

Lecture Notes
Numerical Analysis

Prof. Boris Houska

Contents

1	Error Analysis	3
2	Linear Equations	38
3	Iterative Methods	104
4	Polynomial Interpolation	137
5	Applications of Polynomial Interpolation	203
6	Gauss Approximation	239
7	Numerical Integration	270
8	Newton's Method	309
9	Gauss-Newton Methods	358
10	Nonlinear Differential Equations	377

Chapter 1

Error Analysis

Error Analysis

- Introduction
- Representation of Numbers in a Computer
- Numerical Computations: Stability and Error Analysis
- Application: Numerical Differentiation

Contents

- Introduction
- Representation of Numbers in a Computer
- Numerical Computations: Stability and Error Analysis
- Application: Numerical Differentiation

Objectives

In this lecture we will learn about

- the fact that many computer programs store numbers with finite precision only
- the IEEE standard for storing floating point numbers
- applications, including numerical differentiation

Contents

- Introduction
- Representation of Numbers in a Computer
- Numerical Computations: Stability and Error Analysis
- Application: Numerical Differentiation

Scientific Computing

Computers or calculators typically store numbers with finite precision:

- Example 1: $8 + 8 == 16$?
- Example 2: $(\sqrt{5})^2 == 5$?
- Example 3: $1.1 + 0.1 == 1.2$?

Let's try this with JULIA:

```
julia> (1.1 + 0.1) == 1.2  
false
```

```
julia> 1.1 + 0.1  
1.2000000000000002
```

Problem: numerical error: $\approx 2 * 10^{-16}$.

Scientific Computing

Computers or calculators typically store numbers with finite precision:

- Example 1: $8 + 8 == 16$?
- Example 2: $(\sqrt{5})^2 == 5$?
- Example 3: $1.1 + 0.1 == 1.2$?

Let's try this with JULIA:

```
julia> (1.1 + 0.1) == 1.2  
false
```

```
julia> 1.1 + 0.1  
1.2000000000000002
```

Problem: numerical error: $\approx 2 * 10^{-16}$.

Scientific Computing

Computers or calculators typically store numbers with finite precision:

- Example 1: $8 + 8 == 16$?
- Example 2: $(\sqrt{5})^2 == 5$?
- Example 3: $1.1 + 0.1 == 1.2$?

Let's try this with JULIA:

<pre>julia>(1.1 + 0.1) == 1.2 false</pre>	<pre>julia> 1.1 + 0.1 1.2000000000000002</pre>
--	---

Problem: numerical error: $\approx 2 * 10^{-16}$.

Floating Point Numbers

IEEE standard for double-precision floating point numbers:

$$x = \pm(1 + m) \cdot 2^e \quad \text{with} \quad m = \sum_{i=1}^{52} m_i 2^{-i} \quad \text{and} \quad e = \sum_{i=0}^{10} c_i 2^i - \bar{c},$$

Names: m = mantissa, e = exponent.

Storage requirement:

- 1 bit to store the sign.
- 11 bits to store $c_{10}, \dots, c_0 \in \{0, 1\}$; offset $\bar{c} = 1023$.
- 52 bits to store $m_1, \dots, m_{52} \in \{0, 1\}$.

In total: $(1 + 11 + 52)$ bits = 64 bits = 8 bytes.

Floating Point Numbers

IEEE standard for double-precision floating point numbers:

$$x = \pm(1 + m) \cdot 2^e \quad \text{with} \quad m = \sum_{i=1}^{52} m_i 2^{-i} \quad \text{and} \quad e = \sum_{i=0}^{10} c_i 2^i - \bar{c},$$

Names: m = mantissa, e = exponent.

Storage requirement:

- 1 bit to store the sign.
- 11 bits to store $c_{10}, \dots, c_0 \in \{0, 1\}$; offset $\bar{c} = 1023$.
- 52 bits to store $m_1, \dots, m_{52} \in \{0, 1\}$.

In total: $(1 + 11 + 52)$ bits = 64 bits = 8 bytes.

Floating Point Numbers

IEEE standard for double-precision floating point numbers:

$$x = \pm(1 + m) \cdot 2^e \quad \text{with} \quad m = \sum_{i=1}^{52} m_i 2^{-i} \quad \text{and} \quad e = \sum_{i=0}^{10} c_i 2^i - \bar{c},$$

Names: m = mantissa, e = exponent.

Storage requirement:

- 1 bit to store the sign.
- 11 bits to store $c_{10}, \dots, c_0 \in \{0, 1\}$; offset $\bar{c} = 1023$.
- 52 bits to store $m_1, \dots, m_{52} \in \{0, 1\}$.

In total: $(1 + 11 + 52)$ bits = 64 bits = 8 bytes.

A closer look at the example

```
julia>bits(1.1)
```

```
"0011111111110001100110011001100110011001100110011001100110011010"
```

```
julia>bits(0.1)
```

```
"001111111011100110011001100110011001100110011001100110011010"
```

```
julia>bits(1.2)
```

```
"001111111111001100110011001100110011001100110011001100110011"
```

```
julia>bits(1.1+0.1)
```

```
"001111111111001100110011001100110011001100110011001100110100"
```

A closer look at the example

```
julia>bits(1.1)
```

```
"001111111110001100110011001100110011001100110011001100110011010"
```

```
julia>bits(0.1)
```

```
"0011111110111001100110011001100110011001100110011001100110011010"
```

```
julia>bits(1.2)
```

```
"001111111110011001100110011001100110011001100110011001100110011"
```

```
julia>bits(1.1+0.1)
```

```
"0011111111110011001100110011001100110011001100110011001100110100"
```

