeat the process until the ending criterion is fulfilled.

Intuition Behind Aitken Δ^2 Process

The original sequence converges linearly, meaning each term x_n is incrementally closer to the limit $\hat{x}$ by a factor. Aitken Δ^2 process leverages the differences between successive terms to predict a better estimate for the limit, effectively removing the linear error term. By using the differences, it provides an accelerated convergence towards the limit.

Aitken Δ^2 Process

Aitken Δ^2 process is a method for accelerating the convergence of a sequence. It is particularly useful when dealing with sequences that converge linearly, helping to achieve faster convergence by transforming the original sequence. This method is often applied in iterative numerical methods where convergence speed is crucial.

Given a sequence $\{x_n\}$, Aitken Δ^2 process produces a new sequence $\{\tilde{x}_n\}$ defined by:

$$\tilde{x}_n = x_n - \frac{(x_{n+1} - x_n)^2}{x_{n+2} - 2x_{n+1} + x_n}$$

- **Numerator** $(x_{n+1} - x_n)^2$**:** This term represents the squared difference between successive terms. By squaring the difference, we emphasize the extent of change, making it more significant in the adjustment process.
- **Denominator** $x_{n+2} - 2x_{n+1} + x_n$**:** This is the second forward difference, which captures the acceleration or deceleration in the sequence values. It helps in understanding the curvature or bending of the sequence.

Understanding the Components

- **Forward Differences:** The forward difference Δx_n measures the change between successive terms of the sequence:

  $$\Delta x_n = x_{n+1} - x_n$$

  The second forward difference $\Delta^2 x_n$ measures the change in these changes:

  $$\Delta^2 x_n = \Delta(\Delta x_n) = x_{n+2} - 2x_{n+1} + x_n$$

  These differences help to understand how the sequence is evolving and provide a way to predict its behavior.
- **Linear Convergence:** If a sequence $\{x_n\}$ converges linearly to L, then:

  $$x_n \approx L + C\rho^n$$

  where $0 < \rho < 1$ is the rate of convergence, and C is a constant. The idea is to eliminate the term involving ρ^n to get a better estimate of L.

The Goal

The aim of Aitken Δ^2 process is to estimate the limit of a sequence more accurately by using the information from the sequence itself. If $\{x_n\}$ is converging to a limit L, the idea is to create a new sequence $\{x_n\}$ that converges to L faster than the original sequence.

Derivation of the Aitken Δ^2 formula

For a linearly convergent sequence, we have approximately the following relation between three successive error terms.

$$\frac{e_{k+1}}{e_k} \approx \frac{e_k}{e_{k-1}}$$

$$\frac{x_{k+1}-\hat{x}}{x_k-\hat{x}} \approx \frac{x_k-\hat{x}}{x_{k-1}-\hat{x}}$$

.

If we solve $\frac{x_{k+1}-\hat{x}}{x_k-\hat{x}} \approx \frac{x_k-\hat{x}}{x_{k-1}-\hat{x}}$ for $\hat{x}$.

We will get the formula for Aitken δ^2-method

$$\hat{x} \approx x_k - \frac{(x_{k+1}-x_k)^2}{x_{k+2}-2x_{k+1}+x_k} = \tilde{x}_k.$$

Why It Works?

Aitken Δ^2 process works by effectively removing the linear term $C\rho^n$ from the sequence approximation, leaving a more accurate estimate of the limit L. Here's a step-by-step breakdown:

1. **Approximate x_{n+1} and x_{n+2}:** If the sequence $\{x_n\}$ converges linearly, we can write:

$$x_{n+1} \approx L + C\rho^{n+1}$$

$$x_{n+2} \approx L + C\rho^{n+2}$$

2. **Forward Differences:** Calculate the first and second forward differences:

$$\Delta x_n \approx C\rho^{n+1} - C\rho^n = C\rho^n(\rho-1)$$

$$\Delta^2 x_n \approx (L+C\rho^{n+2}) - 2(L+C\rho^{n+1}) + (L+C\rho^n) = C\rho^n(\rho^2-2\rho+1) = C\rho^n(\rho-1)^2$$

3. **Substitute into the Aitken Formula:**

$$x_n^* = x_n - \frac{(C\rho^n(\rho-1))^2}{C\rho^n(\rho-1)^2} = x_n - \frac{C^2\rho^{2n}(\rho-1)^2}{C\rho^n(\rho-1)^2} = x_n - C\rho^n$$

4. **Result:**

$$x_n^* \approx L + C\rho^n - C\rho^n = L$$

Hence, x_n^* is a better estimate of L and converges more rapidly than the original sequence.

Chapter 5

Iterative solving of systems of linear equations

5.1 Recommended literature

[7] Rainer Kress. *Numerical analysis*. Vol. 181. Springer Science & Business Media, 2012
[5] A. Hadjidimos. “Successive overrelaxation (SOR) and related methods”. In: *Journal of Computational and Applied Mathematics* 123.1 (2000). Numerical Analysis 2000. Vol. III: Linear Algebra, pp. 177–199. ISSN: 0377-0427. DOI: https://doi.org/10.1016/S0377-0427(00)00403-9
[14] Josef Stoer et al. *Introduction to numerical analysis*. Vol. 1993. Springer, 1980

5.2 Introduction

Iterative methods are widely used to solve systems of linear equations, particularly when the system is large and/or sparse. Unlike direct methods (such as Gaussian elimination), iterative methods start with an initial approximation to the solution and improve it step by step. The general form of a linear system is:

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

where $\mathbf{A}$ is an $n \times n$ matrix, $\mathbf{x}$ is an unknown vector, and $\mathbf{b}$ is a known vector. Iterative methods generate a sequence of approximations $\mathbf{x}^{(k)}$ that ideally converge to the true solution $\mathbf{x}$.

The general form of an iterative method can be written as:

$$\mathbf{x}^{(k+1)} = \mathbf{M}\mathbf{x}^{(k)} + \mathbf{c}$$

where $\mathbf{M}$ is the iteration matrix, $\mathbf{c}$ is a constant vector, and $\mathbf{x}^{(k)}$ is the k-th approximation to the solution.

Motivation to Study Large Sparse Matrice

In the realm of numerical analysis and scientific computing, large sparse matrices are a fundamental tool. These matrices, which contain a significant number of zero elements, arise naturally in various applications such as network theory, image processing, and partial differential equations. One particular area of interest is the study of tree diagonal matrices using the discrete version of the heat equation.

The heat equation is a partial differential equation (PDE) that describes how heat diffuses through a given region over time. When discretized, it provides a framework for understanding how heat or other diffusive processes propagate across a network or grid, which can be represented by a graph. In such cases, the graph's adjacency matrix is often large and sparse, with a tree structure representing hierarchical or recursive connections.

Studying the discrete heat equation on such matrices allows us to explore the dynamics of diffusion processes in complex networks, with applications ranging from thermodynamics to financial systems and even machine learning. Moreover, the specific structure of tree diagonal matrices offers computational advantages, as they often allow for more efficient algorithms for solving linear systems, eigenvalue problems, and matrix factorization.

As an example we consider the heat equation, which describes how heat diffuses through a medium:

$$\frac{\partial u(x,t)}{\partial t} = \alpha \frac{\partial^2 u(x,t)}{\partial x^2},$$

where $u(x,t)$ is the temperature at position x and time t, and α is the thermal diffusivity of the material.

To solve this equation numerically, we discretize the spatial domain into equally spaced points, with spacing h. Similarly, we discretize the time domain into equally spaced points, with spacing Δt. Let u_i^n denote the temperature at the i-th spatial point at time t_n. The second derivative with respect to x can be approximated using the finite difference method:

$$\frac{\partial^2 u(x,t)}{\partial x^2} \approx \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{h^2}.$$

Similarly, the time derivative can be approximated as:

$$\frac{\partial u(x,t)}{\partial t} \approx \frac{u_i^{n+1} - u_i^n}{\Delta t}.$$

Substituting these approximations into the heat equation gives:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \alpha \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{h^2}.$$

Rearranging, we obtain:

$$u_i^{n+1} = u_i^n + \frac{\alpha \Delta t}{h^2} \left(u_{i+1}^n - 2u_i^n + u_{i-1}^n \right).$$

This can be written in matrix form as:

$$\mathbf{u}^{n+1} = \mathbf{A}\mathbf{u}^n,$$

where $\mathbf{u}^n = [u_1^n, u_2^n, \ldots, u_{N-1}^n]^T$ is the vector of temperatures at time t_n, and $\mathbf{A}$ is the matrix given by:

$$\mathbf{A} = \mathbf{I} + \frac{\alpha \Delta t}{h^2}\mathbf{L},$$

where $\mathbf{I}$ is the identity matrix, and $\mathbf{L}$ is the Laplacian matrix associated with the discrete spatial grid. For a one-dimensional domain, $\mathbf{L}$ is a tridiagonal matrix:

$$\mathbf{L} = \begin{bmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \cdots & 0 \\ 0 & 1 & -2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & -2 \end{bmatrix}.$$

If you want to avoid the formulation using partial differential equations (PDEs), you can think of it in a more intuitive way by imagining a metal stick and observing how heat spreads through it. Instead of dealing with continuous variables, you select a finite number of points along the stick. At each of these points, the temperature at a given time t is influenced only by the temperatures at the neighboring points, but crucially, only from the previous time step $t - \Delta t$. Of course, the description of the boundary points is different; at these endpoints, the first and last equations may differ, as they need to account for the lack of neighboring points on one side, which could involve fixed temperatures, insulated ends, or other specific boundary conditions. This approach simplifies the problem by focusing on how the temperature at each discrete point evolves based on its immediate surroundings, providing a straightforward way to model heat diffusion without diving into the complexities of PDEs.

5.3 Jacobi and Gauss-Seidel Methods

5.3.1 Jacobi Method

The Jacobi method is one of the simplest iterative methods. It is defined by decomposing the matrix $\mathbf{A}$:

$$\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U}$$

where $\mathbf{L}$ is the strictly lower triangular part and $\mathbf{U}$ the strictly upper triangular part of $\mathbf{A}$.

The iterative scheme in matrix form is then:

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}\mathbf{b} - \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{x}^{(k)}$$

Here, the iteration matrix is $\mathbf{M} = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})$. Convergence of the Jacobi method depends on the spectral radius $\rho(\mathbf{M})$.

We can express the method equivalently in equation form. The i-th equation of the system is:

$$a_{i1}x_1 + \cdots + a_{ii}x_i + \cdots + a_{in}x_n = b_i$$

The component x_i is expressed as:

$$x_i = -\sum_{\substack{j=1 \\ j\neq i}}^{n} \frac{a_{ij}}{a_{ii}} x_j + \frac{b_i}{a_{ii}},$$

and it is used as the new $(k+1)$-th iteration:

$$x_i^{k+1} = -\sum_{\substack{j=1 \\ j\neq i}}^{n} \frac{a_{ij}}{a_{ii}} x_j^k + \frac{b_i}{a_{ii}}.$$

Simply speaking, we express the component x_i from i-th equation.

5.3.2 Gauss-Seidel Method

The Gauss-Seidel method improves upon the Jacobi method by using updated values as soon as they are available. The matrix $\mathbf{A}$ is decomposed as:

$$\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U}$$

where $\mathbf{L}$ is the strictly lower triangular part and $\mathbf{U}$ the strictly upper triangular part of $\mathbf{A}$.

The iterative scheme is:

$$\mathbf{x}^{(k+1)} = (\mathbf{D} + \mathbf{L})^{-1}\mathbf{b} - (\mathbf{D} + \mathbf{L})^{-1}\mathbf{U}\mathbf{x}^{(k)}$$

The iteration matrix for Gauss-Seidel is $\mathbf{M} = -(\mathbf{D} + \mathbf{L})^{-1}\mathbf{U}$. Convergence for Gauss-Seidel is typically faster than for Jacobi (but not necessarly).

We can express the method equivalently in equation form. The i-th equation of the system is:

$$a_{i1}x_1 + \cdots + a_{ii}x_i + \cdots + a_{in}x_n = b_i$$

The component of the new iteration is used in the following step:

$$\begin{aligned}
x_1^{k+1} &= \frac{1}{a_{11}}\left(b_1 - a_{12}x_2^k - a_{13}x_3^k - a_{14}x_4^k - \dots\right) \\
x_2^{k+1} &= \frac{1}{a_{22}}\left(b_2 - a_{21}x_1^{k+1} - a_{23}x_3^k - a_{24}x_4^k - \dots\right) \\
x_3^{k+1} &= \frac{1}{a_{33}}\left(b_3 - a_{31}x_1^{k+1} - a_{32}x_2^{k+1} - a_{34}x_4^k - \dots\right) \\
&\vdots \\
x_i^{k+1} &= \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij}x_j^k\right)
\end{aligned}$$

Simply speaking, we express the component x_i from i-th equation. Unlike the Jacobi method in the Gauss-Seidel method, we use x_i^{k+1} to express x_j^{k+1} for $i < j$, in other words we always use the newest possible interactions.

Computing Specified Inverse Matrices

In both Jacobi and Gauss-Seidel methods, the inverse matrices $\mathbf{D}^{-1}$ and $(\mathbf{D}+\mathbf{L})^{-1}$ play a crucial role. For diagonal matrices $\mathbf{D}$, the inverse is straightforward to compute:

$$\mathbf{D}^{-1} = \operatorname{diag}\left(\frac{1}{a_{11}}, \frac{1}{a_{22}}, \dots, \frac{1}{a_{nn}}\right)$$

For lower triangular matrices $(\mathbf{D}+\mathbf{L})$, the inverse can be computed efficiently through forward substitution.

5.4 Criteria of convergence

A contraction mapping in the context of iterative methods is a function that brings points closer together. Specifically, for an iterative method defined by:

$$\mathbf{x}^{(k+1)} = \mathbf{M}\mathbf{x}^{(k)} + \mathbf{c},$$

the method is a contraction mapping if there exists a constant $\beta < 1$ such that:

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}\| \leq \beta\|\mathbf{x}^{(k)} - \mathbf{x}\|,$$

where $\mathbf{x}$ is the true solution. The key idea is that the distance between successive approximations decreases geometrically, ensuring convergence.

The spectral radius $\rho(\mathbf{M})$ of the iteration matrix $\mathbf{M}$ is directly related to whether the iterative method is a contraction mapping. The spectral radius is defined as the largest absolute value of the eigenvalues of $\mathbf{M}$:

$$\rho(\mathbf{M}) = \max |\lambda_i|,$$

where λ_i are the eigenvalues of $\mathbf{M}$.

For the iterative method to be a contraction mapping (and hence converge), the spectral radius must satisfy:

$$\rho(\mathbf{M}) < 1.$$

This condition ensures that the influence of any errors or deviations from the true solution diminishes with each iteration. Therefore, the condition $\rho(\mathbf{M}) < 1$ guarantees that the iterative process will contract towards the true solution, leading to convergence. Notice that convergence is quaratied for any intial itration.

5.4.1 Diagonal Dominance Theorem

The diagonal dominance theorem provides a sufficient condition for the convergence of the Jacobi and Gauss-Seidel methods. It states:

Theorem (Diagonal Dominance): Let $\mathbf{A}$ be an $n \times n$ matrix. If $\mathbf{A}$ is strictly diagonally dominant, i.e., for every $i = 1, 2, \ldots, n$:

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}| \text{ (strictly row diagonally dominant),}$$

$$|a_{ii}| > \sum_{j \neq i} |a_{jj}| \text{ (strictly column diagonally dominant),}$$

then the iterative methods like Jacobi and Gauss-Seidel are guaranteed to converge.

Explanation:

- Diagonal dominance ensures that the diagonal elements a_{ii} are large enough relative to the off-diagonal elements. This structure means that each equation in the system is more strongly determined by its own unknown than by the unknowns from other equations, making the system stable and conducive to iterative methods.

- The strict diagonal dominance implies that the iteration matrices $\mathbf{D}^{-1}\mathbf{R}$ for Jacobi and $\mathbf{L}^{-1}\mathbf{U}$ for Gauss-Seidel have a spectral radius less than 1, which is why convergence is ensured.

5.4.2 Positive Definiteness Theorem

Positive definiteness is another criterion often used to guarantee convergence, especially in the context of the Gauss-Seidel method.

Theorem (Positive Definiteness): Let $\mathbf{A}$ be an $n \times n$ symmetric matrix. If $\mathbf{A}$ is positive definite, meaning for any non-zero vector $\mathbf{z}$:

$$\mathbf{z}^T\mathbf{A}\mathbf{z} > 0,$$

then the Gauss-Seidel method will converge.

Explanation:

- A matrix $\mathbf{A}$ is positive definite if all its eigenvalues are positive. This property ensures that the matrix $\mathbf{A}$ is well-behaved, meaning it has no negative or zero eigenvalues that could destabilize the iteration process.
- When $\mathbf{A}$ is symmetric and positive definite, the iteration matrix $\mathbf{M}$ used in the Gauss-Seidel method has a spectral radius less than 1, ensuring that the method will converge.

5.5 Relaxation Method

Relaxation method extend the Gauss-Seidel methods by introducing a relaxation parameter ω. The general form for these methods is:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \omega \left(\mathbf{x}_{\text{GS}}^{(k+1)} - \mathbf{x}^{(k)}\right)$$

where $\mathbf{x}_{\text{GS}}^{(k+1)}$ is the updated solution from the Gauss-Seidel iteration. The relaxation parameter ω can accelerate convergence if chosen correctly. The iteration matrix of the method can be expressed as: $\mathbf{M} = -(D{+}\omega L)^{-1}\left[(\omega-1)D + \omega U\right]$, and $\mathbf{c} = (D{+}\omega L)^{-1}(\omega\mathbf{b})$

5.5.1 Optimal Selection of Relaxation Parameter

For relaxation method, selecting the optimal ω is crucial. If ω is too large, the method may diverge. If ω is too small, convergence will be slow. The optimal ω depends on the spectral radius of the iteration matrix. ω is typically chosen within the range $0 < \omega < 2$. If $0 < \omega < 1$, then, the method gives a new estimate that lies somewhere between the old estimate and the Gauss-Seidel estimate; in this case, the algorithm is termed *"under-relaxation"*. If $1 < \omega < 2$, then the new estimate lies on the extension of the vector joining the old estimate and the Gauss-Seidel estimate, and hence the algorithm is termed *"over-relaxation"*.

In practice, ω is often found through experimentation or by using theoretical estimates based on the properties of the matrix $\mathbf{A}$. In some cases we may compute ω analytically. For example, for tridiagonal matrices with non-zero diagonal elements with $\rho(\mathbf{M}_\mathrm{J}) < 1$ we estimate ω in the form

$$\omega_{\mathrm{opt}} = \frac{2}{1 + \sqrt{1 - \rho(\mathbf{M}_\mathrm{J})^2}},$$

where $\rho(\mathbf{M}_\mathrm{J})$ is the spectral radius of the Jacobi iteration matrix.

Relaxation methods can significantly improve the efficiency of solving large systems, particularly when combined with preconditioning techniques that further reduce the spectral radius of the iteration matrix.