A closer look at the example

```
julia>bits(1.1)
```

```
"001111111111000110011001100110011001100110011001100110011010"
```

```
julia>bits(0.1)
```

```
"001111111011100110011001100110011001100110011001100110011010"
```

```
julia>bits(1.2)
```

```
"001111111111001100110011001100110011001100110011001100110011"
```

```
julia>bits(1.1+0.1)
```

```
"0011111111111001100110011001100110011001100110011001100110100"
```


A closer look at the example

```
julia>bits(1.1)
```

```
"0011111111110001100110011001100110011001100110011001100110011010"
```

```
julia>bits(0.1)
```

```
"001111111011100110011001100110011001100110011001100110011010"
```

```
julia>bits(1.2)
```

```
"001111111111001100110011001100110011001100110011001100110011"
```

```
julia>bits(1.1+0.1)
```

```
"001111111111001100110011001100110011001100110011001100110100"
```

Floating Point Numbers

- Numbers between 1 and $1 + 2^{-52}$ cannot be represented.
- The (relative) rounding $\text{eps} = 2^{-52}$ is called *machine precision*.
- The absolute rounding error $\text{eps} * 2^e$ depends on exponent e .
(if we work with larger numbers, we get larger rounding errors)

Important to remember: $\text{eps} = 2^{-52} \approx 2 * 10^{-16}$.

Floating Point Numbers

- Numbers between 1 and $1 + 2^{-52}$ cannot be represented.
- The (relative) rounding $\text{eps} = 2^{-52}$ is called *machine precision*.
- The absolute rounding error $\text{eps} * 2^e$ depends on exponent e .
(if we work with larger numbers, we get larger rounding errors)

Important to remember: $\text{eps} = 2^{-52} \approx 2 * 10^{-16}$.

Contents

- Introduction
- Representation of Numbers in a Computer
- Numerical Computations: Stability and Error Analysis
- Application: Numerical Differentiation

Numerical function evaluation

Let us evaluate the function

$$\Phi(x) = \sin(10^8 x)$$

at $x = \pi$. The exact solution is $\Phi(\pi) = 0$.

```
julia>sin(10^8 pi)  
-3.9082928156687315e - 8
```

Caution: The function Φ is ill-conditioned, i.e., the evaluation error is much larger than $\text{eps} \approx 2 * 10^{-16}$!

Numerical function evaluation

Let us evaluate the function

$$\Phi(x) = \sin(10^8 x)$$

at $x = \pi$. The exact solution is $\Phi(\pi) = 0$.

```
julia>sin(10^8 pi)  
-3.9082928156687315e - 8
```

Caution: The function Φ is ill-conditioned, i.e., the evaluation error is much larger than $\text{eps} \approx 2 * 10^{-16}$!

Numerical function evaluation

Let us evaluate the function

$$\Phi(x) = \sin(10^8 x)$$

at $x = \pi$. The exact solution is $\Phi(\pi) = 0$.

```
julia>sin(10^8 pi)  
-3.9082928156687315e - 8
```

Caution: The function Φ is ill-conditioned, i.e., the evaluation error is much larger than $\text{eps} \approx 2 * 10^{-16}$!

Storing Numbers in a Computer

There exists a variety of ways to represent numbers:

- Floating point numbers be it 64bit (“double precision”) or 32bit (“single precision”).
- Integers are often stored differently. Remark:
`julia>bits(3)` is not the same as `julia>bits(3.)!!!`
- Arbitrary precision arithmetics are an alternative (not our focus).
- Verified arithmetics store intervals rather than single numbers.

Storing Numbers in a Computer

There exists a variety of ways to represent numbers:

- Floating point numbers be it 64bit (“double precision”) or 32bit (“single precision”).

- Integers are often stored differently. Remark:

`julia>bits(3)` is not the same as `julia>bits(3.)!!!`

- Arbitrary precision arithmetics are an alternative (not our focus).
- Verified arithmetics store intervals rather than single numbers.

Storing Numbers in a Computer

There exists a variety of ways to represent numbers:

- Floating point numbers be it 64bit (“double precision”) or 32bit (“single precision”).
- Integers are often stored differently. Remark:
`julia>bits(3)` is not the same as `julia>bits(3.)!!!`
- Arbitrary precision arithmetics are an alternative (not our focus).
- Verified arithmetics store intervals rather than single numbers.

Storing Numbers in a Computer

There exists a variety of ways to represent numbers:

- Floating point numbers be it 64bit (“double precision”) or 32bit (“single precision”).
- Integers are often stored differently. Remark:
`julia>bits(3)` is not the same as `julia>bits(3.)`!!!
- Arbitrary precision arithmetics are an alternative (not our focus).
- Verified arithmetics store intervals rather than single numbers.

Contents

- Introduction
- Representation of Numbers in a Computer
- Numerical Computations: Stability and Error Analysis
- Application: Numerical Differentiation

Finite Differences

The derivative of a twice continuously differentiable function $f : \mathbb{R} \rightarrow \mathbb{R}$ can be approximated by finite differences:

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x+h) - f(x)}{h}$$

- The mathematical approximation error, given by

$$\left| \frac{f(x+h) - f(x)}{h} - \frac{\partial f}{\partial x}(x) \right| \approx \frac{h}{2} \left| \frac{\partial^2 f}{\partial x^2}(x) \right| = \mathbf{O}(h),$$

tends to zero for $h \rightarrow 0$.

- The numerical error is approximately

$$\mathbf{O}\left(\frac{\text{eps}}{h}\right)$$

- If f is well conditioned, we choose

$$h \approx \operatorname{argmin}_h \left(h + \frac{\text{eps}}{h} \right) = \sqrt{\text{eps}}.$$

Finite Differences

The derivative of a twice continuously differentiable function $f : \mathbb{R} \rightarrow \mathbb{R}$ can be approximated by finite differences:

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x+h) - f(x)}{h}$$

- The mathematical approximation error, given by

$$\left| \frac{f(x+h) - f(x)}{h} - \frac{\partial f}{\partial x}(x) \right| \approx \frac{h}{2} \left| \frac{\partial^2 f}{\partial x^2}(x) \right| = \mathbf{O}(h),$$

tends to zero for $h \rightarrow 0$.

- The numerical error is approximately

$$\mathbf{O}\left(\frac{\text{eps}}{h}\right)$$

- If f is well conditioned, we choose

$$h \approx \operatorname{argmin}_h \left(h + \frac{\text{eps}}{h} \right) = \sqrt{\text{eps}}.$$

Finite Differences

The derivative of a twice continuously differentiable function $f : \mathbb{R} \rightarrow \mathbb{R}$ can be approximated by finite differences:

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x+h) - f(x)}{h}$$

- The mathematical approximation error, given by

$$\left| \frac{f(x+h) - f(x)}{h} - \frac{\partial f}{\partial x}(x) \right| \approx \frac{h}{2} \left| \frac{\partial^2 f}{\partial x^2}(x) \right| = \mathbf{O}(h),$$

tends to zero for $h \rightarrow 0$.

- The numerical error is approximately

$$\mathbf{O}\left(\frac{\text{eps}}{h}\right)$$

- If f is well conditioned, we choose

$$h \approx \operatorname{argmin}_h \left(h + \frac{\text{eps}}{h} \right) = \sqrt{\text{eps}}.$$

Finite Differences

The derivative of a twice continuously differentiable function $f : \mathbb{R} \rightarrow \mathbb{R}$ can be approximated by finite differences:

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x+h) - f(x)}{h}$$

- The mathematical approximation error, given by

$$\left| \frac{f(x+h) - f(x)}{h} - \frac{\partial f}{\partial x}(x) \right| \approx \frac{h}{2} \left| \frac{\partial^2 f}{\partial x^2}(x) \right| = \mathbf{O}(h),$$

tends to zero for $h \rightarrow 0$.

- The numerical error is approximately

$$\mathbf{O}\left(\frac{\text{eps}}{h}\right)$$

- If f is well conditioned, we choose

$$h \approx \operatorname{argmin}_h \left(h + \frac{\text{eps}}{h} \right) = \sqrt{\text{eps}}.$$

Central Differences

In order to reduce the mathematical approximation error, we can use central differences

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x+h) - f(x-h)}{2h}$$

to approximate the derivative of f .

- The mathematical approximation error is now

$$\left| \frac{f(x+h) - f(x-h)}{2h} - \frac{\partial f}{\partial x}(x) \right| \leq \mathbf{O}(h^2).$$

- The numerical error is still in the order of

$$\frac{\text{eps}}{h} = \mathbf{O}\left(\frac{\text{eps}}{h}\right)$$

- In practice, if f is well conditioned, we choose $h \approx \sqrt[3]{\text{eps}}$.

Central Differences

In order to reduce the mathematical approximation error, we can use central differences

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x+h) - f(x-h)}{2h}$$

to approximate the derivative of f .

- The mathematical approximation error is now

$$\left| \frac{f(x+h) - f(x-h)}{2h} - \frac{\partial f}{\partial x}(x) \right| \leq \mathbf{O}(h^2).$$

- The numerical error is still in the order of

$$\frac{\text{eps}}{h} = \mathbf{O}\left(\frac{\text{eps}}{h}\right)$$

- In practice, if f is well conditioned, we choose $h \approx \sqrt[3]{\text{eps}}$.

Central Differences

In order to reduce the mathematical approximation error, we can use central differences

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x+h) - f(x-h)}{2h}$$

to approximate the derivative of f .

- The mathematical approximation error is now

$$\left| \frac{f(x+h) - f(x-h)}{2h} - \frac{\partial f}{\partial x}(x) \right| \leq \mathbf{O}(h^2).$$

- The numerical error is still in the order of

$$\frac{\text{eps}}{h} = \mathbf{O}\left(\frac{\text{eps}}{h}\right)$$

- In practice, if f is well conditioned, we choose $h \approx \sqrt[3]{\text{eps}}$.

Central Differences

In order to reduce the mathematical approximation error, we can use central differences

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x+h) - f(x-h)}{2h}$$

to approximate the derivative of f .

- The mathematical approximation error is now

$$\left| \frac{f(x+h) - f(x-h)}{2h} - \frac{\partial f}{\partial x}(x) \right| \leq \mathbf{O}(h^2).$$

- The numerical error is still in the order of

$$\frac{\text{eps}}{h} = \mathbf{O}\left(\frac{\text{eps}}{h}\right)$$

- In practice, if f is well conditioned, we choose $h \approx \sqrt[3]{\text{eps}}$.

Summary

- Programs often store numbers with finite precision only.
- IEEE double precision floating point numbers: $\text{eps} \approx 2 * 10^{-16}$.
- Error and stability analysis is important!
- Application: numerical differentiation.

Chapter 2

Linear Equations

Linear Equations

- Problem Formulation
- Conditioning of Linear Equation Systems
- Gauss Elimination
- Linear Equations with Band-Structured Matrix
- Linear Equations with Positive Definite Matrices
- QR-Decomposition

Contents

- Problem Formulation
- Conditioning of Linear Equation Systems
- Gauss Elimination
- Linear Equations with Band-Structured Matrix
- Linear Equations with Positive Definite Matrices
- QR-Decomposition

Problem Formulation

Given a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $b \in \mathbb{R}^m$. We are searching for solutions of the linear equation system

$$Ax = b .$$

Names:

- For $m < n$: “under-determined linear equation system”
- For $m = n$: “quadratic linear equation system”
- For $m > n$: “over-determined linear equation system”

Problem Formulation

Given a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $b \in \mathbb{R}^m$. We are searching for solutions of the linear equation system

$$Ax = b .$$

Names:

- For $m < n$: “under-determined linear equation system”
- For $m = n$: “quadratic linear equation system”
- For $m > n$: “over-determined linear equation system”

Problem Formulation

Given a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $b \in \mathbb{R}^m$. We are searching for solutions of the linear equation system

$$Ax = b .$$

Names:

- For $m < n$: “under-determined linear equation system”
- For $m = n$: “quadratic linear equation system”
- For $m > n$: “over-determined linear equation system”

Problem Formulation

Given a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $b \in \mathbb{R}^m$. We are searching for solutions of the linear equation system

$$Ax = b .$$

Names:

- For $m < n$: “under-determined linear equation system”
- For $m = n$: “quadratic linear equation system”
- For $m > n$: “over-determined linear equation system”

Known Results from Linear Algebra

The equation system $Ax = b$ has a solution if and only if $\text{rank}(A) = \text{rank}(A, b)$.

This condition can, e.g., be checked with Gram-Schmidt algorithms.

For the quadratic case $n = m$, the following statements are equivalent:

- $Ax = b$ has the unique solution x .
- $\text{rank}(A) = n$.
- $\det(A) \neq 0$.
- All eigenvalues of A are different from 0.

Known Results from Linear Algebra

The equation system $Ax = b$ has a solution if and only if $\text{rank}(A) = \text{rank}(A, b)$.

This condition can, e.g., be checked with Gram-Schmidt algorithms.

For the quadratic case $n = m$, the following statements are equivalent:

- $Ax = b$ has the unique solution x .
- $\text{rank}(A) = n$.
- $\det(A) \neq 0$.
- All eigenvalues of A are different from 0.

Contents

- Problem Formulation
- **Conditioning of Linear Equation Systems**
- Gauss Elimination
- Linear Equations with Band-Structured Matrix
- Linear Equations with Positive Definite Matrices
- QR-Decomposition

Overview: Perturbation Theory

Whenever we solve equations of the form $Ax = b$ numerically, we have to take into account two possible sources of errors:

- Errors that are due to working with inexact numerical values for A and b .
- Errors that are due to numerical rounding problems when implementing the algorithm based on finite precision arithmetics.

Matrix Norms

Recall that any vector norm $\|\cdot\| : \mathbb{R}^n \rightarrow \mathbb{R}$ can be generalized for matrices by defining

$$\|A\| = \max_{x \neq 0} \frac{\|Ax\|}{\|x\|},$$

for any $A \in \mathbb{R}^{m \times n}$.

Simple examples:

- $\|A\|_1 = \max_j \sum_{i=0}^m |A_{i,j}|.$
- $\|A\|_\infty = \max_i \sum_{j=0}^n |A_{i,j}|.$

The Spectral Norm

The eigenvalues $\lambda_1, \dots, \lambda_n \in \mathbb{C}$ of a square matrix $A \in \mathbb{R}^{n \times n}$ are the roots of the characteristic polynomial $p(\lambda) = \det(A - \lambda I)$.

The matrix 2-norm is given by

$$\|A\|_2 = \max_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2} = \max \left\{ \sqrt{\lambda} \mid \det(A^T A - \lambda I) = 0 \right\} ,$$

If A is symmetric this expression for the 2-norm can be simplified

$$\|A\|_2 = \max \{ |\lambda| \mid \det(A - \lambda I) = 0 \} .$$

The Spectral Norm

The eigenvalues $\lambda_1, \dots, \lambda_n \in \mathbb{C}$ of a square matrix $A \in \mathbb{R}^{n \times n}$ are the roots of the characteristic polynomial $p(\lambda) = \det(A - \lambda I)$.

The matrix 2-norm is given by

$$\|A\|_2 = \max_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2} = \max \left\{ \sqrt{\lambda} \mid \det(A^T A - \lambda I) = 0 \right\} ,$$

If A is symmetric this expression for the 2-norm can be simplified

$$\|A\|_2 = \max \{ |\lambda| \mid \det(A - \lambda I) = 0 \} .$$

A Useful Matrix-Norm Inequality

If $\|\cdot\|$ is a matrix norm (assume $\|I\| = 1$) and $A \in \mathbb{R}^{n \times n}$ a given matrix whose norm satisfies $\|A\| < 1$, then the matrix $I + A$ is invertible and we have

$$\|(I + A)^{-1}\| \leq \frac{1}{1 - \|A\|}$$

Proof: From the inequality

$$\|(I + A)x\| \geq \underbrace{(1 - \|A\|)}_{>0} \|x\|$$

we conclude that we have $(I + A)x \neq 0$ for all vectors $x \neq 0$; that is, $I + A$ is invertible. Moreover, we have

$$1 = \|(I + A)(I + A)^{-1}\| \geq \|(I + A)^{-1}\|(1 - \|A\|),$$

which implies the statement.

A Useful Matrix-Norm Inequality

If $\|\cdot\|$ is a matrix norm (assume $\|I\| = 1$) and $A \in \mathbb{R}^{n \times n}$ a given matrix whose norm satisfies $\|A\| < 1$, then the matrix $I + A$ is invertible and we have

$$\|(I + A)^{-1}\| \leq \frac{1}{1 - \|A\|}$$

Proof: From the inequality

$$\|(I + A)x\| \geq \underbrace{(1 - \|A\|)}_{>0} \|x\|$$

we conclude that we have $(I + A)x \neq 0$ for all vectors $x \neq 0$; that is, $I + A$ is invertible. Moreover, we have

$$1 = \|(I + A)(I + A)^{-1}\| \geq \|(I + A)^{-1}\|(1 - \|A\|),$$

which implies the statement.