5.6 Two-Grid method

5.6.1 Defect correction principle

This approach is especially useful in iterative solvers and multigrid methods, where direct inversion of the matrix is impractical due to the size and complexity of the problem.

The defect correction principle is a technique used in numerical linear algebra to improve the accuracy of an approximate solution to a linear system of equations. The basic idea is to iteratively correct the error (or defect) in the current approximate solution, thereby refining it towards the exact solution.

Consider a linear system:

$$A\mathbf{x} = \mathbf{b}$$

where A is an $n \times n$ matrix, $\mathbf{x}$ is the unknown solution vector, and $\mathbf{b}$ is the given right-hand side vector.

If $\mathbf{x}_0$ is an initial approximation to the true solution $\mathbf{x}$, then the residual (or defect) $\mathbf{r}_0$ associated with this approximation is defined as:

$$\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$$

The residual $\mathbf{r}_0$ represents the error in the current approximation. The idea of defect correction is to solve a new system that aims to reduce this residual. Specifically, we seek a correction term $\mathbf{d}$ such that:

$$A\mathbf{d} = \mathbf{r}_0$$

We observe that the correction term $\mathbf{d}$ will, in general, be small compared to $\mathbf{x}_0$, and therefore it is unnecessary to solve the defect correction equation exactly. We may relly on using an approximation of A^{-1}. Once $\mathbf{d}$ is found, the improved approximation $\mathbf{x}_1$ is obtained by:

$$\mathbf{x}_1 = \mathbf{x}_0 + \mathbf{d}$$

This process can be iteratively repeated, with the residual and correction being updated at each step, leading to a sequence of increasingly accurate approximations:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{d}_k$$

where $\mathbf{d}_k$ is the solution to:

$$A\mathbf{d}_k = \mathbf{r}_k \quad \text{and} \quad \mathbf{r}_k = \mathbf{b} - A\mathbf{x}_k.$$

In general, the defect correction principle, combined with the approximation of the inverse matrix A, forms a powerful framework for iterative methods, allowing for efficient and systematic refinement of solutions in large-scale linear systems.

5.6.2 Two-Grid Method

In the context of two-grid method, the defect correction principle is effectively utilized by approximating the inverse of the matrix A on a so-called coarse grid. The coarse grid represents a simplified or downsampled version of the original problem, leading to a smaller matrix than A, which acts on the fine grid.

The key idea is to project the fine-grid problem onto the coarse grid, where the computation of an approximate inverse of the matrix is much more manageable due to the reduced size. This coarse-grid approximation serves as a cost-effective substitute for the exact inverse of A and is particularly useful for eliminating low-frequency errors, which are difficult to address on the fine grid alone.

Once the coarse-grid correction is computed, it is interpolated back to the fine grid to update the fine-grid solution. This process allows for efficient error reduction across different frequency components, leveraging the coarse-grid approximation to accelerate convergence and improve the accuracy of the overall solution.

More precisely the two-grid method to solves the linear system $A\mathbf{x} = \mathbf{b}$ involves:

1. **Pre-Smoothing**: Applying a few iterations of the Gauss-Seidel method on the fine grid.
2. **Residual Computation**: Calculating the residual on the fine grid.
3. **Restriction**: Averaging adjacent fine-grid residuals and fine-grid matrix A to create a smaller coarse-grid system.
4. **Coarse-Grid Solution**: Solving the simplified system on the coarse grid.
5. **Interpolation**: Transferring the coarse-grid correction back to the fine grid.
6. **Post-Smoothing**: Further refining the solution on the fine grid using Gauss-Seidel.

5.7 Examples

Example 1:

Solve the system using Jacobi and Gauss–Seidel iterative methods.

$$\begin{aligned} 3x + y + z &= 4 \\ -x - 4y - z &= 3 \\ x + 2y - 4z &= 4 \end{aligned}$$

Solution:

Jacobi Iterations

- For x:
$$x^{(n+1)} = \frac{4 - y^{(n)} - z^{(n)}}{3}$$
- For y:
$$y^{(n+1)} = \frac{3 + x^{(n)} + z^{(n)}}{-4}$$
- For z:
$$z^{(n+1)} = \frac{4 - x^{(n)} - 2y^{(n)}}{-4}$$

Given the initial guess:

$$\begin{aligned} x^{(0)} &= 0, \\ y^{(0)} &= 0, \\ z^{(0)} &= 0. \end{aligned}$$

Iteration 1

Using the initial guess values, compute the first iteration:

$$\begin{aligned}
x^{(1)} &= \frac{4-0-0}{3} = \frac{4}{3}, \\
y^{(1)} &= \frac{3+0+0}{-4} = -\frac{3}{4}, \\
z^{(1)} &= \frac{4-0-2\times 0}{-4} = -1.
\end{aligned}$$

Iteration 2

Based on the results from Iteration 1:

$$\begin{aligned}
x^{(2)} &= \frac{4+\frac{3}{4}+1}{3} = \frac{19}{12} + \frac{4}{12} = \frac{23}{12}, \\
y^{(2)} &= \frac{3+\frac{4}{3}-1}{-4} = \frac{\frac{10}{3}}{-4} = -\frac{10}{12} = -\frac{5}{6}, \\
z^{(2)} &= \frac{4-\frac{23}{12}+\frac{3}{2}}{-4} = \frac{\frac{25}{12}}{-4} = -\frac{25}{24}.
\end{aligned}$$

Gauss-Seidel Iterations

- For x:

$$x^{(n+1)} = \frac{4-y^{(n)}-z^{(n)}}{3}$$

- For y:

$$y^{(n+1)} = \frac{3+x^{(n+1)}+z^{(n)}}{-4}$$

- For z:

$$z^{(n+1)} = \frac{4-x^{(n+1)}-2y^{(n+1)}}{-4}$$

Given the initial guess:

$$\begin{aligned}
x^{(0)} &= 0, \\
y^{(0)} &= 0, \\
z^{(0)} &= 0.
\end{aligned}$$

Iteration 1

Using the initial guess values, compute the first iteration:

$$\begin{aligned}
x^{(1)} &= \frac{4-0-0}{3} = \frac{4}{3}, \\
y^{(1)} &= \frac{3+\frac{4}{3}+0}{-4} = -\frac{3+\frac{4}{3}}{4} = -\frac{13}{12}, \\
z^{(1)} &= \frac{4-\frac{4}{3}-2\left(-\frac{13}{12}\right)}{-4} = \frac{4-\frac{4}{3}+\frac{13}{6}}{-4} = -\frac{29}{24}.
\end{aligned}$$

Iteration 2

Using updated values from the previous iteration:

$$\begin{aligned}
x^{(2)} &= \frac{4+\frac{13}{12}+\frac{29}{24}}{3} = \frac{151}{72}, \\
y^{(2)} &= \frac{3+\frac{151}{72}-\frac{29}{24}}{-4} = -\frac{35}{36}, \\
z^{(2)} &= \frac{4-\frac{151}{72}-2\left(-\frac{35}{36}\right)}{-4} = -\frac{277}{288}.
\end{aligned}$$

Example 2:

Solve systems of linear equations and graphically demonstrate the solution. Try changing the order of equations.

1.

$$\begin{aligned}
4x - y &= 3 \\
x + 3y &= 4
\end{aligned}$$

2.

$$\begin{aligned}
x + y &= 3 \\
-x + y &= 4
\end{aligned}$$

3.

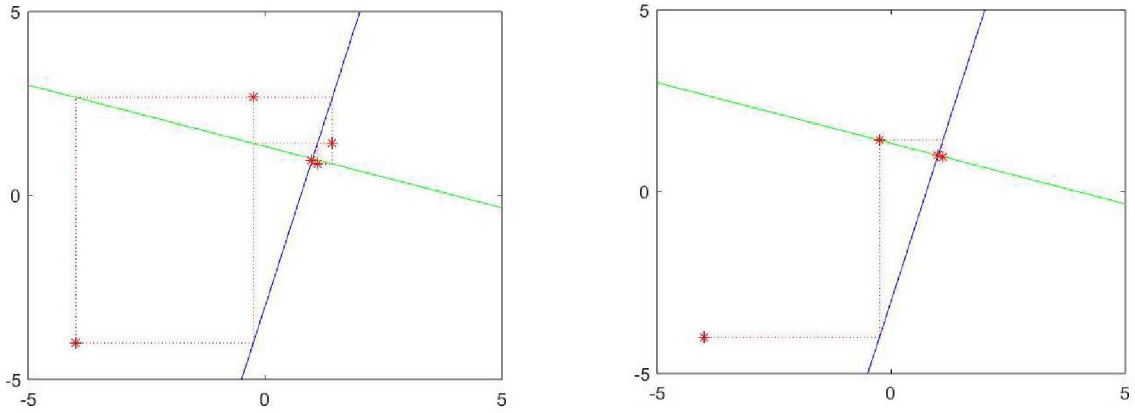
$$\begin{aligned}
1.1x + y &= 3 \\
-x + y &= 4
\end{aligned}$$

Solution

Part 1

System:

$$
\begin{aligned}
1.\quad & 4x - y = 3 \\
2.\quad & x + 3y = 4
\end{aligned}
$$



(a) Jacobi method

(b) Gauss-Seidel method

Fig. 5.1: Visualization of iterations for Part 1 starting with initial conditions $x_0 = -4$, $y_0 = -4$.

Observations:

- Convergence of the Jacobi method is slower than the Gauss-Seidel method.
- The Jacobi method can be graphically represented the following way: Create a line segment parallel to the x-axis starting at the initial iteration and ending at a point on the strait line representing the first equation. Create a line segment parallel to the y-axis starting at the initial iteration and ending at a point on the strait line representing the second equation. Create a rectangle that consists of the two lines described above and two additional parallel lines. The next iteration lies in the point that is opposite to the initial one.
- The Gauss-Seidel method can be graphically represented the following way: Create a line segment parallel to the x-axis starting at the initial iteration and ending at a new point on the strait line representing the first equation. Create a line segment parallel to the y-axis starting at the new point and ending at a point on the strait line representing the second equation.
- Gauss-Seidel method iterations lie on the straight line representing the second equation.
- Applying iterative methods such as Gauss-Seidel and Jacobi can exhibit a shift from convergent to divergent behaviors when the order of equations is altered.
- In the reverse order the divergence of the Gauss-Seidel method is faster than of the Jacobi method.

Matrix Formulation:

$$\mathbf{A} = \begin{bmatrix} 4 & -1 \\ 1 & 3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

Decompose A into D, L, and U:

$$\mathbf{D} = \begin{bmatrix} 4 & 0 \\ 0 & 3 \end{bmatrix}, \quad \mathbf{L} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} 0 & -1 \\ 0 & 0 \end{bmatrix}$$

Jacobi Iteration Matrix $\mathbf{M}_{\textbf{Jacobi}}$:

$$\mathbf{M}_{\text{Jacobi}} = \mathbf{D}^{-1}(\mathbf{L}+\mathbf{U}) = \begin{bmatrix} \frac{1}{4} & 0 \\ 0 & \frac{1}{3} \end{bmatrix} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -\frac{1}{4} \\ \frac{1}{3} & 0 \end{bmatrix}$$

Having eigenvalues $0.2887i, -0.2887i$. Therefore $\rho(\mathbf{M}_{\text{Jacobi}}) = 0.2887$.

Gauss-Seidel Iteration Matrix $\mathbf{M}_{\textbf{Gauss-Seidel}}$:

$$\mathbf{M}_{\text{Gauss-Seidel}} = (\mathbf{D}+\mathbf{L})^{-1}\mathbf{U} = \begin{bmatrix} 4 & 0 \\ 1 & 3 \end{bmatrix}^{-1} \begin{bmatrix} 0 & -1 \\ 0 & 0 \end{bmatrix}$$

First, compute $(\mathbf{D}+\mathbf{L})^{-1}$:

$$(\mathbf{D}+\mathbf{L})^{-1} = \begin{bmatrix} \frac{1}{4} & 0 \\ -\frac{1}{12} & \frac{1}{3} \end{bmatrix}$$

Thus, $\mathbf{M}_{\text{Gauss-Seidel}}$ is:

$$\mathbf{M}_{\text{Gauss-Seidel}} = \begin{bmatrix} \frac{1}{4} & 0 \\ -\frac{1}{12} & \frac{1}{3} \end{bmatrix} \begin{bmatrix} 0 & -1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -\frac{1}{4} \\ 0 & \frac{1}{12} \end{bmatrix}$$

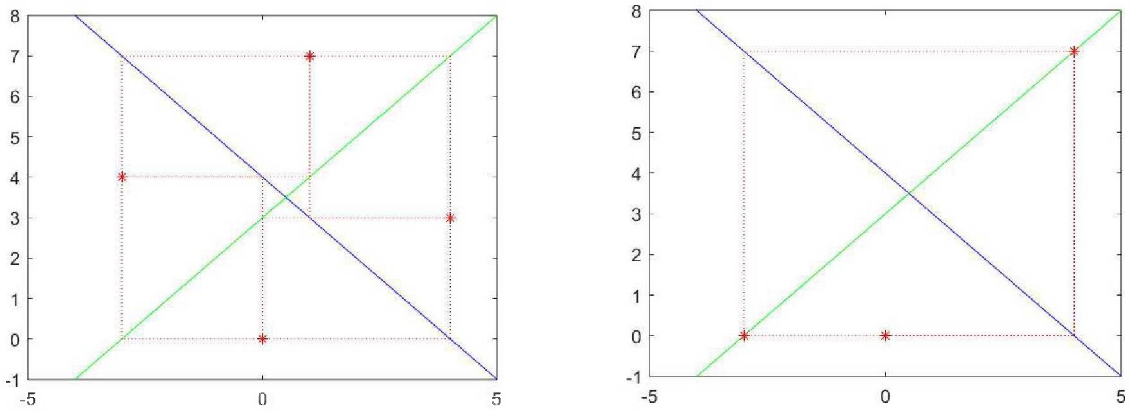
Having eigenvalues $0, 0.0833$. Therefore $\rho(\mathbf{M}_{\text{Gauss-Seidel}}) = 0.0833 < \mathbf{M}_{\text{Gauss-Seidel}}$.

Part 2

System:

$$\begin{aligned} &1. \quad x + y = 3 \\ &2. \quad -x + y = 4 \end{aligned}$$

Observations:



(a) Jacobi method

(b) Gauss-Seidel method

Fig. 5.2: Visualization of iterations for Part 2 starting with initial conditions $x_0 = 0$, $y_0 = 0$.

- Here, the equations describe perpendicular lines, leading to neither the Jacobi nor the Gauss-Seidel method converging, regardless of the equation order.
- The solutions tend to cycle, demonstrating oscillatory behavior without settling at a convergence point.

Matrix Formulation:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

Decompose A into D, L, and U:

$$\mathbf{D} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{L} = \begin{bmatrix} 0 & 0 \\ -1 & 0 \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

Jacobi Iteration Matrix $\mathbf{M}_{\textbf{Jacobi}}$:

$$\mathbf{M}_{\text{Jacobi}} = \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

Having eigenvalues $i, -i$. Therefore $\rho(\mathbf{M}_{\text{Jacobi}}) = 1$.

Gauss-Seidel Iteration Matrix $\mathbf{M}_{\textbf{Gauss-Seidel}}$:

$$\mathbf{M}_{\text{Gauss-Seidel}} = (\mathbf{D} + \mathbf{L})^{-1}\mathbf{U} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

First, compute $(\mathbf{D}+\mathbf{L})^{-1}$:

$$(\mathbf{D}+\mathbf{L})^{-1} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Thus, $\mathbf{M}_{\text{Gauss-Seidel}}$ is:

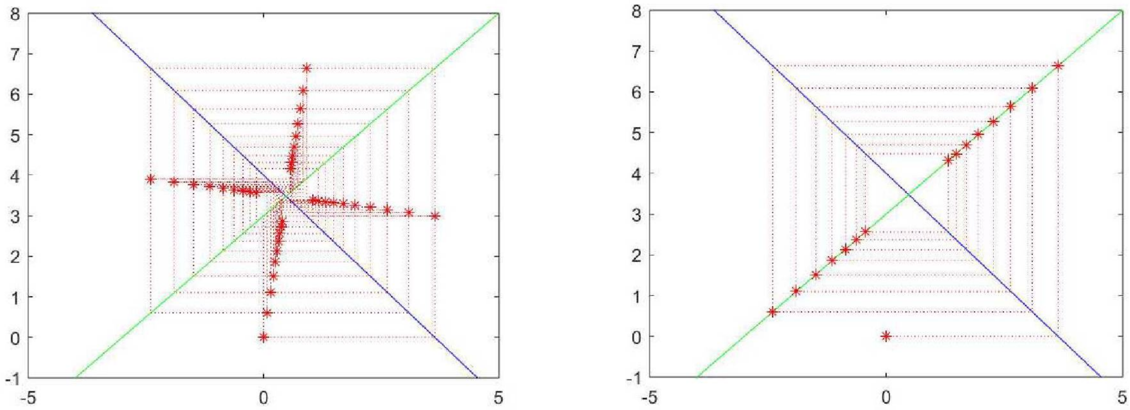
$$\mathbf{M}_{\text{Gauss-Seidel}} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$$

Having eigenvalues $0, 1$. Therefore $\rho(\mathbf{M}_{\text{Gauss-Seidel}}) = 1$.

Part 3

System:

$$\begin{aligned} &1. \quad 1.1x + y = 3 \\ &2. \quad -x + y = 4 \end{aligned}$$



(a) Jacobi method

(b) Gauss-Seidel method

Fig. 5.3: Visualization of iterations for Part 3 starting with initial conditions $x_0 = 0$, $y_0 = 0$.

Observations:

- The lines are nearly perpendicular, resulting in slow convergence or divergence in the iterative solutions.
- Such behaviors are sensitive to the initial guess and the numerical details of the applied method.

Matrix Formulation:

$$\mathbf{A} = \begin{bmatrix} 1.1 & 1 \\ -1 & 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

Decompose A into D, L, and U:

$$\mathbf{D} = \begin{bmatrix} 1.1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{L} = \begin{bmatrix} 0 & 0 \\ -1 & 0 \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

Jacobi Iteration Matrix $\mathbf{M_{Jacobi}}$:

$$\mathbf{M}_{\text{Jacobi}} = \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U}) = \begin{bmatrix} \frac{1}{1.1} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & \frac{1}{1.1} \\ -1 & 0 \end{bmatrix}$$

Having eigenvalues $0.9492i, -0.9492i$. Therefore $\rho(\mathbf{M}_{\text{Jacobi}}) = 0.9492$.