Condition Numbers

Theorem:

Let x denote the solution of the invertible and quadratic linear equation system $Ax = b$ and let δx be such that

$$(A + \delta A)(x + \delta x) = b + \delta b .$$

assuming $\|\delta A\| < \frac{1}{\|A^{-1}\|}$ and $\|b\| \neq 0$ as well as $\|A\| \neq 0$. Then we have

$$\frac{\|\delta x\|}{\|x\|} \leq \frac{\text{cond}(A)}{1 - \text{cond}(A) \frac{\|\delta A\|}{\|A\|}} \left(\frac{\|\delta b\|}{\|b\|} + \frac{\|\delta A\|}{\|A\|} \right) \quad \text{with} \quad \text{cond}(A) = \|A\| \cdot \|A^{-1}\| .$$

Interpretation:

The condition number $\text{cond}(A)$ can be interpreted as an error amplification factor,

$$\frac{\|\delta x\|}{\|x\|} \approx \text{cond}(A) \left(\frac{\|\delta b\|}{\|b\|} + \frac{\|\delta A\|}{\|A\|} \right) .$$

Condition Numbers

Theorem:

Let x denote the solution of the invertible and quadratic linear equation system $Ax = b$ and let δx be such that

$$(A + \delta A)(x + \delta x) = b + \delta b .$$

assuming $\|\delta A\| < \frac{1}{\|A^{-1}\|}$ and $\|b\| \neq 0$ as well as $\|A\| \neq 0$. Then we have

$$\frac{\|\delta x\|}{\|x\|} \leq \frac{\text{cond}(A)}{1 - \text{cond}(A) \frac{\|\delta A\|}{\|A\|}} \left(\frac{\|\delta b\|}{\|b\|} + \frac{\|\delta A\|}{\|A\|} \right) \quad \text{with} \quad \text{cond}(A) = \|A\| \cdot \|A^{-1}\| .$$

Interpretation:

The condition number $\text{cond}(A)$ can be interpreted as an error amplification factor,

$$\frac{\|\delta x\|}{\|x\|} \approx \text{cond}(A) \left(\frac{\|\delta b\|}{\|b\|} + \frac{\|\delta A\|}{\|A\|} \right) .$$

Proof

From $Ax = b$ and the equation

$$(A + \delta A)(x + \delta x) = b + \delta b .$$

we find $\delta x = (A + \delta A)^{-1}(\delta b - \delta Ax)$ (recall that $\|\delta A\| \leq \frac{1}{\|A^{-1}\|}$ implies that $A + \delta A$ is invertible). This yields the estimate

$$\begin{aligned}\|\delta x\| &\leq \|(A + \delta A)^{-1}\| (\|\delta b\| + \|\delta A\| \|x\|) \\ &\leq \|(I + A^{-1}\delta A)^{-1}\| \|A^{-1}\| (\|\delta b\| + \|\delta A\| \|x\|) \\ &\leq \frac{\|A^{-1}\|}{1 - \|A^{-1}\| \|\delta A\|} (\|\delta b\| + \|\delta A\| \|x\|)\end{aligned}$$

(Last step has used the matrix-norm ineq. from previous slides)

Proof (continued)

Because we have $Ax = b$, it follows that $\|A\|\|x\| \geq \|b\|$ and

$$\begin{aligned}\|\delta x\| &\leq \frac{\|A^{-1}\|}{1 - \|A^{-1}\|\|\delta A\|} (\|\delta b\| + \|\delta A\|\|x\|) \\ &\leq \frac{\|A^{-1}\|\|A\|\|x\|}{1 - \|A^{-1}\|\|\delta A\|} \left(\frac{\|\delta b\|}{\|b\|} + \frac{\|\delta A\|}{\|A\|} \right).\end{aligned}$$

In the last step we substitute $\text{cond}(A) = \|A\| \cdot \|A^{-1}\|$:

$$\|\delta x\| \leq \frac{\text{cond}(A)\|x\|}{1 - \text{cond}(A)\frac{\|\delta A\|}{\|A\|}} \left(\frac{\|\delta b\|}{\|b\|} + \frac{\|\delta A\|}{\|A\|} \right).$$

The statement of the theorem follows after dividing by $\|x\|$.

Contents

- Problem Formulation
- Conditioning of Linear Equation Systems
- **Gauss Elimination**
- Linear Equations with Band-Structured Matrix
- Linear Equations with Positive Definite Matrices
- QR-Decomposition

Triangular Matrices

If the matrix A is triangular, i.e.,

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ 0 & a_{2,2} & \dots & a_{2,n} \\ \vdots & \ddots & \ddots & \\ 0 & \dots & 0 & a_{n,n} \end{pmatrix},$$

the system Ax can be solved by a “backwards substitution”:

$$x_n = \frac{b_n}{a_{n,n}} \quad \text{and} \quad x_j = \frac{1}{a_{j,j}} \left(b_j - \sum_{k=j+1}^n a_{j,k} x_k \right)$$

for $j = n - 1, \dots, 1$. (Complexity: $\mathbf{O}(n^2)$.)

Gauss Elimination

For general matrices A the main idea is to transform the matrix step by step into an upper triangular matrix. For this aim, three types of operations can be applied:

- We may swap (permute) rows of A if necessary.
- We may swap (permute) columns as long as we re-enumerate the unknowns x_j , too.
- We may multiply one row by a scalar factor and add it to another row.

Gauss Elimination

For general matrices A the main idea is to transform the matrix step by step into an upper triangular matrix. For this aim, three types of operations can be applied:

- We may swap (permute) rows of A if necessary.
- We may swap (permute) columns as long as we re-enumerate the unknowns x_j , too.
- We may multiply one row by a scalar factor and add it to another row.

Gauss Elimination

For general matrices A the main idea is to transform the matrix step by step into an upper triangular matrix. For this aim, three types of operations can be applied:

- We may swap (permute) rows of A if necessary.
- We may swap (permute) columns as long as we re-enumerate the unknowns x_j , too.
- We may multiply one row by a scalar factor and add it to another row.

Gauss Elimination

For general matrices A the main idea is to transform the matrix step by step into an upper triangular matrix. For this aim, three types of operations can be applied:

- We may swap (permute) rows of A if necessary.
- We may swap (permute) columns as long as we re-enumerate the unknowns x_j , too.
- We may multiply one row by a scalar factor and add it to another row.

Gauss Elimination

In the first step of the Gauss elimination procedure is as follows:

- Permute the rows of A such that $a_{1,1} \neq 0$.
- For all row indices $j = 2, \dots, n$: subtract $q_j = \frac{a_{j,1}}{a_{1,1}}$ times the first row from the j -th row.
- The result of these operations is a matrix of the form

$$C = \begin{pmatrix} c_{1,1} & c_{1,2} & \dots & c_{1,n} \\ 0 & c_{2,2} & \dots & c_{2,n} \\ \vdots & \vdots & & \vdots \\ 0 & c_{n,2} & \dots & c_{n,n} \end{pmatrix}.$$

Gauss Elimination

In the first step of the Gauss elimination procedure is as follows:

- Permute the rows of A such that $a_{1,1} \neq 0$.
- For all row indices $j = 2, \dots, n$: subtract $q_j = \frac{a_{j,1}}{a_{1,1}}$ times the first row from the j -th row.
- The result of these operations is a matrix of the form

$$C = \begin{pmatrix} c_{1,1} & c_{1,2} & \dots & c_{1,n} \\ 0 & c_{2,2} & \dots & c_{2,n} \\ \vdots & \vdots & & \vdots \\ 0 & c_{n,2} & \dots & c_{n,n} \end{pmatrix}.$$

Gauss Elimination

In the first step of the Gauss elimination procedure is as follows:

- Permute the rows of A such that $a_{1,1} \neq 0$.
- For all row indices $j = 2, \dots, n$: subtract $q_j = \frac{a_{j,1}}{a_{1,1}}$ times the first row from the j -th row.
- The result of these operations is a matrix of the form

$$C = \begin{pmatrix} c_{1,1} & c_{1,2} & \dots & c_{1,n} \\ 0 & c_{2,2} & \dots & c_{2,n} \\ \vdots & \vdots & & \vdots \\ 0 & c_{n,2} & \dots & c_{n,n} \end{pmatrix}.$$

Gauss Elimination

The first step of the Gauss elimination procedure can also be summarized in the form:

$$C = G_1 P_1 A \quad \text{and} \quad d = G_1 P_1 b$$

such that the equation system $Ax = b$ is equivalent to the equation system $Cx = d$. Here, P_1 is a permutation matrix and G_1 a Frobenius matrix of the form

$$G_1 = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ q_2 & 1 & 0 & \dots & 0 \\ q_3 & 0 & 1 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ q_n & 0 & \dots & 0 & 1 \end{pmatrix}.$$

Gauss Elimination

In the second step of the Gauss elimination we apply the same transformation strategy to the remaining $(n - 1) \times (n - 1)$ dense block such that

$$E = G_2 P_2 G_1 P_1 A \quad \text{and} \quad f = G_2 P_2 G_1 P_1 b$$

such that the equation system $Ax = b$ is equivalent to the equation system $Ex = f$. Here, E has the form

$$E = \begin{pmatrix} c_{1,1} & c_{1,2} & c_{1,3} & \cdots & c_{1,n} \\ 0 & e_{2,2} & e_{2,3} & \cdots & e_{2,n} \\ 0 & 0 & e_{3,3} & \cdots & e_{3,n} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & e_{n,3} & \cdots & e_{n,n} \end{pmatrix}.$$

Gauss Elimination

If we keep on applying the same strategy we end up with a triangular matrix

$$R = G_{n-1}P_{n-1} \dots G_1P_1A .$$

and a vector $s = G_{n-1}P_{n-1} \dots G_1P_1b$ such that the equation system $Ax = b$ is equivalent to the equation system $Rx = s$. The triangular system can then be solved by backwards substitution. The complexity of computing R is of order $\mathcal{O}(n^3)$.

Gauss Elimination

If we keep on applying the same strategy we end up with a triangular matrix

$$R = G_{n-1}P_{n-1} \dots G_1P_1A .$$

and a vector $s = G_{n-1}P_{n-1} \dots G_1P_1b$ such that the equation system $Ax = b$ is equivalent to the equation system $Rx = s$. The triangular system can then be solved by backwards substitution. The complexity of computing R is of order $\mathbf{O}(n^3)$.

LR Decomposition

We assume for simplicity that we do not have to permute the rows of A .

The matrix

$$L = G_1^{-1}G_2^{-1}\dots G_{n-1}^{-1} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ l_{2,1} & 1 & 0 & \dots & 0 \\ l_{3,1} & l_{3,2} & 1 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ l_{n,1} & l_{n,2} & \dots & l_{n,n-1} & 1 \end{pmatrix}$$

is lower triangular and satisfies $LR = A$ by construction.

Uniqueness of LR-Decomposition

The LR-Decomposition is unique: if there were two LR decompositions

$$A = L_1 R_1 = L_2 R_2$$

we would have

$$L_2^{-1} L_1 = R_2 R_1^{-1} = I ,$$

since the matrix $L_2^{-1} L_1$ is lower triangular (with ones on the diagonal) and R is an upper triangular matrix. Thus, we have $R_1 = R_2$ and $L_1 = L_2$.

Application of LR Decomposition in Practice

In practice, we are often in the situation that we want to solve multiple linear equation systems of the form

$$Ax_i = b_i, \quad i = 1, 2, \dots, N$$

for changing vectors $b_i \in \mathbb{R}^n$. A naive implementation of Gauss elimination would lead to the complexity $\mathbf{O}(N \cdot n^3)$ for computing $x_1, \dots, x_N$.