Gauss-Seidel Iteration Matrix $\mathbf{M_{Gauss\text{-}Seidel}}$:

$$\mathbf{M}_{\text{Gauss-Seidel}} = (\mathbf{D} + \mathbf{L})^{-1}\mathbf{U} = \begin{bmatrix} 1.1 & 0 \\ -1 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

First, compute $(\mathbf{D} + \mathbf{L})^{-1}$:

$$(\mathbf{D} + \mathbf{L})^{-1} = \begin{bmatrix} \frac{1}{1.1} & 0 \\ \frac{1}{1.1} & 1 \end{bmatrix}$$

Thus, $\mathbf{M}_{\text{Gauss-Seidel}}$ is:

$$\mathbf{M}_{\text{Gauss-Seidel}} = \begin{bmatrix} \frac{1}{1.1} & 0 \\ \frac{1}{1.1} & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & \frac{1}{1.1} \\ 0 & \frac{1}{1.1} \end{bmatrix}$$

Having eigenvalues $0, 0.9009$. Therefore $\rho(\mathbf{M}_{\text{Gauss-Seidel}}) = 0.9009$.

Example 3:

Use the relaxation parameter ω to speed up the Gauss–Seidel method for the system.

$$\begin{aligned} 1.1x + y &= 3 \\ -x + y &= 4 \end{aligned}$$

Solution

We can apply relaxation methods with various values $\omega \in (0, 2)$. We may for example try all values in $\{0.01, 0.02, \dots 1.99\}$ to solve the system using relaxation method with the given ω with limit of 500 iterations and tolerance $\varepsilon < 10^{-3}$. This leads to $\omega_{opt} = 0.75$. To find ω_{opt} is a one-dimensional optimization task on interval $(0, 2)$. We may more sophisticated optimization method to find a minimum, plausible methods will be discussed in the next chapter in section 6.2.

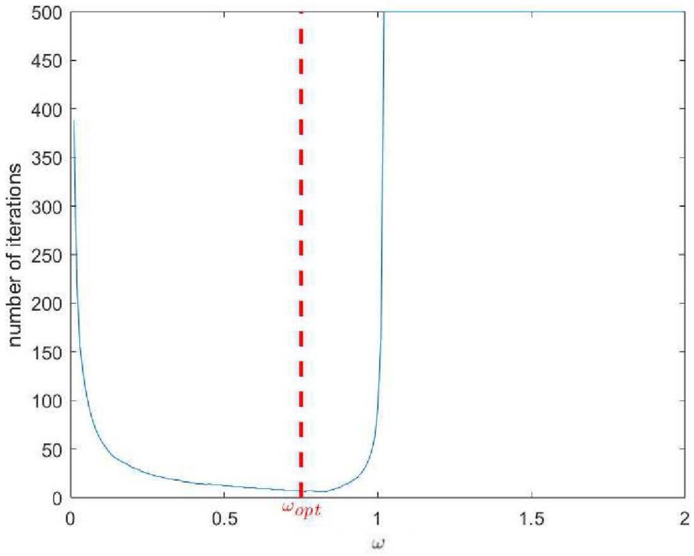


Fig. 5.4: Number of iteration used to find a solution of linear system using relaxation methods with different ω.

Example 4:

Solve the three-diagonal system of 10 equations with 10 unknowns using Jacobi and Gauss–Seidel iterative methods. Use also more sophisticated relaxation method and two-grid method.

$$\begin{pmatrix} 2 & -1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & \cdots & 0 & 0 & 0 \\ \vdots & & \ddots & \ddots & \ddots & & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & \dots & 0 & -1 & 2 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$$

Solution

The Jacobi method, known for its simplicity and parallelization capability, provided a solution vector $\mathbf{x}_n$ after 512 iterations. The spectral radius for this method was approximately 0.959, indicating a relatively slow convergence rate. The final solution vector was:

$$\mathbf{x}_n = \begin{bmatrix} 4.9999999972 \\ 8.9999999946 \\ 11.9999999925 \\ 13.999999991 \\ 14.9999999901 \\ 14.9999999901 \\ 13.999999991 \\ 11.9999999925 \\ 8.9999999946 \\ 4.9999999972 \end{bmatrix}$$

While the method converges, the high number of iterations required reflects the method's inefficiency for large or stiff systems.

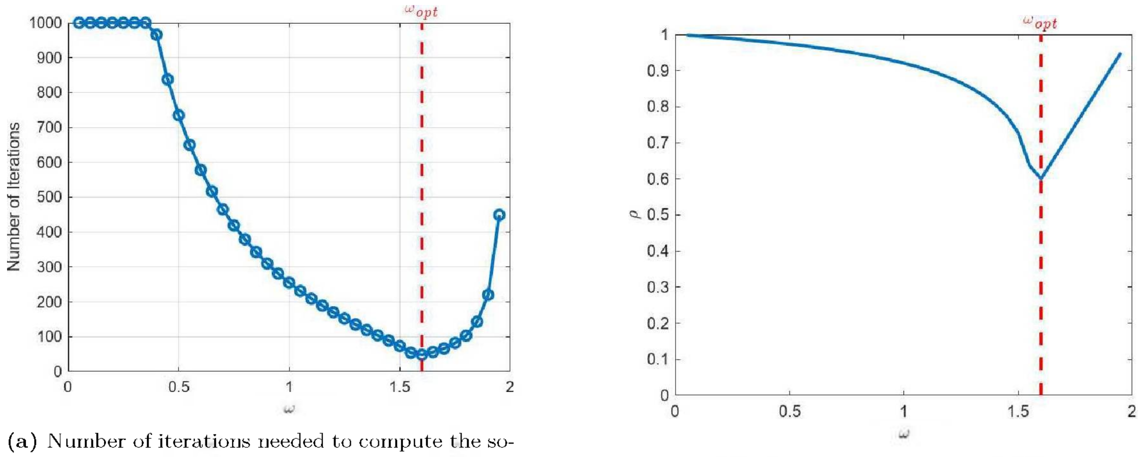
The Gauss-Seidel method improves upon the Jacobi method by using the updated values as soon as they are available, leading to faster convergence. In your results, this method required 265 iterations with a spectral radius of approximately 0.921. The solution vector was:

$$\mathbf{x}_n = \begin{bmatrix} 4.9999999984 \\ 8.9999999970 \\ 11.9999999959 \\ 13.9999999953 \\ 14.9999999951 \\ 14.9999999953 \\ 13.9999999958 \\ 11.9999999967 \\ 8.9999999977 \\ 4.9999999989 \end{bmatrix}$$

This method shows a better convergence rate compared to Jacobi, but still, the iteration count is significant, making it less ideal for very large systems.

The relaxation method, which is an extension of the Gauss-Seidel method, introduced a relaxation factor (ω) to accelerate convergence. We may estimate relaxation parameter ω through simulations resulting in ω of 1.60. With an optimal ω of 1.60, the method achieved convergence in just 48 iterations. The resulting solution vector was:

$$\mathbf{x} = \begin{bmatrix} 5.0000000004 \\ 9.0000000013 \\ 12.0000000013 \\ 14.0000000012 \\ 15.0000000010 \\ 15.0000000008 \\ 14.0000000005 \\ 12.0000000003 \\ 9.0000000001 \\ 4.9999999999 \end{bmatrix}$$



(a) Number of iterations needed to compute the solution

(b) Spectral radius of the SOR iteration matrix

Fig. 5.5: Estimation of optimal parameter ω_{opt} for SOR method.

This dramatic reduction in iteration count highlights the efficiency of the Successive Over-Relaxation (SOR) method, especially when the optimal relaxation parameter is used. The accuracy of the solution is also markedly high, with negligible deviations from the true values.

These results clearly demonstrate the trade-offs between simplicity and efficiency among the three methods. While the Jacobi method is the easiest to implement, it is also the slowest in terms of convergence. The Gauss-Seidel method offers a moderate improvement, reducing the number of iterations significantly. However, the SOR method stands out for its exceptional convergence speed, provided that the relaxation factor is well-chosen. Thus, for solving large linear systems where computational efficiency is crucial, the SOR method is typically the preferred choice. A plausible workaround for SOR method is (i) to solve the system with crude tolerance to estimate optimal ω, or (ii) to compute the spectral radius of the iteration matrix, and then in the second step use the optimal ω to improve the estimate of the solution. Another option is to study theoretical properties of matrix of the linear system. For tridiagonal matrix with spectral radius for Jacobi method smaller than 1 we may use the estimate

$$\omega_{\text{opt}} = \frac{2}{1 + \sqrt{1 - \rho(\mathbf{M}_{\text{J}})^2}} \approx 1.56.$$

Another possible extension of the Gauss-Seidel method is the two-grid method. In applying the two-grid method to solve the studied three-diagonal linear system, the method converged in 11 iterations For each iteration, 3 Gauss-Seidel iterations were performed during both the pre-smoothing and post-smoothing phases. The coarse-grid correction was calculated by solving the smaller 5 × 5 system exactly. The performance of the two-grid method in this setting was similar to that of the relaxation method for achieving the same final precision. When increasing the dimension of the original linear system from 10 × 10 to 20 × 20 while keeping the remaining setting the same, the two-grid method required 12 iterations, which was slightly better, but not dramatically, than the relaxation method, with the total iteration count reaching 89. In contrast, a simple Jacobi method would require almost 2000 iterations, Gauss-Seidel would need nearly 1000 iterations.

Chapter 6

Optimization

6.1 Recommended literature

[11] Jorge Nocedal and Stephen J Wright. *Numerical optimization.* Springer, 1999

6.2 Optimization in $\mathcal{R}$

Optimization is a fundamental concept in mathematics and applied sciences, dealing with the process of finding the best solution to a given problem. In the context of real-valued functions, optimization refers to finding the minimum or maximum value of a function within a certain domain.

Consider a continuous real function f defined on an interval $I = [a, b]$. The function f attains its minimum value on I at the point $\hat{x} \in I$, referred to as the *minimum point* of f. The numerical methods for finding the minimum point $\hat{x}$ can be broadly categorized into:

- Comparative methods (we need only values of function f to find the minimum)
- Gradient methods (we need also derivatives of function f to find the minimum)

A function f is called *unimodal* on I if it is decreasing on $[a, \hat{x}]$ and increasing on $[\hat{x}, b]$. We will focus on searching minimum of unimodal functions. Firstly we recall two simple methods.

6.2.1 Analytic Method

The analytic approach to finding the minimum involves the following steps:

- Identify the stationary point x^*, where $\frac{\partial f(x)}{\partial x}(x^*) = 0$.
- If $\frac{\partial^2 f(x)}{\partial x^2}(x^*) > 0$, then x^* is a minimum.
- For a unimodal function f, the point x^* is always a minimum.

This method requires the differentiation of the function f and solving system of nonlinear equations to identify the stationary point. We may apply a Newton to identify the stationary point. We solve system

$$\frac{\partial f(x)}{\partial x}(x^*) = f'(x) = 0,$$

assuming $f''(x)$ is known.

Algorithm

- Start with an initial condition x_0.
- Update the point using the formula:

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

6.2.2 Simple Division Method

The simple division method follows these steps:

- Define equally spaced points $a = x_0, x_1, \ldots, x_{n-1}, x_n = b$ on I.
- Evaluate the function at each point and find the minimum value among $f(x_0), \ldots, f(x_n)$.
- Approximate $\hat{x} \approx x_k$ with an error of $h = \frac{b-a}{n}$ for a unimodal function f.

This is an easiest comparative method. However, requires significant computational effort. In the following sections we will focus on decreasing number of evaluations of function f needed to obtain the solution.

6.2.3 Bisection Method

The bisection method is a more efficient alternative and follows this algorithm:

- Define equally spaced points $a = x_0, x_1, x_2, x_3, x_4 = b$ on I with step size $h = \frac{b-a}{4}$.
- Identify the minimum value from $f(x_1), f(x_2), f(x_3)$.
- Narrow down the interval to $[x_{k-1}, x_{k+1}]$, which is half the length of the previous interval.
- Repeat the process with two new points in every step until the interval is sufficiently small.

The computational complexity of the method involves calculating two new functional values at each step while halving the interval length.

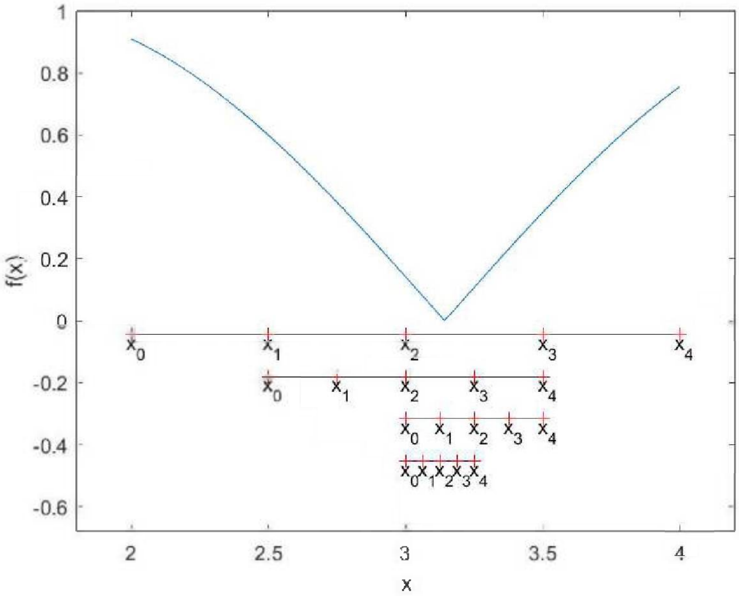


Fig. 6.1: Bisection optimization method for $f(x) = |\sin(x)|$ on interval $[2, 4]$.

6.2.4 Golden Ratio Method

The golden ratio method involves the following steps:

- Define points $a = x_1, x_2, x_3, x_4 = b$ in I, dividing the points according to the golden ratio $g = \frac{1+\sqrt{5}}{2}$.
- Find the minimal value from $f(x_2)$ and $f(x_3)$ at x_k.
- Narrow the interval to $[x_{k-1}, x_{k+1}]$, which is $1/g$ the length of the previous interval. Three of the original four points remain the same.
- Calculate the missing point and its functional value, then repeat the process until the interval is sufficiently small.

The computational complexity involves calculating one new functional value at each step while reducing the interval length by approximately 0.618 of the previous length.

6.2.5 Computational complexity of comparative methods

When analyzing the number of function evaluations required by the simple division method, the bisection method, and the golden ratio method, given an error tolerance of $\varepsilon = \frac{b-a}{n}$, where n is a natural number, we observe specific patterns in the efficiency of each method. Given that $b - a = 1$, the precision p is $\frac{1}{n}$. The analysis is performed using $n \in \{2, 3, 4, \ldots, 5000\}$, which provides the following details:

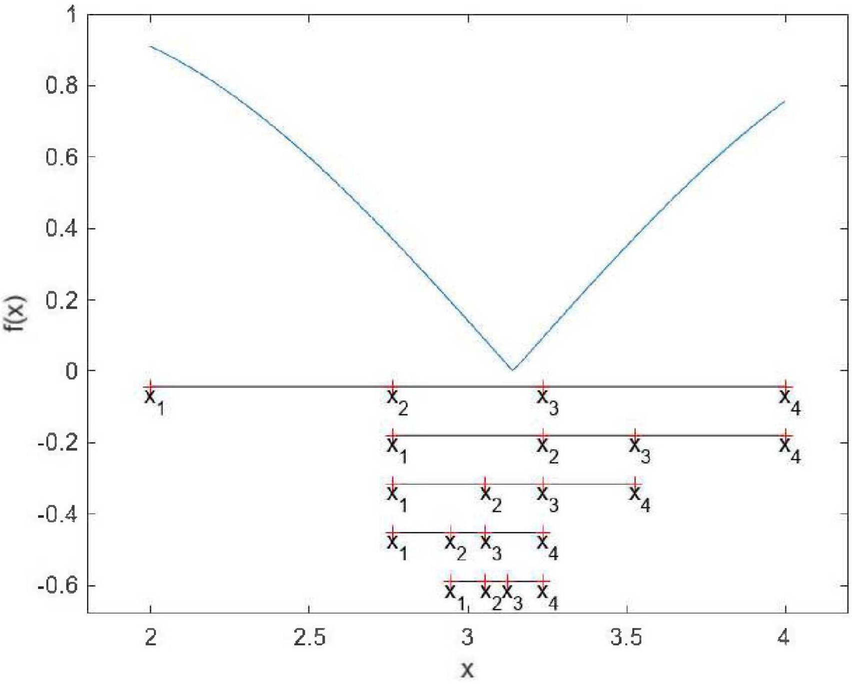


Fig. 6.2: Golden Ratio method for $f(x) = |\sin(x)|$ on interval $[2, 4]$.

1. **Simple Division Method**:
   - The number of function evaluations required is $n + 1$.
   - This method exhibits a linear relationship with n, meaning as n increases, the number of evaluations increases directly. For example, when $n = 1000$, the number of evaluations is 1001.
2. **Bisection Method**:
   - The number of function evaluations required is $2 \cdot (\lceil \log_2(n) \rceil - 1) + 3$.
   - This results in a logarithmic relationship with n. For instance, when $n = 1000$, the number of evaluations is 21, and for $n = 5000$, it increases to 25.
3. **Golden Ratio Method**:
   - The number of function evaluations required is $\lceil \log_\phi(n) \rceil + 1$, where ϕ is the golden ratio, approximately 1.618.
   - This also results in a logarithmic relationship, but with a different base compared to the bisection method. For example, when $n = 1000$, the number of evaluations is 19, and for $n = 5000$, it is 22.

Plot 6.3 demonstrates that for smaller values of n, the simple division method requires far more evaluations compared to the bisection and golden ratio methods. As n increases, representing higher precision, the golden ratio method consistently requires the fewest evaluations, followed by the bisection method. The simple division method, however, continues to scale linearly, resulting in the highest number of evaluations among the three methods for any given precision.

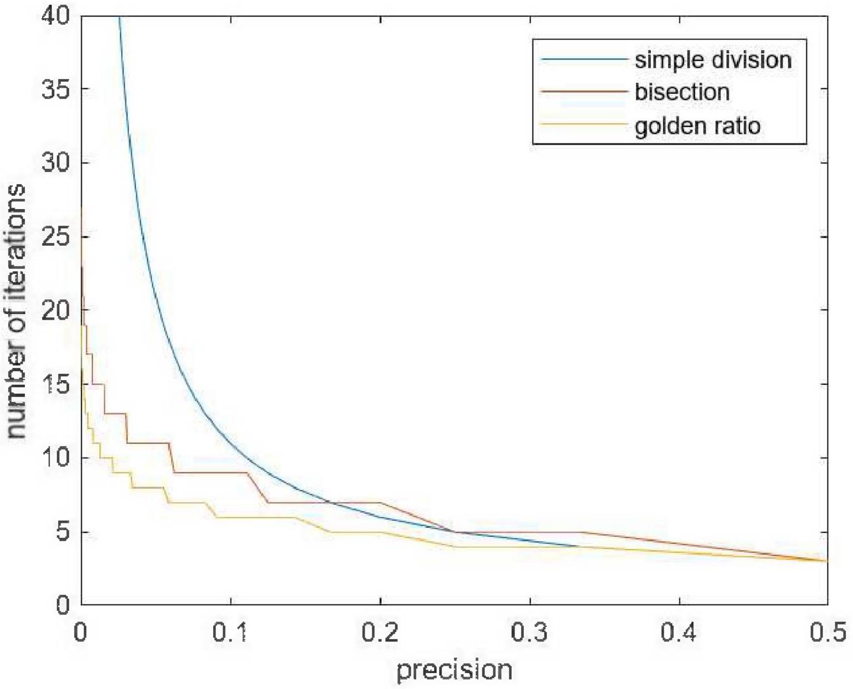


Fig. 6.3: Computational complexity of comparative methods

This comparison highlights the superior efficiency of the golden ratio method, especially as the required precision increases, making it the most suitable choice for applications demanding high accuracy with minimal function evaluations.