A major speed-up can be obtained by storing the LR-decomposition, $A = LR$, of the matrix A and then compute

$$x_1 = R^{-1}L^{-1}b_1, \quad x_2 = R^{-1}L^{-2}b_2 \dots$$

Now, the complexity is $\mathbf{O}(n^3 + N \cdot n^2)$.

Application of LR Decomposition in Practice

In practice, we are often in the situation that we want to solve multiple linear equation systems of the form

$$Ax_i = b_i, \quad i = 1, 2, \dots, N$$

for changing vectors $b_i \in \mathbb{R}^n$. A naive implementation of Gauss elimination would lead to the complexity $\mathbf{O}(N \cdot n^3)$ for computing $x_1, \dots, x_N$.

A major speed-up can be obtained by storing the LR-decomposition, $A = LR$, of the matrix A and then compute

$$x_1 = R^{-1}L^{-1}b_1, \quad x_2 = R^{-1}L^{-2}b_2 \dots$$

Now, the complexity is $\mathbf{O}(n^3 + N \cdot n^2)$.

Computation of Determinants

If a LR-decomposition of the matrix $A \in \mathbb{R}^{n \times n}$ is known, the determinant can be computed from

$$\det(A) = \det(LR) = \det(L)\det(R) = \det(R) = \prod_{i=1}^n R_{ii} .$$

Contents

- Problem Formulation
- Conditioning of Linear Equation Systems
- Gauss Elimination
- **Linear Equations with Band-Structured Matrix**
- Linear Equations with Positive Definite Matrices
- QR-Decomposition

Band-Structured Matrices

A band-structured matrix is a matrix whose components satisfy

$$a_{j,k} = 0 \quad \text{for} \quad k < j - m_l \quad \text{or} \quad k > j + m_r .$$

Here m_l and m_r are called the widths of the band-matrix A .

Examples:

- For $m_l = 0$ and $m_r \leq n - 1$: upper triangular matrix.
- For $m_r = 0$ and $m_l \leq n - 1$: lower triangular matrix.
- For $m_l = 1$ and $m_r = 1$: tridiagonal matrix.

Band-Structured Matrices

A band-structured matrix is a matrix whose components satisfy

$$a_{j,k} = 0 \quad \text{for} \quad k < j - m_l \quad \text{or} \quad k > j + m_r .$$

Here m_l and m_r are called the widths of the band-matrix A .

Examples:

- For $m_l = 0$ and $m_r \leq n - 1$: upper triangular matrix.
- For $m_r = 0$ and $m_l \leq n - 1$: lower triangular matrix.
- For $m_l = 1$ and $m_r = 1$: tridiagonal matrix.

Gauss Elimination for Band-Structured Matrices

If we do not have to permute rows, equation systems of the form

$$Ax = b$$

can be solved with Gauss-elimination, too, while exploiting the band-structure. The complexity is in this case

$$\mathbf{O}(nm_l m_r) + \mathbf{O}(n(m_l + m_r)) .$$

Example: Tridiagonal Matrices

For the case that A is a tridiagonal matrix of the form

$$A = \begin{pmatrix} a_1 & b_1 & & & \\ c_2 & \ddots & \ddots & & \\ & \ddots & \ddots & b_{n-1} & \\ & & c_n & a_n & \end{pmatrix}$$

the matrices L and R have the form

$$L = \begin{pmatrix} 1 & & & & \\ \gamma_2 & \ddots & & & \\ & \ddots & 1 & & \\ & & \gamma_n & 1 & \end{pmatrix} \quad \text{and} \quad R = \begin{pmatrix} \alpha_1 & b_1 & & & \\ & \ddots & \ddots & & \\ & & \alpha_{n-1} & b_{n-1} & \\ & & & \alpha_n & \end{pmatrix}.$$

Example: Tridiagonal Matrices

The coefficients α_i and γ_i can be computed by the recursion

$$\gamma_i = \frac{c_i}{\alpha_{i-1}}, \quad \text{and} \quad \alpha_i = a_i - \gamma_i b_i$$

for $i = 2, \dots, n$.

Contents

- Problem Formulation
- Conditioning of Linear Equation Systems
- Gauss Elimination
- Linear Equations with Band-Structured Matrix
- **Linear Equations with Positive Definite Matrices**
- QR-Decomposition

Symmetric Positive Definite Matrices

A matrix $A \in \mathbb{R}^{n \times n}$ is called symmetric positive definite if we have $A = A^T$ and

$$\forall v \in \mathbb{R}^n \setminus \{0\}, \quad v^T A v > 0 .$$

Equation systems of the form $Ax = b$ with A being symmetric positive semi-definite are solved by representing A in the form

$$A = LDL^T ,$$

where $L \in \mathbb{R}^{n \times n}$ is a lower diagonal matrix with ones on the diagonal and $D \in \mathbb{R}^{n \times n}$ a diagonal matrix.

Symmetric Positive Definite Matrices

A matrix $A \in \mathbb{R}^{n \times n}$ is called symmetric positive definite if we have $A = A^T$ and

$$\forall v \in \mathbb{R}^n \setminus \{0\}, \quad v^T A v > 0 .$$

Equation systems of the form $Ax = b$ with A being symmetric positive semi-definite are solved by representing A in the form

$$A = LDL^T ,$$

where $L \in \mathbb{R}^{n \times n}$ is a lower diagonal matrix with ones on the diagonal and $D \in \mathbb{R}^{n \times n}$ a diagonal matrix.