6.2.6 Fibonacci Method

The Fibonacci method is asymptotically equivalent to the golden ratio method and is particularly useful when the number of steps $N > 2$ is predetermined.

The Fibonacci sequence is defined as $F_0 = F_1 = 1$, $F_{k+1} = F_k + F_{k-1}$ for $k = 1, 2, \dots,$ and it asymptotically approaches the golden ratio. An alternative definition is $F_k = \frac{g^{k+1}-(1-g)^{k+1}}{\sqrt{5}}$.

The Fibonacci method involves the following steps:

- Divide the interval $[a, b]$ using the Fibonacci sequence ratio:

$$d_0 = b - a, \quad d_1 = d_0 \frac{F_N}{F_{N+1}}$$

- For each step, update the distance $d_k = d_{k-1} \frac{F_{N+1-k}}{F_{N+2-k}}$.
- The points are chosen similarly to the golden ratio method.

6.2.7 Golden Ratio vs. Fibonacci

To derive an alternative definition, we solve the Fibonacci sequence's difference equation:

$$F_n = F_{n-1} + F_{n-2},$$

with the initial conditions $F_0 = 0$ and $F_1 = 1$. The standard method for solving such a second-order linear recurrence relation involves assuming a solution of the form $F_n = r^n$, where r is a constant. Substituting this into the difference equation yields the characteristic equation:

$$r^2 - r - 1 = 0.$$

The solutions to this quadratic equation are:

$$r = \frac{1 \pm \sqrt{5}}{2},$$

which correspond to the golden ratio $g = \frac{1+\sqrt{5}}{2}$ and its conjugate $1 - g = \frac{1-\sqrt{5}}{2}$.

Thus, the general solution to the difference equation is:

$$F_n = A \cdot g^n + B \cdot (1-g)^n,$$

where A and B are constants determined by the initial conditions. Applying the initial conditions, we obtain Binet's formula:

$$F_n = \frac{g^n - (1-g)^n}{\sqrt{5}},$$

where $g = \frac{1+\sqrt{5}}{2}$ is the golden ratio. The term $(1-g)^n$ is negligible for large n, leading to the approximation:

$$F_n \approx \frac{g^n}{\sqrt{5}}.$$

When comparing the Fibonacci method and the Golden ratio method in the context of function optimization, we analyze the error reduction after N steps.

For the Golden ratio method, the error after N steps is given by:

$$d_{G,N} = \frac{d_0}{g^N},$$

where d_0 is the initial interval size, and g is the golden ratio.

For the Fibonacci method, the error after N steps is given by:

$$d_{F,N} = \frac{d_0}{F_{N+1}},$$

where F_{N+1} is the $(N+1)$-th Fibonacci number.

We know that:

$$F_N = \frac{g^{N+1} - (1-g)^{N+1}}{\sqrt{5}} \approx \frac{g^{N+1}}{\sqrt{5}},$$

because $(1-g)^{N+1}$ becomes negligible for large N. Substituting this approximation into the expression for $d_{F,N}$, we get:

$$d_{F,N} \approx \frac{d_0 \cdot \sqrt{5}}{g^{N+1}}.$$

Now, we can compare the two methods by evaluating the ratio of the errors:

$$\frac{d_{G,N}}{d_{F,N}} \approx \frac{\frac{d_0}{g^N}}{\frac{d_0 \cdot \sqrt{5}}{g^{N+1}}} = \frac{g^{N+1}}{g^N \cdot \sqrt{5}} = \frac{g}{\sqrt{5}}.$$

Given that $g = \frac{1+\sqrt{5}}{2} \approx 1.618$, we have:

$$\frac{d_{G,N}}{d_{F,N}} \approx \frac{1.618}{\sqrt{5}} \approx 1.17.$$

This result indicates that the Fibonacci method reduces the interval size about 1.17 times faster than the Golden ratio method after the same number of steps.

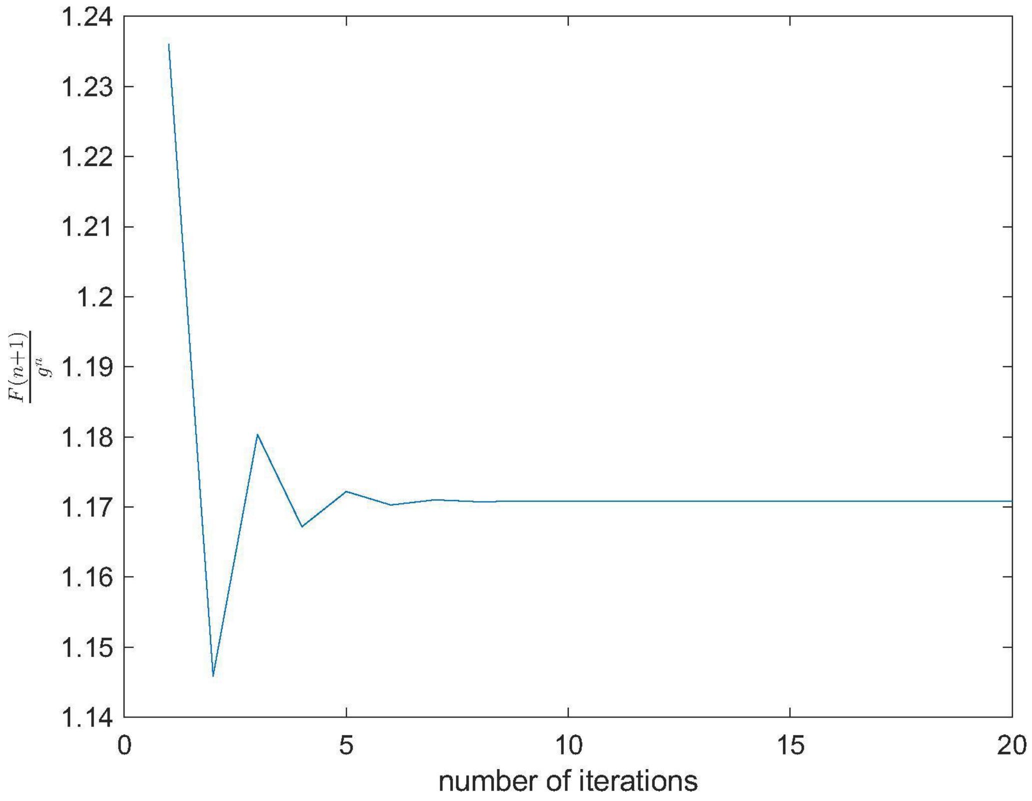
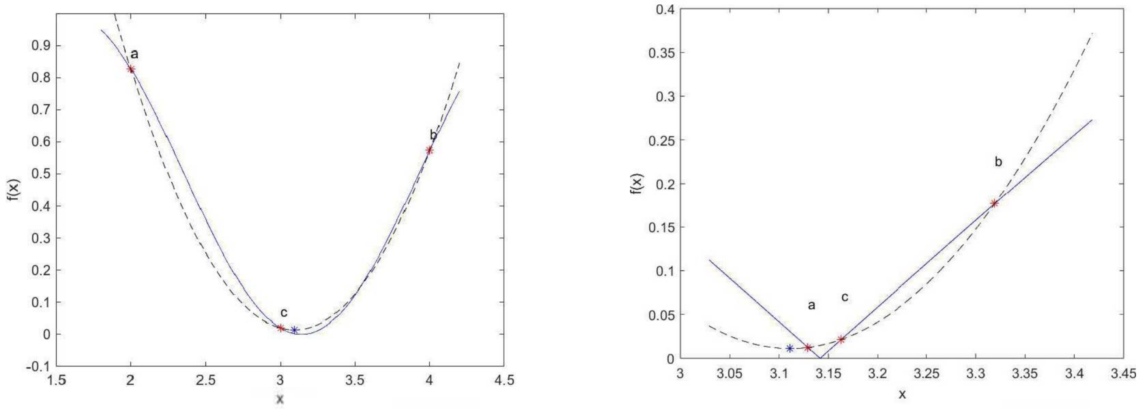


Fig. 6.4: Comparison of the shirking ratio between Fibonacci method and the golden ratio method for different number of iterations.

6.3 Quadratic Interpolation Method

Quadratic interpolation is a method used to approximate a function by a quadratic polynomial (a second-degree polynomial). This method is particularly useful in numerical optimization and root-finding problems. The idea behind quadratic interpolation is to



(a) Quadratic interpolation method $f(x) = \sin^2(x)$ on interval $[2, 4]$.

(b) Quadratic interpolation method $f(x) = |\sin(x)|$ on interval $[2, 4]$ after several iterations.

Fig. 6.5: Quadratic interpolation method: estimation of minimum (blue star), given point a, b, c (red stars), original function (blue line), polynomial approximation (

approximate the function with a parabola, which is then used to estimate the location of the function's extremum (minimum or maximum) or root.

Given three points (x_1, y_1), (x_2, y_2), and (x_3, y_3) on the graph of a function $f(x)$, quadratic interpolation fits a parabola $P(x) = ax^2 + bx + c$ through these points, see figure 6.5a. In practice $P(x)$ is an interpolation polynomial. We may proceed using following algorithm:

- Let $c \in [a, b]$, where c is typically the midpoint of the interval.
- Construct an interpolation polynomial (a parabola) through points a, b, and c.
- Find the minimum at point d – the zero point of the derivative:

$$d = \frac{1}{2}\left(a + b - \frac{f[a, b]}{f[a, b, c]}\right)$$

- Repeat the construction for points a, d, c, or c, d, b, depending on the subinterval containing d.
- If $c = d$, choose a different c. Plausible add/subtract a small number to/from c.

The convergence of the quadratic interpolation method, especially when applied to optimization problems, depends on several factors:

- **Proximity to the True Extremum:** The success and convergence of quadratic interpolation depend significantly on how close the initial interpolation points are to the true extremum or root of the function. If the points are well-chosen and lie near the desired extremum, the method is more likely to converge quickly.
- **Spacing of Points:** If the points are too far apart or unevenly spaced, the quadratic approximation may poorly represent the underlying function, leading to slow or non-convergence. Ideally, the points should be chosen such that they are neither too close

(which can cause numerical instability) nor too far apart (which can lead to poor approximation).

- **Continuity and Differentiability:** Quadratic interpolation requires the function to be smooth, at least continuous and differentiable up to the second derivative. Functions with discontinuities or sharp changes in slope may lead to inaccurate interpolation results and impede convergence.
- **Local Curvature:** The method assumes that the function behaves quadratically near the extremum or root. If the function is highly non-quadratic or has a flat region near the extremum, the quadratic model might not provide a good approximation, resulting in slow convergence or divergence.
- **Second Derivative (Curvature):** If the second derivative is very small near the extremum, indicating a flat region, the quadratic approximation might poorly estimate the extremum, delaying convergence.
- **Sensitivity to Round-Off Errors:** The method can be sensitive to numerical round-off errors, especially if the points are not chosen carefully. Ill-conditioned problems where the interpolated points are very close together can lead to large errors in the coefficients of the quadratic polynomial, thereby affecting convergence.

To illustrate potential issues with the convergence of the method, refer to Figure 6.5.

6.3.1 Examples

Example 1:

Start with Example 3 from the previous chapter 5. Use the relaxation parameter ω to speed up the Gauss–Seidel method for the system.

$$\begin{aligned} 1.1x + y &= 3 \\ -x + y &= 4 \end{aligned}$$

Find ω using one-dimensional optimization on interval $[0, 2]$.

Solution:

First, we define a function f, which has its minimum at ω_{opt}. This function employs relaxation methods, where ω is the input parameter and the number of iterations required to approximate the solution serves as the output. Optimization is conducted over the interval $[0.5, 1]$, as simulations in the previous chapter indicate that values outside this range result in excessively high iteration counts and potential inaccuracies in computations. The system is solved using the relaxation method with a limit of 500 iterations and a tolerance of $\varepsilon < 10^{-3}$. This process determines $\omega_{\text{opt}} = 0.8021$, achieving convergence after 6 iterations. The difference from previous estimates arises because the same number of iterations is observed for multiple values of ω. The function f is not unimodal, introducing challenges in the estimation process.

For the second approach, we redefine f using relaxation methods with ω as the input parameter, but instead of calculating the number of iterations, we determine the spectral radius of the matrix $M = (D - \omega L)^{-1} [(1-\omega)D + \omega U]$ and use this as the output value. Optimization is performed over the interval $[0, 2]$. This approach yields $\omega_{\text{opt}} = 0.8394$, with the solution reached after 7 iterations.

Example 2:

Start with Example 12 from the chapter 1. The following table contains measurements of the height of students from a local primary, secondary, and high school, ranging in age from 5 to 18 years. Assume that the height increases linearly for girls between the ages of 5 and some unknown age w. For girls older than w, the height remains constant. Use the one-dimensional optimization and the least-squares method to estimate the exact dependence for the given data. What height does your model predict for:

(a) A 12-year-old girl?

(b) A 5-year-old girl?

Age (years)	15.5	17.8	16.5	7.8	5.5	9.0	6.2	12.0	10.5	8.3	14.0	13.5	18.0
Height (cm)	170	168	167	134	114	139	121	154	144	130	160	159	169

Table 6.1: Height and Age Data of Girls (sample of data), full data are stored in file `heigth_of_girls.txt`

6.3.2 Solution:

Again, we begin by extracting the relevant information from the table, namely the age X and height Y of the girls. We then define f using a piecewise linear model where the height increases linearly until the age of w and remains constant thereafter. Parameter w is an input to the function f. The output should measure how good the given fit is, and for this purpose, we use the sum of residuals squared (SRS).

To construct the model, we define two functions:

$$f(x) = (x < w) \cdot (x - w)$$

representing the linear increase for ages below w, and

$$g(x) = 1$$

representing the constant height for ages above w.

Using these functions, we form a design matrix A and apply the least squares method to find the coefficients c that best fit the data. The resulting height function is:

$$h(x) = c_1 \cdot f(x) + c_2 \cdot g(x)$$

The sum of residuals squared (SRS) is calculated as the output of the function f, providing a measure of how well the model fits the data:

$$\text{SRS} = \sum_{i=1}^{n} (Y_i - h(X_i))^2$$

where Y_i are the observed heights and $h(X_i)$ are the predicted heights from our model. A smaller SRS value indicates a better fit.

The optimal value of w is $w_{\text{opt}} = 12.2169$. For $w = w_{\text{opt}}$, the model predicts the following heights:

- **For a 12-year-old girl:** The height is approximately **164.3877 cm**.
- **For a 5-year-old girl:** The height is approximately **108.4261 cm**.

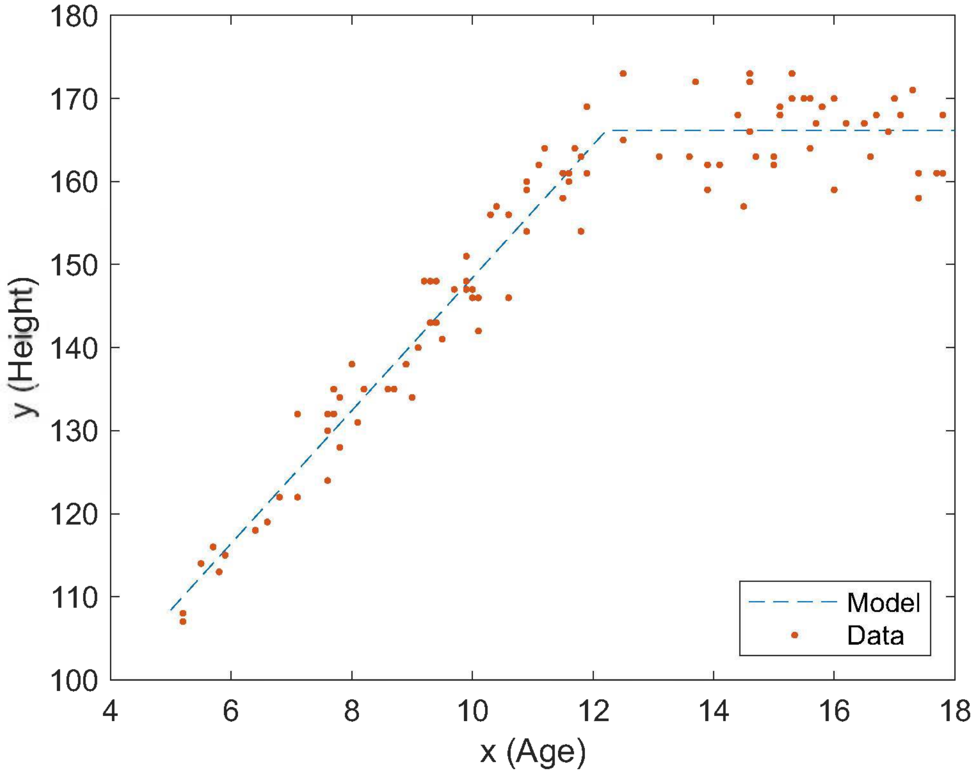


Fig. 6.6: Model of Height vs Age of Girls using one-dimensional optimization and least-square method.

This analysis demonstrates the use of change point regression, a numerical technique designed to model data that exhibits a distinct shift in behavior at a specific point, known as the change point. In this example, the change point w represents the age at which the rate of height increase transitions from a linear growth to a plateau. By incorporating this change point into the model, we can more accurately capture the underlying dynamics of the data. Change point regression is particularly valuable in statistical analysis when the relationship between variables is not uniform across the entire range of data. It allows for the detection and modeling of structural breaks, leading to improved model accuracy and deeper insights into the data's behavior.

6.4 Optimization in $\mathcal{R}^n$

This section focuses on optimization techniques in $\mathbb{R}^n$, the space of n-dimensional real vectors, which is a common setting for many practical optimization problems. The core objective in optimization is to minimize (or maximize) a function $f : \mathbb{R}^n \to \mathbb{R}$. Formally, given a function $f(\mathbf{x})$, where $\mathbf{x} \in \mathbb{R}^n$, we seek a point $\mathbf{x}^* \in \mathbb{R}^n$ such that:

$$f(\mathbf{x}^*) \leq f(\mathbf{x}) \quad \text{for all } \mathbf{x} \in \mathbb{R}^n.$$

There are various methods to tackle this problem, especially when $f(\mathbf{x})$ is nonlinear. Similarly to optimization in $\mathcal{R}$, these methods can broadly be classified into gradient (derivative-based) and comparative (derivative-free) methods. In this chapter, we will explore several widely-used optimization techniques in $\mathbb{R}^n$, including the Nelder-Mead method, gradient descent method, Newton's method and its variants (Quasi-Newton methods), and the Conjugate Gradient method.

For visualization of the algorithms described in the text see `https://www.benfrederickson.com/numerical-optimization/`.

6.5 Nelder-Mead Method

The Nelder-Mead method, also known as the downhill simplex method, is a widely-used derivative-free optimization technique. It is particularly useful when the gradient of the function is difficult or impossible to compute. The method operates by constructing a simplex, which is a geometric figure with $n + 1$ vertices in $\mathbb{R}^n$. For instance, in $\mathbb{R}^2$, a simplex is a triangle, and in $\mathbb{R}^3$, it is a tetrahedron.

6.5.1 Algorithm

1. **Initialization**: Start with $n + 1$ initial points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{n+1}$, which form the vertices of the simplex.
2. **Order**: Evaluate the function $f(\mathbf{x})$ at each vertex and order the vertices such that $f(\mathbf{x}_1) \leq f(\mathbf{x}_2) \leq \cdots \leq f(\mathbf{x}_{n+1})$.

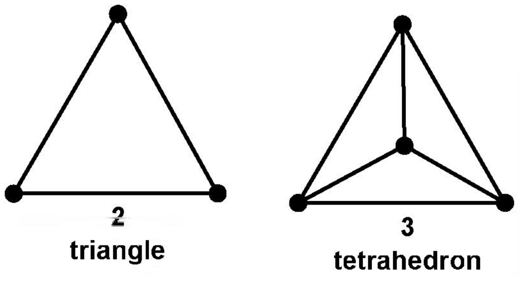


Fig. 6.7: Simplexes in $\mathcal{R}^2$ and $\mathcal{R}^3$.

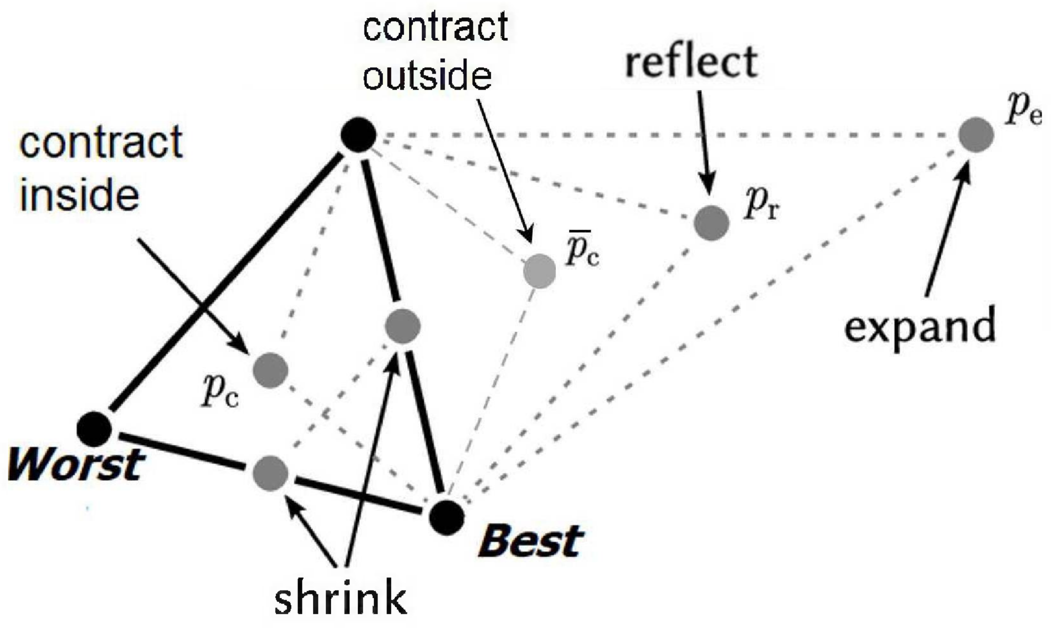


Fig. 6.8: Visualization of Nelder-Mead Method in $\mathcal{R}^2$.

3. **Reflection**: Compute the centroid $\mathbf{x}_c$ of the first n vertices (excluding the worst vertex $\mathbf{x}_{n+1}$) and reflect the worst point $\mathbf{x}_{n+1}$ through the centroid:

$$\mathbf{p}_r = \mathbf{x}_c + \alpha(\mathbf{x}_c - \mathbf{x}_{n+1}),$$

where $\alpha > 0$ is the reflection coefficient.

4. **Expansion**: If $f(\mathbf{p}_r)$ is better than $f(\mathbf{x}_1)$, expand the simplex by extending the reflection:

$$\mathbf{p}_e = \mathbf{x}_c + \gamma(\mathbf{p}_r - \mathbf{x}_c),$$

where $\gamma > 1$ is the expansion coefficient.

5. **Contraction**: If $f(\mathbf{p}_r)$ is worse than $f(\mathbf{x}_n)$, perform a contraction. The contraction can be either outside or inside, depending on the performance of the reflection:

- **Outside Contraction** (closer to the original simplex than the reflection):

$$\mathbf{p}_o = \mathbf{x}_c + \beta(\mathbf{p}_r - \mathbf{x}_c),$$

where $0 < \beta < 1$ is the outside contraction coefficient.

- **Inside Contraction** (inside the original simplex):

$$\mathbf{p}_i = \mathbf{x}_c - \beta(\mathbf{x}_c - \mathbf{x}_{n+1}),$$

where $0 < \beta < 1$ is the inside contraction coefficient.

6. **Shrink**: If the reflection, expansion, or contraction do not improve the function, shrink the simplex towards the best vertex:

$$\mathbf{x}_i = \mathbf{x}_1 + \delta(\mathbf{x}_i - \mathbf{x}_1) \quad \text{for all } i = 2, \ldots, n+1,$$

where $0 < \delta < 1$ is the shrink coefficient.

7. **Termination**: Repeat the steps above until convergence, typically when the size of the simplex becomes sufficiently small or the change in function values between iterations falls below a certain threshold.

The Nelder-Mead method is simple to implement and works well for many types of functions, but it may struggle with high-dimensional problems or functions with many local minima.

6.6 Gradient Descent Method

The Gradient Descent method is a first-order optimization algorithm used to minimize functions by iteratively moving in the direction of the steepest descent, i.e., the negative of the gradient. This method assumes that the function $f(\mathbf{x})$ is differentiable, and the gradient $\nabla f(\mathbf{x})$ can be computed.

6.6.1 Algorithm

1. **Initialization**: Choose an initial point $\mathbf{x}_0$ and a learning rate $\eta > 0$.

2. **Iteration**: Update the current point $\mathbf{x}_k$ using the gradient:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \nabla f(\mathbf{x}_k).$$

3. **Termination**: Repeat the iteration until convergence, typically when the gradient $\nabla f(\mathbf{x}_k)$ becomes very small or the change in function value is below a certain threshold.

The convergence of convergence gradient descent method can be slow, especially for functions with ill-conditioned Hessians. Modifications such as adaptive learning rates can enhance the performance of the basic gradient descent algorithm.

6.7 Newton Method

Newton Method is a second-order optimization technique that utilizes both the gradient and the Hessian matrix (second-order partial derivatives) of the function to find the minimum. The method is known for its fast convergence near the optimum, especially for convex functions.

6.7.1 Local extremes in $\mathbb{R}^n$

The Newton method is derived from an analytical approach to find local extremes of function $f : \mathbb{R}^n \to \mathbb{R}$. To find the extreme points (minima, maxima, or saddle points) of function $f : \mathbb{R}^n \to \mathbb{R}$ analytically, the following steps are generally used:

1. **Compute the Gradient (∇f)**: The gradient of f is a vector of all its first partial derivatives:
$$\nabla f(x_1, x_2, \ldots, x_n) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right)$$
The points where $\nabla f = 0$ are the critical points, which are candidates for extreme points.

2. **Set the Gradient to Zero**: Solve the system of equations obtained by setting each component of the gradient equal to zero:
$$\nabla f(x_1, x_2, \ldots, x_n) = 0$$
This gives you the critical points $\mathbf{x}^*$.

3. **Compute the Hessian Matrix ($H(f)$)**: The Hessian matrix is the matrix of all second partial derivatives:
$$H(f) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$
The Hessian matrix evaluated at a critical point gives information about the nature of the extreme.

6.7.2 Algorithm

The key idea is to solve the system of equations given by the gradient of the function $\nabla f = 0$ using Newton method for systems of nonlinear equation.

1. **Initialization**: Start with an initial guess $\mathbf{x}_0$.

2. **Iteration**: Update the current point $\mathbf{x}_k$ using the Newton step:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{H}^{-1}(\mathbf{x}_k)\nabla f(\mathbf{x}_k),$$

   where $\mathbf{H}(\mathbf{x}_k)$ is the Hessian matrix of $f(\mathbf{x})$ at $\mathbf{x}_k$.

3. **Termination**: Repeat the iteration until convergence, typically when the gradient $\nabla f(\mathbf{x}_k)$ is close to zero.

Similarly to Newton method for solving systems of nonlinear equations, Newton optimization method can converge very quickly, especially for functions that are well-approximated by a quadratic near the minimum. The main drawback is the computational cost of computing and inverting the Hessian matrix, particularly in high dimensions. Additionally, if the Hessian is not positive definite, the method may not converge.

6.8 Quasi-Newton Methods

Quasi-Newton methods are an extension of Newton method that avoid the explicit computation of the Hessian matrix. Instead, they construct an approximation to the Hessian matrix using information from the gradients.

6.8.1 Broyden-Fletcher-Goldfarb-Shanno (BFGS) Algorithm

One of the most popular Quasi-Newton methods is the BFGS algorithm. It updates an estimate of the inverse Hessian matrix using gradient information from successive iterations.

6.8.1.1 Algorithm

1. **Initialization**: Choose an initial point $\mathbf{x}_0$ and an initial Hessian approximation $\mathbf{H}_0 = \mathbf{I}$ (identity matrix).

2. **Iteration**: For each iteration k:

   (a) Compute the gradient $\mathbf{g}_k = \nabla f(\mathbf{x}_k)$.

   (b) Compute the step $\mathbf{p}_k = -\mathbf{H}_k\mathbf{g}_k$.

   (c) Update the point $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{p}_k$.

   (d) Compute the difference in gradients $\mathbf{y}_k = \mathbf{g}_{k+1} - \mathbf{g}_k$ and the difference in points $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$.

   (e) Update the Hessian approximation:

$$\mathbf{H}_{k+1} = \mathbf{H}_k + \frac{\mathbf{y}_k\mathbf{y}_k^T}{\mathbf{y}_k^T\mathbf{s}_k} - \frac{\mathbf{H}_k\mathbf{s}_k\mathbf{s}_k^T\mathbf{H}_k^T}{\mathbf{s}_k^T\mathbf{H}_k\mathbf{s}_k}.$$

3. **Termination**: Continue iterating until convergence.

About the update of Hessian approximation

The formula updates the inverse Hessian matrix at each iteration:

$$\mathbf{H}_{k+1} = \mathbf{H}_k + \frac{\mathbf{y}_k \mathbf{y}_k^T}{\mathbf{y}_k^T \mathbf{s}_k} - \frac{\mathbf{H}_k \mathbf{s}_k \mathbf{s}_k^T \mathbf{H}_k^T}{\mathbf{s}_k^T \mathbf{H}_k \mathbf{s}_k},$$

where

- $\mathbf{H}_k$: The inverse Hessian matrix approximation at iteration k.
- $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$: The difference between consecutive iterations of the variable vector $\mathbf{x}$.
- $\mathbf{y}_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)$: The difference between the gradients (first derivatives) of the function at consecutive points.

Each term can be interpreted as following:

- First Term: $\mathbf{H}_k$ is the current approximation of the inverse Hessian matrix. The new approximation $\mathbf{H}_{k+1}$ starts with this value.
- Second Term: $\frac{\mathbf{y}_k \mathbf{y}_k^T}{\mathbf{y}_k^T \mathbf{s}_k}$ is a rank-one update that adjusts $\mathbf{H}_k$ based on the curvature information provided by $\mathbf{y}_k$ and $\mathbf{s}_k$. This term ensures that the updated matrix satisfies the secant equation, a key condition in quasi-Newton methods.
- Third Term: $\frac{\mathbf{H}_k \mathbf{s}_k \mathbf{s}_k^T \mathbf{H}_k^T}{\mathbf{s}_k^T \mathbf{H}_k \mathbf{s}_k}$ is a rank-two update that corrects the approximation to ensure it remains symmetric and positive definite, properties necessary for convergence to the correct minimum.

Let us explain in detail the key second term in the BFGS formula.

$$\frac{\mathbf{y}_k \mathbf{y}_k^T}{\mathbf{y}_k^T \mathbf{s}_k},$$

is specifically designed to ensure that the updated inverse Hessian matrix satisfies the secant equation. This term is a rank-one update (since it's the outer product of two vectors), meaning it adds a small, specific adjustment to the current inverse Hessian matrix $\mathbf{H}_k$. By adding this term, the update guarantees that $\mathbf{H}_{k+1}$ satisfies the secant equation $\mathbf{H}_{k+1}\mathbf{y}_k = \mathbf{s}_k$. This adjustment corrects the previous approximation to account for the new curvature information provided by $\mathbf{y}_k$ and $\mathbf{s}_k$.

BFGS is widely used due to its balance between the simplicity of gradient descent and the rapid convergence of Newton method. It provides a practical approach to solving large-scale optimization problems where computing the full Hessian is infeasible.

6.9 Conjugate Gradient Method

The Conjugate Gradient method is a powerful technique for solving large-scale optimization problems, particularly those that involve minimizing a quadratic function or solving large systems of linear equations. It is especially effective for functions where the Hessian matrix is sparse or large, making it impractical to compute or store explicitly.

6.9.1 Algorithm

The first step of the method copies the step of the gradient descend method. Meaning we just take one step in a direction of the steepest descend of the function starting from the original guess. But then it uses information about the geometry of the surface in the upcoming steps to get to the minimum faster.

1. **Initialization**: Start with an initial guess $\mathbf{x}_0$ and compute the initial gradient $\mathbf{g}_0 = \nabla f(\mathbf{x}_0)$. Set the initial search direction $\mathbf{d}_0 = -\mathbf{g}_0$.
2. **Iteration**: For each iteration k:
   (a) Compute the step size α_k that minimizes $f(\mathbf{x}_k + \alpha_k \mathbf{d}_k)$.
   (b) Update the point $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$.
   (c) Compute the new gradient $\mathbf{g}_{k+1} = \nabla f(\mathbf{x}_{k+1})$.
   (d) Update the search direction:
   $$\mathbf{d}_{k+1} = -\mathbf{g}_{k+1} + \beta_k \mathbf{d}_k,$$
   where β_k is a scalar that ensures $\mathbf{d}_{k+1}$ is conjugate to the previous directions.
3. **Termination**: Repeat the iteration until the gradient $\mathbf{g}_{k+1}$ is sufficiently small, indicating that a minimum has been approached.

6.9.2 Computation of α_k and β_k

- **Step size** α_k: The optimal step size α_k is typically found by performing a line search along the direction $\mathbf{d}_k$.
- **Conjugate Coefficient** β_k: The coefficient β_k can be computed using different formulas, depending on the variant of the Conjugate Gradient method:
  - **Fletcher-Reeves**: This formula calculates β_k based on the norm of the gradient vectors at the current and previous steps. Geometrically, this can be seen as a measure of how much the gradient has changed in length between two consecutive iterations. If the gradient remains nearly the same in length, β_k will be close to 1, retaining much of the previous search direction. If the gradient changes significantly, β_k will adjust the new search direction accordingly, reducing or increasing the influence of the previous direction.
  $$\beta_k = \frac{\|\mathbf{g}_{k+1}\|^2}{\|\mathbf{g}_k\|^2},$$

- **Polak-Ribiere**: This version of β_k takes into account not only the norms of the gradients but also the difference between the gradients at consecutive steps. Geometrically, this reflects the change in direction of the gradient vectors. If the gradient direction changes rapidly (indicating curvature in the optimization landscape), β_k adjusts more significantly, ensuring the new search direction better accounts for this curvature.

$$\beta_k = \frac{\mathbf{g}_{k+1}^T(\mathbf{g}_{k+1} - \mathbf{g}_k)}{\|\mathbf{g}_k\|^2}.$$

The Conjugate Gradient method is particularly useful for large-scale problems where the Hessian matrix is too large to store or compute explicitly. It is commonly used in machine learning, numerical simulations, and other areas that require solving large systems of linear equations. The method is especially efficient when the objective function is quadratic, but it can also be extended to more general nonlinear functions.

6.10 Examples

Example 3:

Start with Example 2 from the this chapter 6. The following table contains measurements of the height of students from a local primary, secondary, and high school, ranging in age from 5 to 18 years. Assume that the height increases linearly for girls between the ages of 5 and some unknown age $w1$, than the slope changes, but the increase is still linear till some unknown age $w2$. For girls older than $w2$, the height remains constant. Use the two-dimensional optimization and the least-squares method to estimate the exact dependence for the given data. What height does your model predict for:

(a) A 12-year-old girl?

(b) A 5-year-old girl?

Age (years)	10.5	6.2	12.8	11.1	14.0	14.1	13.3	5.4	5.9	9.2
Height (cm)	153	122	171	152	162	159	172	115	118	152

Table 6.2: Weight and Age Data of Girls (sample of data), full data are stored in file `heigth_of_girls_2.txt`

Solution:

Similarly to example 3, to estimate the growth patterns of girls from collected age and height data, we utilize the function f, designed to optimize our piecewise linear model by determining two critical breakpoints, w_1 and w_2.

The modeling process involves several phases:

- **Before** w_1: Height growth is modeled by $f(x) = (x < w_1) \cdot (x - w_1)$ plus $g(x) = (x < w_1) \cdot (x - w_2)$, describing the linear increase in height during the early growth stages.
- **Between** w_1 **and** w_2: The growth is modeled by $g(x) = (x < w_2) \cdot (x - w_2)$, accounting for the continued linear increase at a different rate.
- **After** w_2: Height is considered constant, modeled by $h(x) = 1$, indicating no further growth.

These base functions contribute to constructing the design matrix A within the function f. This matrix, combined with the height vector Y, enables the computation of coefficients through a least squares method. The resultant model $h(x) = c_1 \cdot f(x) + c_2 \cdot g(x) + c_3 \cdot h(x)$ is derived from these calculations.

The function f accepts w_1 and w_2 as inputs and outputs the sum of residuals squared (SRS), which quantifies the discrepancy between the observed heights and those predicted by our model. By minimizing the SRS through the optimization of w_1 and w_2 in f, we enhance the model's accuracy.

The optimal values of (w_1, w_2) are $(w_{1\text{opt}}, w_{2\text{opt}}) = (8.5000, 12.6458)$. For the optimal values the model predicts the following heights:

- **For a 12-year-old girl:** The height is approximately **161.6799 cm**.
- **For a 5-year-old girl:** The height is approximately **108.9935 cm**.

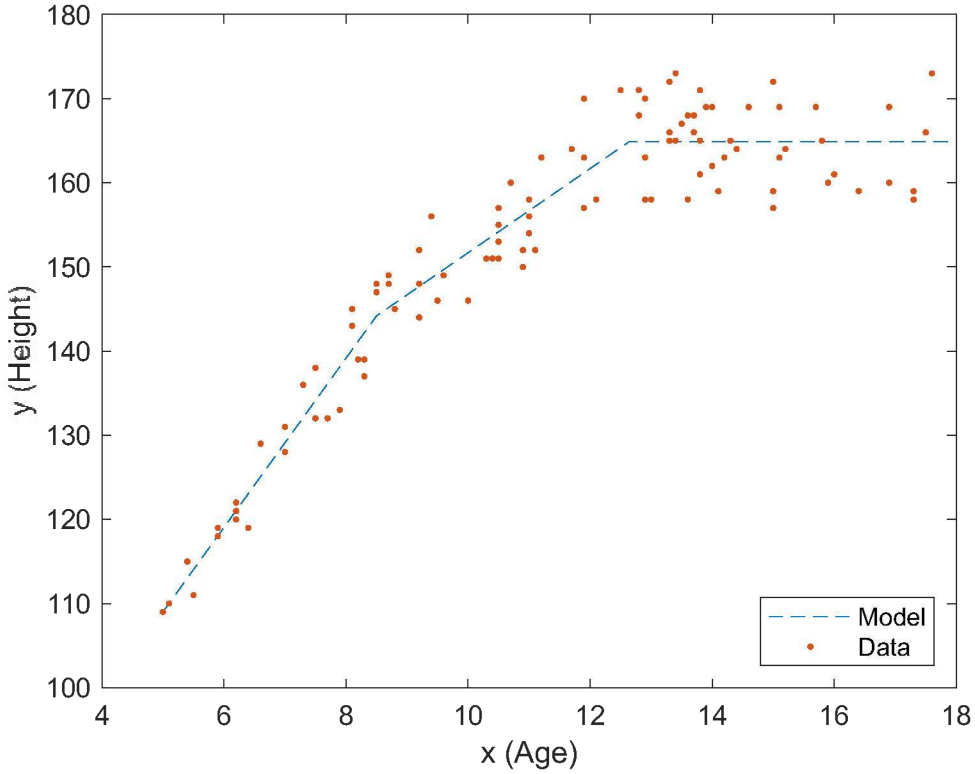


Fig. 6.9: Model of Height vs Age of Girls using two-dimensional optimization and least-square method.

Chapter 7

Numerical integration

7.1 Recommended literature

7.2 Quadrature Formulae

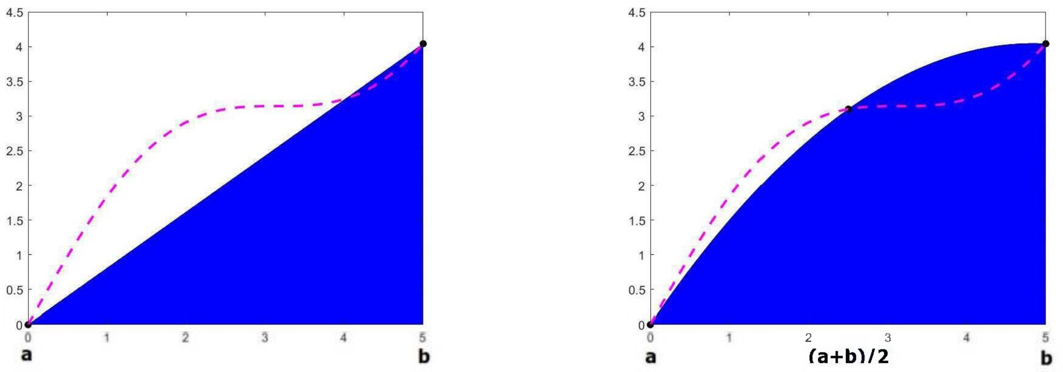
Given points $x_0, \ldots, x_n$ are such that $a \leq x_0 < x_1 < \cdots < x_n \leq b$. The function values at these points are $f_0, \ldots, f_n$, where $f_k = f(x_k)$.

To approximate the integral of a function $f(x)$ over the interval $[a, b]$, we use the interpolation polynomial $P(x)$ for the given data:

$$\int_a^b f(x)\, dx \approx \int_a^b P(x)\, dx$$

When the points are equally spaced with a step h, we obtain the Newton–Cotes formulae.

7.3 Newton–Cotes formulae



(a) Trapezoidal rule

(b) Simpson rule

Fig. 7.1: Newton–Cotes formulae to integrate $x + \sin(x)$ over interval $[0, 5]$.

Trapezoidal Rule

For $n = 1$, where $a = x_0$ and $b = x_1$, with function values $f(a)$ and $f(b)$, the interpolation polynomial $P(x)$ is:

$$P(x) = \frac{x - b}{a - b} f(a) + \frac{x - a}{b - a} f(b)$$

The integral of this polynomial over $[a, b]$ is given by:

$$\int_a^b P(x)\,dx = \frac{f(a) + f(b)}{2}(b - a)$$

Error: The error for the Trapezoidal rule is given by:

$$E(f) = \frac{1}{12}h^3 f^{(2)}(\xi)$$

where ξ is some point in the interval $[a, b]$.

Simpson Rule

For $n = 2$, let $x_0 = a$, $x_1 = a + h = \frac{a+b}{2}$, and $x_2 = a + 2h = b$. The function values are f_0, f_1, and f_2, where $f_i = f(x_i)$. The interpolation polynomial $P(x)$ is:

$$P(x) = f_0\frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} + f_1\frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} + f_2\frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}$$

The integral of this polynomial over $[a, b]$ is:

$$\int_a^b f(x)\,dx \approx \int_a^b P(x)\,dx = \frac{b - a}{6}\left[f(a) + 4f\left(\frac{a + b}{2}\right) + f(b)\right] = \frac{h}{3}\left[f_0 + 4f_1 + f_2\right]$$

Error: The error for Simpson's rule is given by:

$$E(f) = \frac{1}{90}h^5 f^{(4)}(\xi)$$

where ξ is some point in the interval $[a, b]$.

3/8-Rule

For $n = 3$, let $x_0 = a$, $x_1 = a + h = \frac{2a+b}{3}$, $x_2 = a + 2h = \frac{a+2b}{3}$, and $x_3 = a + 3h = b$. The function values are f_0, f_1, f_2, and f_3, where $f_i = f(x_i)$. The integral of the interpolation polynomial is:

$$\int_a^b f(x)\,dx \approx \int_a^b P(x)\,dx = \frac{3h}{8}\left[f_0 + 3f_1 + 3f_2 + f_3\right]$$

Error: The error for the 3/8-rule is given by:

$$E(f) = \frac{3}{80}h^5 f^{(4)}(\xi)$$

where ξ is some point in the interval $[a, b]$.

Milne Rule

For $n = 4$, let $x_0 = a$, $x_1 = a + h$, $x_2 = a + 2h$, $x_3 = a + 3h$, and $x_4 = a + 4h = b$. The function values are f_0, f_1, f_2, f_3, and f_4, where $f_i = f(x_i)$. The integral of the interpolation polynomial is:

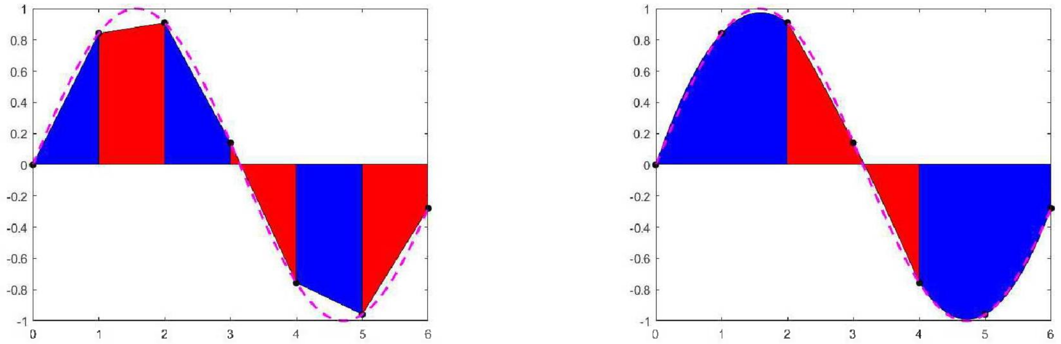
$$\int_a^b f(x)\,dx \approx \int_a^b P(x)\,dx = \frac{2h}{45}\left[7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_4\right]$$

Error: The error for Milne rule is given by:

$$E(f) = \frac{8}{945}h^7 f^{(6)}(\xi)$$

where ξ is some point in the interval $[a, b]$.

7.3.1 Composite Rules



(a) Trapezoidal rule using 6 sub-intervals

(b) Simpson rule using 3 sub-intervals.

Fig. 7.2: Newton–Cotes formulae to integrate sin(x) over interval $[0, 6]$ using 7 points.

Newton-Cotes rule	**Number of sub-intervals (m)**
Trapezoidal Rule	$m = N - 1$
Simpson Rule	$m = \frac{N-1}{2}$
3/8 Rule	$m = \frac{N-1}{3}$
Milne Rule	$m = \frac{N-1}{4}$
General Rule	$m = \frac{N-1}{k}$

Table 7.1: Number of sub-intervals m for given number of function evaluations N in Newton-Cotes composite rules

Composite Trapezoidal Rule

Given equidistant points $a = x_0 < x_1 < \cdots < b = x_n$, with $x_{i+1} = x_i + h$, the trapezoidal rule is applied to each interval $[x_i, x_{i+1}]$:

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left[f_0 + 2f_1 + 2f_2 + \cdots + 2f_{n-1} + f_n\right]$$

Composite Simpson Rule

Given equidistant points $a = x_0 < x_1 < \cdots < b = x_n$, with n even and $x_{i+1} = x_i + h$, the Simpson's rule is applied to each interval $[x_{2i}, x_{2i+2}]$:

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left[f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \cdots + 2f_{n-2} + 4f_{n-1} + f_n\right]$$

7.4 Gaussian Quadrature Formulae

The degree of precision of the quadrature formula $Q(f)$ is m if $E(P_i) = 0$ for all polynomials P_i of degree i with $0 \leq i \leq m$, and $E(P_{m+1}) \neq 0$. Gaussian quadrature formulae achieve the highest possible degree of precision, $2n+1$. These formulae allow the selection of both the points x_i and the weights A_i to optimize accuracy. Before introducing them we prove that $2n+1$ is the highest possible degree of precision.

1. The quadrature formula obtained by integrating the interpolation polynomial at points $x_0, \ldots, x_n$ has a degree of precision of at least n.

   Proof:

   Consider any polynomial $p(x)$ of degree $m \leq n$. Since $p(x)$ is of degree $m \leq n$, the interpolation polynomial $P(x)$ for this function is exactly $p(x)$ itself, i.e., $P(x) = p(x)$. Therefore:

   $$Q(f) = \int_a^b f(x)\,dx$$

   This shows that the quadrature formula is exact for any polynomial of degree $m \leq n$, meaning the error $E(f) = 0$ for all such polynomials.

   Hence, the quadrature formula has a degree of precision of at least n.

   ■

2. The quadrature formula $Q(f) = \sum_{i=0}^{n} A_i f_i$ has a degree of precision at most $2n+1$.

   Proof:

   To show that the quadrature formula has a degree of precision of at most $2n+1$, we need to show that it cannot be exact for polynomials of degree higher than $2n+1$.

   Assume that the quadrature formula is exact for all polynomials of degree $m \leq 2n+1$. Consider the polynomial:

   $$\omega(x) = \prod_{i=0}^{n}(x - x_i)$$

   This polynomial $\omega(x)$ has degree $n+1$ and vanishes at each of the interpolation points $x_0, x_1, \ldots, x_n$, i.e., $\omega(x_i) = 0$ for $i = 0, 1, \ldots, n$.

   Now consider the polynomial $\omega^2(x)$, which is of degree $2n+2$. By assumption, the quadrature formula should be exact for this polynomial if it were to have a degree of precision greater than $2n+1$. However, we know:

   $$\sum_{i=0}^{n} A_i \omega^2(x_i) = \sum_{i=0}^{n} A_i \cdot 0 = 0$$

but:

$$\int_a^b \omega^2(x)\,dx > 0$$

This contradiction shows that the quadrature formula cannot be exact for polynomials of degree $2n+2$. Therefore, the highest degree polynomial for which the quadrature formula can be exact is $2n+1$.

Hence, the quadrature formula $Q(f) = \sum_{i=0}^{n} A_i f_i$ has a degree of precision at most $2n+1$.

■

7.5 Gauss quadrature formulae

Gaussian quadrature is a powerful numerical integration technique that offers high accuracy by optimally choosing both the evaluation points (nodes) and the corresponding weights. Unlike other quadrature methods, such as the Newton-Cotes formulas, which use equally spaced nodes, Gaussian quadrature strategically selects nodes that are the roots of orthogonal polynomials with respect to a given weight function. This approach minimizes the error and maximizes the degree of precision for the integral approximation.

The central concept of Gaussian quadrature is that you can select $n+1$ nodes and their corresponding weights (in total $2n+2$ unknowns) such that the resulting formula exactly integrates polynomials of degree up to $2n+1$. This makes Gaussian quadrature particularly effective for evaluating integrals where the integrand can be well-approximated by a polynomial.

The integral over $[a,b]$ with a weight function $w(x)$ is given by:

$$\int_a^b w(x)f(x)\,dx = \sum_{i=0}^{n} A_i f(x_i) + E(f)$$

where $w(x)$ is the weight function that may include singularities.

There are various types of Gaussian quadrature, each associated with a different family of orthogonal polynomials, such as Gauss-Legendre, and Gauss-Chebyshev quadrature. Each type is suited to integrals with different weight functions and domains.

7.5.1 Gauss–Legendre Integration

In Gauss–Legendre integration we integrate over interval $[-1,1]$. The nodes x_i are the roots of the Legendre polynomial $P_n(x)$, where $n+1$ is the number of points in the quadrature.

$$P_n(x) = \frac{1}{2^n n!}\frac{d^n}{dx^n}\left((x^2-1)^n\right)$$

These roots are symmetric about zero. The weights w_i associated with each node x_i are non-negative and are found using the relation:

$$w_i = \frac{2}{(1-x_i^2)\,[P_n'(x_i)]^2}$$

n	Nodes x_i	Weights w_i	Legendre Polynomial $P_n(x)$
0	$x_1 = 0$	$w_1 = 2$	$P_0(x) = x$
1	$x_1 = -\frac{1}{\sqrt{3}}$ $x_2 = \frac{1}{\sqrt{3}}$	$w_1 = 1$ $w_2 = 1$	$P_1(x) = \frac{1}{2}(3x^2 - 1)$
2	$x_1 = -\sqrt{\frac{3}{5}}$ $x_2 = 0$ $x_3 = \sqrt{\frac{3}{5}}$	$w_1 = \frac{5}{9}$ $w_2 = \frac{8}{9}$ $w_3 = \frac{5}{9}$	$P_2(x) = \frac{1}{2}(5x^3 - 3x)$
3	$x_1 = -\sqrt{\frac{3}{7} + \frac{2}{7}\sqrt{\frac{6}{5}}}$ $x_2 = -\sqrt{\frac{3}{7} - \frac{2}{7}\sqrt{\frac{6}{5}}}$ $x_3 = \sqrt{\frac{3}{7} - \frac{2}{7}\sqrt{\frac{6}{5}}}$ $x_4 = \sqrt{\frac{3}{7} + \frac{2}{7}\sqrt{\frac{6}{5}}}$	$w_1 = \frac{18-\sqrt{30}}{36}$ $w_2 = \frac{18+\sqrt{30}}{36}$ $w_3 = \frac{18+\sqrt{30}}{36}$ $w_4 = \frac{18-\sqrt{30}}{36}$	$P_4(x) = \frac{1}{8}(35x^4 - 30x^2 + 3)$

Table 7.2: Gauss-Legendre Integration Coefficients and Legendre Polynomials

Properties of Legendre polynomials

- **Orthogonality:** Legendre polynomials are orthogonal with respect to the inner product defined by:

$$\langle P_m, P_n \rangle = \int_{-1}^{1} P_m(x) P_n(x)\, dx = 0 \quad \text{for } m \neq n$$

  This orthogonality ensures that the Legendre polynomials are independent, meaning no polynomial in the set can be expressed as a linear combination of the others.

- **Polynomial Basis:** The set of Legendre polynomials $\{P_0(x), P_1(x), \ldots, P_n(x)\}$ forms a basis for the space of polynomials of degree $m \leq n$. This means that any polynomial $p(x)$ of degree $m \leq n$ can be written as:

$$p(x) = c_0 P_0(x) + c_1 P_1(x) + \cdots + c_n P_n(x)$$

  where $c_0, c_1, \ldots, c_n$ are constants.

Adjustment of Nodes and Weights for the Interval $[a, b]$

When using Gauss-Legendre quadrature, the nodes x_i and weights w_i are originally defined for the standard interval $[-1, 1]$. However, if you need to integrate over a different interval $[a, b]$, you must adjust the nodes and weights accordingly. The nodes x_i in the interval $[-1, 1]$ are transformed to the corresponding nodes $\tilde{x}_i$ in the interval $[a, b]$ using the following linear transformation:

$$\tilde{x}_i = \frac{b-a}{2} x_i + \frac{b+a}{2},$$

where

- $\frac{b-a}{2}$ scales the interval $[-1,1]$ to match the length of the interval $[a,b]$.
- $\frac{b+a}{2}$ shifts the nodes from the center of $[-1,1]$ to the center of $[a,b]$.

The weights w_i in the interval $[-1,1]$ are adjusted to match the interval $[a,b]$ by simply scaling them by the factor $\frac{b-a}{2}$:

$$\tilde{w}_i = \frac{b-a}{2} w_i.$$

With these adjustments, the Gauss-Legendre quadrature formula for integrating a function $f(x)$ over the interval $[a,b]$ with n points is:

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n} \tilde{w}_i f(\tilde{x}_i) = \frac{b-a}{2}\sum_{i=1}^{n} w_i f\left(\frac{b-a}{2}x_i + \frac{b+a}{2}\right)$$

Example 1:

Let us integrate the function $\sin(x)$ over the interval $[0,6]$ using the Gauss-Legendre quadrature formula with 1 to 4 points.

Solution: For integrating a function $f(x)$ over an interval $[a,b]$ using Gauss-Legendre quadrature with n points, the formula is:

$$\int_a^b f(x)\,dx \approx \frac{b-a}{2}\sum_{i=1}^{n} w_i f\left(\frac{b-a}{2}x_i + \frac{b+a}{2}\right)$$

Here, x_i and w_i are the nodes and weights for the standard interval $[-1,1]$. For the interval $[a,b]$, the nodes and weights are adjusted as shown in the formula.

1-Point Gauss-Legendre Quadrature:

Node: $x_1 = 0$
Weight: $w_1 = 2$

$$\int_0^6 \sin(x)\,dx \approx \frac{6-0}{2} \times \left[2 \times \sin\left(\frac{6-0}{2} \times 0 + \frac{6+0}{2}\right)\right] \approx 0.8467$$

2-Point Gauss-Legendre Quadrature:

Nodes: $x_1 = -\frac{1}{\sqrt{3}}$, $x_2 = \frac{1}{\sqrt{3}}$
Weights: $w_1 = 1$, $w_2 = 1$

$$\int_0^6 \sin(x)\,dx \approx \frac{6-0}{2} \times \left[1 \times \sin\left(3 \times \left(-\frac{1}{\sqrt{3}}\right) + 3\right) + 1 \times \sin\left(3 \times \frac{1}{\sqrt{3}} + 3\right)\right] \approx -0.1395$$

3-Point Gauss-Legendre Quadrature:

Nodes: $x_1 = -\sqrt{\frac{3}{5}}$, $x_2 = 0$, $x_3 = \sqrt{\frac{3}{5}}$
Weights: $w_1 = \frac{5}{9}$, $w_2 = \frac{8}{9}$, $w_3 = \frac{5}{9}$

$$\int_0^6 \sin(x)\,dx \approx \frac{6-0}{2} \times \left[\frac{5}{9} \times \sin\left(3 \times \left(-\sqrt{\frac{3}{5}}\right) + 3\right) + \frac{8}{9} \times \sin(3) + \frac{5}{9} \times \sin\left(3 \times \sqrt{\frac{3}{5}} + 3\right)\right] \approx 0.0546$$

4-Point Gauss-Legendre Quadrature:

Nodes: $x_1 = -\sqrt{\frac{3}{7} + \frac{2}{7}\sqrt{\frac{6}{5}}}$, $x_2 = -\sqrt{\frac{3}{7} - \frac{2}{7}\sqrt{\frac{6}{5}}}$, $x_3 = \sqrt{\frac{3}{7} - \frac{2}{7}\sqrt{\frac{6}{5}}}$, $x_4 = \sqrt{\frac{3}{7} + \frac{2}{7}\sqrt{\frac{6}{5}}}$
Weights: $w_1 = w_4 = \frac{18-\sqrt{30}}{36}$, $w_2 = w_3 = \frac{18+\sqrt{30}}{36}$

$$\int_0^6 \sin(x)\,dx \approx 0.0392$$

7.5.2 Gauss–Chebyshev Integration

Gauss–Chebyshev integration uses the interval $[-1, 1]$ with the weight function $w(x) = \frac{1}{\sqrt{1-x^2}}$. The integration points x_i are the roots of the Chebyshev polynomials $T_n(x)$, defined by:

$$T_n(x) = \cos(n \arccos(x)).$$

n	Nodes x_i	Weights w_i	Chebyshev Polynomial $T_n(x)$
0	$x_1 = \cos\left(\frac{\pi}{2}\right) = 0$	$w_1 = \frac{\pi}{1} = \pi$	$T_1(x) = x$
1	$x_1 = \cos\left(\frac{3\pi}{4}\right) = -\frac{1}{\sqrt{2}}$ $x_2 = \cos\left(\frac{\pi}{4}\right) = \frac{1}{\sqrt{2}}$	$w_1 = \frac{\pi}{2}$ $w_2 = \frac{\pi}{2}$	$T_2(x) = 2x^2 - 1$
2	$x_1 = \cos\left(\frac{5\pi}{6}\right) = -\frac{\sqrt{3}}{2}$ $x_2 = \cos\left(\frac{\pi}{2}\right) = 0$ $x_3 = \cos\left(\frac{\pi}{6}\right) = \frac{\sqrt{3}}{2}$	$w_1 = \frac{\pi}{3}$ $w_2 = \frac{\pi}{3}$ $w_3 = \frac{\pi}{3}$	$T_3(x) = 4x^3 - 3x$
3	$x_1 = \cos\left(\frac{7\pi}{8}\right) = -\cos\left(\frac{\pi}{8}\right)$ $x_2 = \cos\left(\frac{5\pi}{8}\right) = -\cos\left(\frac{3\pi}{8}\right)$ $x_3 = \cos\left(\frac{3\pi}{8}\right) = \cos\left(\frac{5\pi}{8}\right)$ $x_4 = \cos\left(\frac{\pi}{8}\right) = \cos\left(\frac{7\pi}{8}\right)$	$w_1 = \frac{\pi}{4}$ $w_2 = \frac{\pi}{4}$ $w_3 = \frac{\pi}{4}$ $w_4 = \frac{\pi}{4}$	$T_4(x) = 8x^4 - 8x^2 + 1$

Table 7.3: Gauss-Chebyshev Integration Coefficients and Chebyshev Polynomials

Again, these polynomials are orthogonal over $[-1, 1]$ with the weight function $w(x)$ and set of all Chebyshev polynomials up to given n forms a basis for the space of polynomials of degree up to n. The Gauss–Chebyshev quadrature rule for an integral is given by:

$$\int_{-1}^{1} \frac{f(x)}{\sqrt{1-x^2}}\,dx \approx \sum_{i=1}^{n} A_i f(x_i)$$

where x_i are the roots of the Chebyshev polynomial $T_n(x)$ and the weights $A_i = \frac{\pi}{n}$ are constant.

7.6 Monte Carlo integration

The idea behind this method is simple yet powerful. We simulate random points uniformly distributed over the rectangle $[a,b] \times [0,M]$. The fraction of points that lie below the curve $f(x)$ approximates the area under the curve. This approximation becomes more accurate as the number of random points increases, aligning with the law of large numbers, which states that the average of a large number of independent and identically distributed random variables converges to the expected value.

Let f be a non-negative function defined on the interval $[a,b]$, and suppose $f(x) \leq M$ for every $x \in [a,b]$. Consider the scenario where $[X_1,Y_1],\ldots,[X_n,Y_n]$ are observations of the random vector $[X,Y]$ distributed uniformly on the rectangle $[a,b] \times [0,M]$. The probability that Y is less than or equal to $f(X)$ can be expressed as:

$$P(Y \leq f(X)) = \frac{\int_a^b f(x)\,dx}{M(b-a)}$$

This probability can be approximated using the following Monte Carlo method:

$$P(Y \leq f(X)) \approx \frac{1}{n}\sum_{i=1}^{n} I_{Y_i \leq f(X_i)}$$

where I is the indicator function, which equals 1 if $Y_i \leq f(X_i)$ and 0 otherwise. Thus, the integral $\int_a^b f(x)\,dx$ can be approximated as:

$$\int_a^b f(x)\,dx \approx \frac{M(b-a)}{n}\sum_{i=1}^{n} I_{Y_i \leq f(X_i)}$$

This method leverages the law of large numbers, where the approximation becomes more accurate as the number of observations n increases.

Example 4: Estimation of $\int_0^{\frac{\pi}{2}} \sin(x)\,dx$

Let us use the Monte Carlo method to estimate the integral:

$$\int_0^{\frac{\pi}{2}} \sin(x)\,dx$$

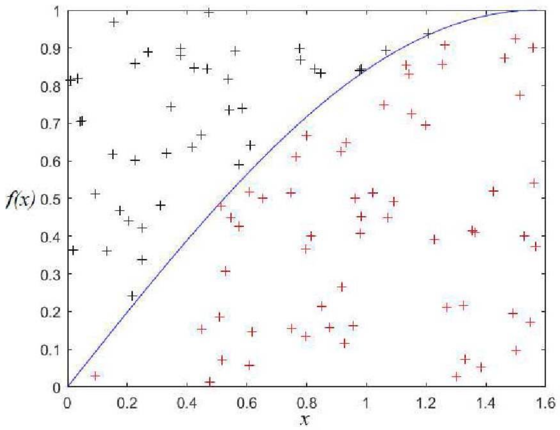


Fig. 7.3: Montecarlo integration of function $\sin(x)$.

Solution:

We can apply the above method by choosing $f(x) = \sin(x)$, $a = 0$, $b = \frac{\pi}{2}$, and $M = 1$ (since $\sin(x) \leq 1$ for all $x \in [0, \frac{\pi}{2}]$). By generating random points $[X_i, Y_i]$ in the rectangle $[0, \frac{\pi}{2}] \times [0, 1]$, we can estimate the integral using:

$$\int_0^{\frac{\pi}{2}} \sin(x)\, dx \approx \frac{\frac{\pi}{2} \cdot 1}{n} \sum_{i=1}^{n} I_{Y_i \leq \sin(X_i)}$$

Example 5: Approximation of π

The Monte Carlo method can also be used to approximate π. Consider a square of side length 2, centered at the origin, and a circle of radius 1 inscribed in the square. The area of the square is 4, and the area of the circle is π.

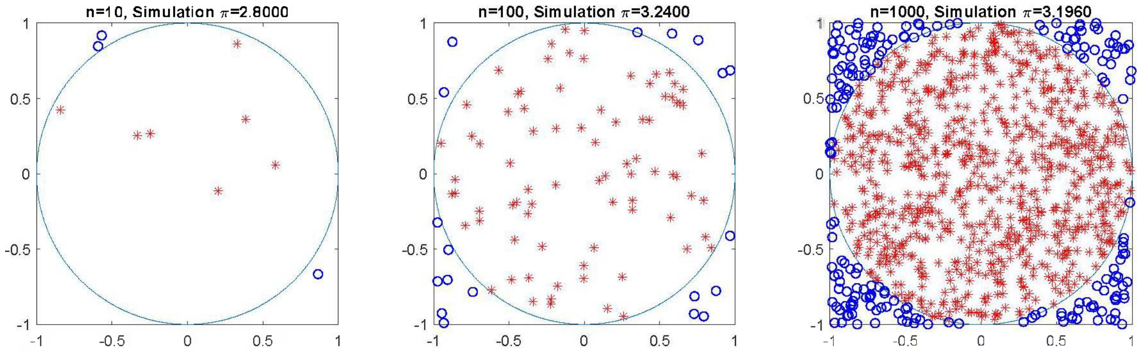


Fig. 7.4: Approximation of π using Monte Carlo method.

Solution:

To approximate π, generate random coordinates (x, y) uniformly distributed in the square $[-1, 1] \times [-1, 1]$. The fraction of points that fall inside the circle, determined by the condition $x^2 + y^2 < 1$, is:

$$\frac{\text{Number of points inside the circle}}{\text{Total number of points}} \approx \frac{\pi}{4}$$

Thus, π can be estimated as:

$$\pi \approx 4 \times \frac{\text{Number of points inside the circle}}{\text{Total number of points}}$$

By generating $n = 10, 100, 1000$ points and counting the number of points inside the circle, we can estimate π and compare the results with the true value.

Example 6: Volume of an Ellipsoid

The Monte Carlo method can also be extended to three dimensions to calculate the volume of an ellipsoid with principal semiaxes $a = 2$, $b = 3$, and $c = 1$.

Solution:

The volume of the ellipsoid is given by:

$$V = \frac{4}{3}\pi abc = \frac{4}{3}\pi \times 2 \times 3 \times 1 = 8\pi$$

To approximate this volume using the Monte Carlo method, generate random coordinates (x, y, z) uniformly distributed in the cuboid $[-2, 2] \times [-3, 3] \times [-1, 1]$. The fraction of points that lie inside the ellipsoid, determined by the condition:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} < 1$$

is used to estimate the volume:

$$V \approx \frac{V_{\text{cuboid}}}{n} \times \text{Number of points inside the ellipsoid}$$

where $V_{\text{cuboid}} = 8 \times 6 \times 2 = 96$. By repeating this process $k = 100$ times with $n = 1000$ points, we can obtain a reliable estimate of the ellipsoid's volume and compare it with the true volume 8π.

Chapter 8

Numerical differentiation and solving differential equations

8.1 Recommended literature

[14] Josef Stoer et al. *Introduction to numerical analysis.* Vol. 1993. Springer, 1980
[13] L Richard. “Burden and J. Douglas Faires. Numerical analysis. Brooks”. In: *Cole Publishing Company, Pacific Grove, California* 93950 (1997), p. 78

8.2 Numerical Computation of Derivative

Numerical computation of derivatives is essential in solving differential equations, where exact analytical solutions are often unavailable. The interpolation polynomial and its derivative, alongside Taylor series expansions, provide effective methods for approximating derivatives from discrete data points.

8.2.1 Derivative Approximation Using Interpolation Polynomial

Given a set of data points $(x_0, f_0), (x_1, f_1), \ldots, (x_n, f_n)$, we can approximate the derivative of the function $f(x)$ using the derivative of the interpolation polynomial $P(x)$ that fits these points. The general approach is:

- **First-Order Difference:** When $n = 1$ (i.e., two points x_0 and x_1), the interpolation polynomial $P(x)$ is linear, and its derivative at any point x within the interval is given by:

  $$f'(x) \approx P'(x) = \frac{f_1 - f_0}{x_1 - x_0}$$

  This represents a simple finite difference approximation of the derivative.

- **Second-Order Difference:** When $n = 2$ (i.e., three points x_0, x_1, and x_2), the interpolation polynomial $P(x)$ is quadratic, and is given by:

  $$P(x) = f_0\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + f_1\frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} + f_2\frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}$$

  The derivative of this polynomial, $P'(x)$, can be calculated as:

  $$P'(x) = f_0\frac{2x-x_1-x_2}{(x_0-x_1)(x_0-x_2)} + f_1\frac{2x-x_0-x_2}{(x_1-x_0)(x_1-x_2)} + f_2\frac{2x-x_0-x_1}{(x_2-x_0)(x_2-x_1)}$$

For equally spaced points, where $x_1 - x_0 = x_2 - x_1 = h$, this simplifies to:

$$P'(x) = \frac{1}{2h^2}\left[f_0(2x - x_1 - x_2) - 2f_1(2x - x_0 - x_2) + f_2(2x - x_0 - x_1)\right]$$

Specifically, at the given points:

$$P'(x_0) = \frac{1}{2h}(-3f_0 + 4f_1 - f_2)$$

$$P'(x_1) = \frac{1}{2h}(f_2 - f_0)$$

$$P'(x_2) = \frac{1}{2h}(f_0 - 4f_1 + 3f_2)$$

The second derivative of the interpolation polynomial $P''(x)$ at these points is given by:

$$P''(x) = \frac{1}{h^2}(f_0 - 2f_1 + f_2).$$

8.2.2 Derivative Approximation Using Taylor Series

Taylor series expansions provide another method to approximate derivatives. Given a function $f(x)$ and its Taylor series expansion around a point x, we can derive approximations for the first and second derivatives:

- **First Derivative:** Using the Taylor series expansions of $f(x+h)$ and $f(x-h)$:

  $$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

  This central difference formula is widely used for its second-order accuracy.

  We may derive the formula the following way:

  - Taylor Series Expansion of $f(x+h)$:

    $$f(x+h) = f(x) + f'(x)h + \frac{f''(x)h^2}{2} + \frac{f'''(x)h^3}{6} + O(h^4)$$

  - Taylor Series Expansion of $f(x-h)$:

    $$f(x-h) = f(x) - f'(x)h + \frac{f''(x)h^2}{2} - \frac{f'''(x)h^3}{6} + O(h^4)$$

  Now, subtract the expansion of $f(x-h)$ from $f(x+h)$:

  $$f(x+h) - f(x-h) = 2f'(x)h + \frac{f'''(x)h^3}{3} + O(h^4)$$

  Dividing both sides by $2h$, we get the central difference approximation for the first derivative:

  $$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

- **Second Derivative:** Using the Taylor series expansions of $f(x+h)$ and $f(x-h)$, but adding them instead of subtracting, we get:

  $$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2)$$

  This formula is also second-order accurate and is commonly used in numerical differentiation.

8.3 Numerical Solution of Ordinary Differential Equations

Numerical methods are essential for solving ordinary differential equations (ODEs) when analytical solutions are unavailable. The Euler methods—explicit and implicit—are foundational techniques in this area.

8.3.1 Explicit Euler Method

The explicit Euler method is straightforward and computes the next value of the solution using the current value and the derivative:

$$y_{n+1} = y_n + hf(x_n, y_n)$$

This method is easy to implement but may suffer from stability issues, especially for stiff equations.

8.3.2 Implicit Euler Method

The implicit Euler method improves stability by using the derivative at the next time step:

$$y_{n+1} = y_n + hf(x_{n+1}, y_{n+1})$$

Although more stable, this method requires solving an implicit equation at each step, often necessitating iterative solvers.

8.3.3 Predictor-Corrector Methods

Predictor-corrector methods combine the simplicity of the explicit Euler method with the stability of the implicit method. The **predictor** step estimates the solution using an explicit method, and the **corrector** step refines this estimate using an implicit method.

8.3.4 Solving ODEs Using Runge-Kutta Methods

Runge-Kutta methods are a powerful and widely used family of numerical techniques for solving ordinary differential equations (ODEs). These methods improve upon simpler approaches, such as the Euler method, by providing higher-order accuracy while maintaining relatively low computational complexity.

8.3.4.1 The Fourth-Order Runge-Kutta Method (RK4)

The fourth-order Runge-Kutta method (RK4) is the most popular and commonly used variant of the Runge-Kutta family. It strikes an excellent balance between accuracy and computational effort, making it the method of choice for many practical applications.

Given an initial value problem of the form:

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0,$$

the RK4 method advances the solution from t_n to $t_{n+1} = t_n + \Delta t$ by calculating four intermediate slopes:

$$\begin{aligned}
k_1 &= f(t_n, y_n), \\
k_2 &= f\left(t_n + \frac{\Delta t}{2}, y_n + \frac{\Delta t}{2} k_1\right), \\
k_3 &= f\left(t_n + \frac{\Delta t}{2}, y_n + \frac{\Delta t}{2} k_2\right), \\
k_4 &= f(t_n + \Delta t, y_n + \Delta t k_3).
\end{aligned}$$

The solution at the next time step is then given by:

$$y_{n+1} = y_n + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4).$$

Here's what each step represents:

- k_1 is the slope at the beginning of the time step.
- k_2 is the slope at the midpoint of the interval, using the slope from k_1.
- k_3 is another midpoint slope, but using k_2 to refine the estimate.
- k_4 is the slope at the end of the interval, incorporating all previous estimates.

By combining these slopes with appropriate weights, RK4 effectively captures the behavior of the solution across the entire interval, leading to a fourth-order accurate method. This means that the error per step is on the order of $O(\Delta t^5)$, and the total accumulated error across the entire interval is $O(\Delta t^4)$.

Runge-Kutta 4(5), or RK45, is an extension of the RK4 method that provides adaptive step size control. RK45 simultaneously computes two solutions—one using a fourth-order method and another using a fifth-order method. The difference between these two solutions provides an estimate of the local truncation error. This error estimate is then used to adjust the step size dynamically, making the method both efficient and accurate. RK45 is particularly useful for problems where the solution exhibits rapid changes in certain regions, as it automatically reduces the step size to maintain accuracy and increases it when possible to save computational time.

8.4 Second-Order Differential Equations with Boundary Conditions

Second-order differential equations are common in various physical systems, including mechanical vibrations, electrical circuits, and fluid dynamics. Solving these equations numerically involves discretizing the equation using finite differences, which converts the differential equation into a system of algebraic equations.

Given a second-order differential equation of the form:

$$y''(x) + p(x)y'(x) + q(x)y(x) = r(x)$$

with boundary conditions $y(a) = \alpha$ and $y(b) = \beta$, we can discretize the domain into equally spaced points $x_0, x_1, \ldots, x_n$ with step size h. The second derivative at each point can be approximated using:

$$y''(x_i) \approx \frac{y_{i-1} - 2y_i + y_{i+1}}{h^2}$$

Substituting this into the differential equation gives a system of linear equations that can be solved for $y_0, y_1, \ldots, y_n$. The boundary conditions are used to set the values at the endpoints.

8.5 Numerical Solution of Partial Differential Equations

Partial differential equations (PDEs) describe the behavior of physical systems involving multiple independent variables, such as time and space. In the one-dimensional case, we consider PDEs where the quantity of interest depends on a single spatial variable x and time t. Common examples include the heat equation, the wave equation, and the advection equation.

8.5.1 Discretization of the Spatial and Temporal Domains

The numerical solution of a one-dimensional PDE typically involves discretizing both the spatial domain and the temporal domain. This process converts the continuous PDE into a set of algebraic equations that can be solved using computational methods.

8.5.1.1 Spatial Discretization

Consider the spatial domain $x \in [a, b]$. We discretize this domain into M equally spaced points:

$$x_i = a + i\Delta x, \quad \text{for } i = 0, 1, \ldots, M,$$

where $\Delta x = \frac{b-a}{M}$ is the spatial step size. The solution $u(x,t)$ at these points is approximated by $u_i(t) = u(x_i, t)$.

8.5.1.2 Temporal Discretization

Similarly, the time domain $t \in [0, T]$ is discretized into N time steps:

$$t^n = n\Delta t, \quad \text{for } n = 0, 1, \ldots, N,$$

where $\Delta t = \frac{T}{N}$ is the time step size. The solution at time t^n is approximated by $u_i^n = u(x_i, t^n)$.

8.5.2 Finite Difference Method

The finite difference method is one of the most commonly used techniques for numerically solving PDEs. It involves approximating the derivatives in the PDE with finite differences.

8.5.2.1 Approximation of Spatial Derivatives

Consider a PDE involving the spatial derivative $\frac{\partial u}{\partial x}$ or the second-order spatial derivative $\frac{\partial^2 u}{\partial x^2}$. These derivatives can be approximated using finite difference formulas:

- **First-Order Derivative:** The first derivative can be approximated using the central difference formula:

  $$\left.\frac{\partial u}{\partial x}\right|_{x_i} \approx \frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x}.$$

  Alternatively, forward or backward difference schemes can be used:

  $$\left.\frac{\partial u}{\partial x}\right|_{x_i} \approx \frac{u_{i+1}^n - u_i^n}{\Delta x} \quad \text{(forward difference)},$$

$$\left.\frac{\partial u}{\partial x}\right|_{x_i} \approx \frac{u_i^n - u_{i-1}^n}{\Delta x} \quad \text{(backward difference).}$$

- **Second-Order Derivative:** The second derivative can be approximated using the central difference formula:

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{x_i} \approx \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}.$$

8.5.2.2 Approximation of Temporal Derivatives

For the time derivative $\frac{\partial u}{\partial t}$, we can use similar finite difference approximations:

- **Forward Difference (Explicit):** The forward difference formula approximates the time derivative as:

$$\left.\frac{\partial u}{\partial t}\right|_{t^n} \approx \frac{u_i^{n+1} - u_i^n}{\Delta t}.$$

  This is commonly used in explicit time-stepping schemes.

- **Backward Difference (Implicit):** The backward difference formula is given by:

$$\left.\frac{\partial u}{\partial t}\right|_{t^n} \approx \frac{u_i^n - u_i^{n-1}}{\Delta t}.$$

  This is typically used in implicit time-stepping schemes, which are more stable for stiff problems.

- **Central Difference:** The central difference formula for the time derivative is:

$$\left.\frac{\partial u}{\partial t}\right|_{t^n} \approx \frac{u_i^{n+1} - u_i^{n-1}}{2\Delta t}.$$

  This is often used in schemes that require a higher order of accuracy.

8.5.3 Explicit and Implicit Schemes

Based on the finite difference approximations, we can develop explicit or implicit numerical schemes to solve the PDE.

8.5.3.1 Explicit Scheme

In an explicit scheme, the solution at the next time step u_i^{n+1} is calculated directly from known values at the current or previous time steps. For example, applying the explicit scheme to the heat equation:

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

we get:

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

Explicit schemes are straightforward to implement but may require very small time steps Δt to ensure stability, particularly when the spatial step size Δx is small.

8.5.3.2 Implicit Scheme

In an implicit scheme, the solution at the next time step u_i^{n+1} depends on unknown values at the same time step. For the heat equation, the implicit scheme might look like:

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

This equation is implicit because u_i^{n+1} appears on both sides. Implicit schemes are generally more stable and allow for larger time steps, but they require solving a system of linear equations at each time step, which can be computationally expensive.

8.5.4 Boundary and Initial Conditions

The numerical solution of PDEs requires specifying boundary and initial conditions.

8.5.4.1 Initial Conditions

For time-dependent PDEs, initial conditions specify the state of the system at the start of the simulation, i.e., $u(x, 0) = f(x)$. These conditions are used to initialize the numerical scheme.

8.5.4.2 Boundary Conditions

Boundary conditions specify the behavior of the solution at the edges of the spatial domain. Common types of boundary conditions include:

- **Dirichlet Boundary Conditions:** Specify the value of the solution at the boundaries, e.g., $u(a, t) = g_1(t)$ and $u(b, t) = g_2(t)$.
- **Neumann Boundary Conditions:** Specify the value of the derivative of the solution at the boundaries, e.g., $\frac{\partial u}{\partial x}(a, t) = h_1(t)$ and $\frac{\partial u}{\partial x}(b, t) = h_2(t)$.
- **Robin Boundary Conditions:** A combination of Dirichlet and Neumann conditions, where a linear combination of the function value and its derivative is specified at the boundaries.

8.6 Examples

Example 1:

Consider the second-order differential equation:

$$y'' + y = \cos(x)$$

with boundary conditions $y(0) = 0$ and $y\left(\frac{\pi}{2}\right) = 1$.

1. Discretize the interval $[0, \frac{\pi}{2}]$ with points $x_0 = 0$, $x_1 = \frac{\pi}{4}$, and $x_2 = \frac{\pi}{2}$.
2. Use the formula for the second derivative:
$$y''(x) \approx \frac{16}{\pi^2}[y(x_0) - 2y(x_1) + y(x_2)] = \frac{16}{\pi^2}[0 - 2y(\frac{\pi}{4}) + 1]$$
3. Substitute this into the differential equation:
$$\frac{16}{\pi^2}[0 - 2y(\frac{\pi}{4}) + 1] + y(x) = \cos(x)$$
4. Evaluate at $x_1 = \frac{\pi}{4}$:
$$\frac{16}{\pi^2}[0 - 2y(\frac{\pi}{4}) + 1] + y\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2}$$
5. Solve for $y\left(\frac{\pi}{4}\right)$:
$$y\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}\pi^2 - 32}{2(\pi^2 - 32)}$$

When the size step size $h = \frac{a+b}{2}$, we may compute the solution in point $\frac{b-a}{2}$ using a single linear equation of one unknown. For smaller step size h this approach leads to system of three-diagonal system of linear equations. In this concrete example, we use step size $h = \frac{\pi}{20}$ allowing to get solution $y(x)$ in nine points $y_1, \ldots y_9$. This leads to system of nine linear equations:

$$\begin{gathered}
40.528\, y_2 - 80.057\, y_1 = 0.98769 \\
40.528\, y_1 - 80.057\, y_2 + 40.528\, y_3 = 0.95106 \\
40.528\, y_2 - 80.057\, y_3 + 40.528\, y_4 = 0.89101 \\
40.528\, y_3 - 80.057\, y_4 + 40.528\, y_5 = 0.80902 \\
40.528\, y_4 - 80.057\, y_5 + 40.528\, y_6 = 0.70711 \\
40.528\, y_5 - 80.057\, y_6 + 40.528\, y_7 = 0.58779 \\
40.528\, y_6 - 80.057\, y_7 + 40.528\, y_8 = 0.45399 \\
40.528\, y_7 - 80.057\, y_8 + 40.528\, y_9 = 0.30902 \\
40.528\, y_8 - 80.057\, y_9 = -40.372
\end{gathered}$$

The system is thee-diagonal with

$$A = \begin{pmatrix}
-80.0569 & 40.5285 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
40.5285 & -80.0569 & 40.5285 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 40.5285 & -80.0569 & 40.5285 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 40.5285 & -80.0569 & 40.5285 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 40.5285 & -80.0569 & 40.5285 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 40.5285 & -80.0569 & 40.5285 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 40.5285 & -80.0569 & 40.5285 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 40.5285 & -80.0569 & 40.5285 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 40.5285 & -80.0569
\end{pmatrix},$$

$$b = \begin{pmatrix} 0.9877 \\ 0.9511 \\ 0.8910 \\ 0.8090 \\ 0.7071 \\ 0.5878 \\ 0.4540 \\ 0.3090 \\ -40.3720 \end{pmatrix},$$

and is effectively solvable using exactly by Crout or Doolitle decomposition, or numerically by relaxation method or two-grid method.

For this particular example analytical solution is known and can be expressed as $y(x) = \sin(x)(\frac{x}{2} + 1 - \frac{\pi}{4})$. When we compare it visually with the solution obtained numerically, the differences in results are almost unnoticeable.

Example 2: Van der Pol Oscillator and Limit Cycle Detection

The Van der Pol oscillator is a well-known example of a nonlinear dynamical system that exhibits a limit cycle, which is a closed trajectory in phase space that the system's trajectories converge to, regardless of initial conditions. The governing equation for the Van der Pol oscillator is given by:

$$\frac{d^2y}{dt^2} - \mu(1 - y^2)\frac{dy}{dt} + y = 0,$$

where μ is a parameter that controls the nonlinearity and the strength of the damping.

Conversion of the Second-Order ODE to a System of First-Order ODEs

To solve this equation numerically, it is often advantageous to convert the second-order differential equation into a system of first-order differential equations. This is a standard technique in numerical analysis, as many numerical methods are designed specifically for first-order systems.

Given the second-order ODE:

$$\frac{d^2y}{dt^2} = \mu(1 - y^2)\frac{dy}{dt} - y,$$

we introduce a new variable $z = \frac{dy}{dt}$, which represents the first derivative of y with respect to time t. By substituting z into the original equation, we obtain a system of two first-order ODEs:

$$\frac{dy}{dt} = z,$$

$$\frac{dz}{dt} = \mu(1 - y^2)z - y.$$

This system now describes the same dynamics as the original second-order equation but in a form that is more amenable to numerical methods.

Numerical Solution of the Van der Pol Oscillator

The Van der Pol oscillator can be solved as either an initial value problem (IVP) or a boundary value problem (BVP), depending on the specifics of the physical situation or mathematical formulation.

Initial Value Problem

In the initial value problem (IVP), the solution is determined by specifying the initial conditions at a single point in time, typically at $t = t_0$. The goal is to find a function $y(t)$ that satisfies the differential equations and adheres to these initial conditions as the system evolves over time. For the Van der Pol oscillator, with initial conditions $y(t_0) = y_0$ and $z(t_0) = z_0$, we can employ various numerical methods, such as the explicit Euler method, the implicit Euler method, or the Runge-Kutta methods, to solve the system:

$$\frac{dy}{dt} = z, \quad \frac{dz}{dt} = \mu(1 - y^2)z - y.$$

Explicit Euler Method:

$$y_{n+1} = y_n + \Delta t \cdot z_n,$$
$$z_{n+1} = z_n + \Delta t \cdot \left[\mu(1 - y_n^2)z_n - y_n\right].$$

Implicit Euler Method:

$$y_{n+1} = y_n + \Delta t \cdot z_{n+1},$$
$$z_{n+1} = z_n + \Delta t \cdot \left[\mu(1 - y_{n+1}^2)z_{n+1} - y_{n+1}\right].$$

Predictor-Corrector Method:

1. Predictor (Explicit Euler):

$$y_{n+1}^{(\text{predict})} = y_n + \Delta t \cdot z_n,$$
$$z_{n+1}^{(\text{predict})} = z_n + \Delta t \cdot \left[\mu(1 - y_n^2)z_n - y_n\right].$$

2. Corrector (Implicit Euler):

$$y_{n+1} = y_n + \frac{\Delta t}{2} \cdot (z_n + z_{n+1}^{(\text{predict})}),$$
$$z_{n+1} = z_n + \frac{\Delta t}{2} \cdot \left[\mu(1 - y_n^2)z_n - y_n + \mu(1 - (y_{n+1}^{(\text{predict})})^2)z_{n+1}^{(\text{predict})} - y_{n+1}^{(\text{predict})}\right].$$

These methods are typically used to simulate the Van der Pol oscillator over time, allowing us to observe the evolution of the system from given initial conditions.

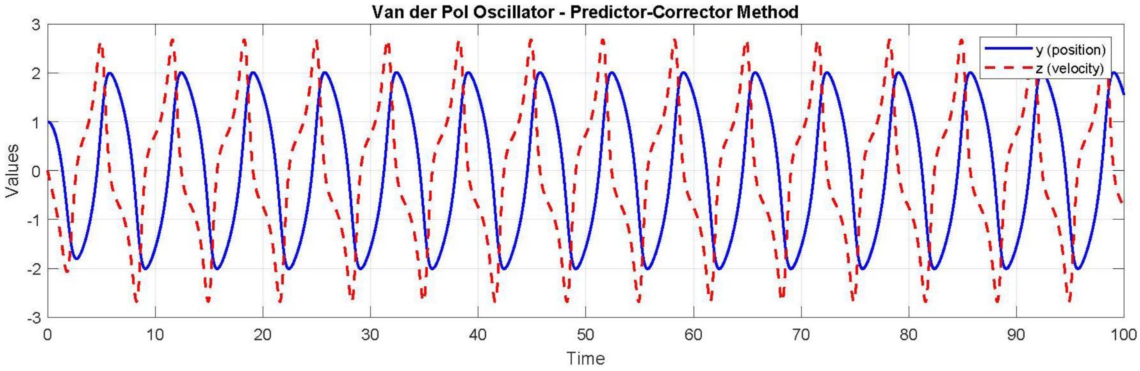


Fig. 8.1: Solution of the Van der Pol Oscillator equation with $\mu = 1.0$, total time span $T = 100$ and step size $dt = 0.01$ using the predictor corrector method using explict Euler as predictor and implicit Euler as corrector.

Boundary Value Problem

In a boundary value problem (BVP), the solution is determined by specifying the conditions at two or more points, typically at the boundaries of the domain, such as $t = t_0$ and $t = t_f$. The goal is to find a function $y(t)$ that satisfies the differential equations and meets these boundary conditions. For instance, we may need to find $y(t)$ that satisfies:

$$y(t_0) = y_0, \quad y(t_f) = y_f.$$

To solve the Van der Pol oscillator as a boundary value problem, we can use numerical methods such as the finite difference method or the shooting method. In the shooting method, we treat the BVP as an initial value problem, guessing the initial conditions at $t = t_0$ and iterating until the boundary condition at $t = t_f$ is satisfied.

Limit Cycle Detection as a Boundary Value Problem

Limit cycle detection in dynamical systems, such as the Van der Pol oscillator, can be framed as a boundary value problem (BVP). Instead of focusing on the initial conditions alone, we seek to determine a periodic solution $y(t)$ that satisfies the system's equations over one complete cycle, ensuring that the solution returns to the same state after a period T. For the Van der Pol oscillator, the system is given by:

$$\frac{d^2y}{dt^2} - \mu(1 - y^2)\frac{dy}{dt} + y = 0,$$

which, when converted into a system of first-order ODEs, becomes:

$$\frac{dy}{dt} = z,$$

$$\frac{dz}{dt} = \mu(1 - y^2)z - y.$$

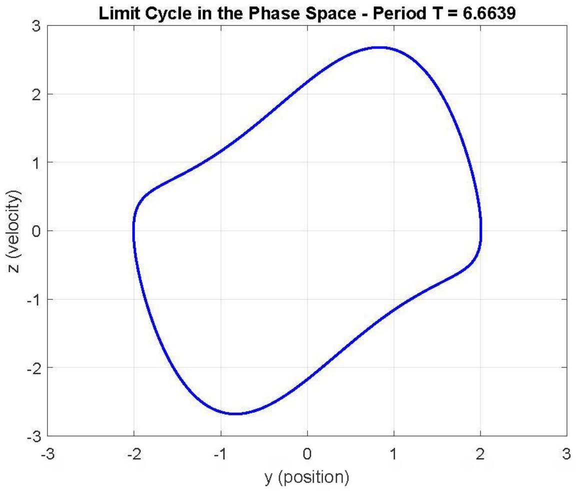


Fig. 8.2: Shooting Method to Find Limit Cycle of the Van der Pol Oscillator equation with $\mu = 1.0$, transient time to settle on the limit cycle $T = 500$, and step size $dt = 0.0001$ using secant method to solve $y(0) - y(T) = 0$, where $y(0)$ is initial condition after transient, and using initial guess of period $T_0 = 6, T_1 = 7$.

To detect a limit cycle, we impose the periodic boundary conditions:

$$y(0) = y(T), \quad z(0) = z(T),$$

where T is the unknown period of the oscillation. The goal is to find T, along with the functions $y(t)$ and $z(t)$, such that these conditions are satisfied.

This can be approached by treating the detection of the limit cycle as a BVP where we seek to find a periodic solution that satisfies:

$$\mathbf{F}(y(0), z(0), T) = \begin{pmatrix} y(0) - y(T) \\ z(0) - z(T) \end{pmatrix} = \mathbf{0}.$$

Here, $\mathbf{F}$ represents the difference between the states at the start and end of one period.

Numerical Approach

A numerical method suitable for this kind of BVP is the shooting method, where we iteratively adjust the initial conditions and the period T to satisfy the periodic boundary conditions. Specifically, we:

1. Guess an initial period T and initial conditions $y(0)$ and $z(0)$.
2. Integrate the system of ODEs from $t = 0$ to $t = T$ using a numerical method such as the Euler method.
3. Evaluate the difference $y(T) - y(0)$ and $z(T) - z(0)$ obtained from integration and the initial values $y(0)$ and $z(0)$. Compute a norm of the vector of differences.
4. Adjust T, $y(0)$, and $z(0)$ iteratively using a derivative free root-finding method to minimize the difference and satisfy $y(0) = y(T)$ or/and $z(0) = z(T)$.

When the conditions are met, the solution corresponds to a limit cycle of the system. By formulating the problem in this way, the periodicity of the limit cycle is enforced through the boundary conditions, ensuring that the solution is indeed periodic and that the system has settled into a stable oscillatory pattern.