Cholesky Algorithm (LDL^T -Variant)

Let us work out the matrix product

$$\begin{aligned}
 LDL^T &= \begin{pmatrix} 1 & & & \\ L_{2,1} & 1 & & \\ L_{3,1} & L_{3,2} & 1 & \\ \vdots & \vdots & & \ddots \end{pmatrix} \begin{pmatrix} D_1 & & & \\ & D_2 & & \\ & & D_3 & \\ & & & \ddots \end{pmatrix} \begin{pmatrix} 1 & & & \\ L_{2,1} & 1 & & \\ L_{3,1} & L_{3,2} & 1 & \\ \vdots & \vdots & & \ddots \end{pmatrix}^T \\
 &= \begin{pmatrix} D_1 & & & \\ L_{2,1}D_1 & L_{2,1}^2D_1 + D_2 & & \\ L_{3,1}D_1 & L_{3,1}L_{2,1}D_1 + L_{3,2}D_2 & L_{3,1}^2D_1 + L_{3,2}^2D_2 + D_3 & \\ \vdots & \vdots & & \ddots \end{pmatrix} \begin{matrix} \\ \\ \text{symmetric} \\ \end{matrix}
 \end{aligned}$$

Cholesky Algorithm (LDL^T -Variant)

We want to determine L and D such that

$$A = \begin{pmatrix} D_1 & & & & \\ L_{2,1}D_1 & L_{2,1}^2D_1 + D_2 & & & \\ L_{3,1}D_1 & L_{3,1}L_{2,1}D_1 + L_{3,2}D_2 & L_{3,1}^2D_1 + L_{3,2}^2D_2 + D_3 & & \\ \vdots & \vdots & & \ddots & \\ & & & & \ddots \end{pmatrix} \quad \text{symmetric}$$

- We first set $D_1 = A_{1,1} > 0$ (since A is positive definite).
- We can find $L_{2,1} = A_{2,1}/D_1$.
- The diagonal entry $D_2 = A_{2,2} - L_{2,1}^2D_1 > 0$ must be positive.
- General recursion (for $i > j$):

$$D_{j,j} = A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2 D_k \quad \text{and} \quad L_{i,j} = \frac{1}{D_j} \left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k} D_k \right).$$

Cholesky Algorithm (LDL^T -Variant)

We want to determine L and D such that

$$A = \begin{pmatrix} D_1 & & & & \\ L_{2,1}D_1 & L_{2,1}^2D_1 + D_2 & & & \\ L_{3,1}D_1 & L_{3,1}L_{2,1}D_1 + L_{3,2}D_2 & L_{3,1}^2D_1 + L_{3,2}^2D_2 + D_3 & & \\ \vdots & \vdots & & \ddots & \\ & & & & \ddots \end{pmatrix} \quad \text{symmetric}$$

- We first set $D_1 = A_{1,1} > 0$ (since A is positive definite).
- We can find $L_{2,1} = A_{2,1}/D_1$.
- The diagonal entry $D_2 = A_{2,2} - L_{2,1}^2D_1 > 0$ must be positive.
- General recursion (for $i > j$):

$$D_{j,j} = A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2 D_k \quad \text{and} \quad L_{i,j} = \frac{1}{D_j} \left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k} D_k \right).$$

Cholesky Algorithm (LDL^T -Variant)

We want to determine L and D such that

$$A = \begin{pmatrix} D_1 & & & & \\ L_{2,1}D_1 & L_{2,1}^2D_1 + D_2 & & & \\ L_{3,1}D_1 & L_{3,1}L_{2,1}D_1 + L_{3,2}D_2 & L_{3,1}^2D_1 + L_{3,2}^2D_2 + D_3 & & \\ \vdots & \vdots & & \ddots & \\ & & & & \ddots \end{pmatrix} \quad \text{symmetric}$$

- We first set $D_1 = A_{1,1} > 0$ (since A is positive definite).
- We can find $L_{2,1} = A_{2,1}/D_1$.
- The diagonal entry $D_2 = A_{2,2} - L_{2,1}^2D_1 > 0$ must be positive.
- General recursion (for $i > j$):

$$D_{j,j} = A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2 D_k \quad \text{and} \quad L_{i,j} = \frac{1}{D_j} \left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k} D_k \right).$$

Cholesky Algorithm (LDL^T -Variant)

We want to determine L and D such that

$$A = \begin{pmatrix} D_1 & & & & \\ L_{2,1}D_1 & L_{2,1}^2D_1 + D_2 & & & \\ L_{3,1}D_2 & L_{3,1}L_{2,1}D_1 + L_{3,2}D_2 & L_{3,1}^2D_1 + L_{3,2}^2D_2 + D_3 & & \\ \vdots & \vdots & & \ddots & \\ & & & & \ddots \end{pmatrix} \quad \text{symmetric}$$

- We first set $D_1 = A_{1,1} > 0$ (since A is positive definite).
- We can find $L_{2,1} = A_{2,1}/D_1$.
- The diagonal entry $D_2 = A_{2,2} - L_{2,1}^2D_1 > 0$ must be positive.
- General recursion (for $i > j$):

$$D_{j,j} = A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2 D_k \quad \text{and} \quad L_{i,j} = \frac{1}{D_j} \left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k} D_k \right).$$

Cholesky Algorithm (LDL^T -Variant)

We want to determine L and D such that

$$A = \begin{pmatrix} D_1 & & & & \\ L_{2,1}D_1 & L_{2,1}^2D_1 + D_2 & & & \\ L_{3,1}D_1 & L_{3,1}L_{2,1}D_1 + L_{3,2}D_2 & L_{3,1}^2D_1 + L_{3,2}^2D_2 + D_3 & & \\ \vdots & \vdots & & \ddots & \\ & & & & \ddots \end{pmatrix} \quad \text{symmetric}$$

- We first set $D_1 = A_{1,1} > 0$ (since A is positive definite).
- We can find $L_{2,1} = A_{2,1}/D_1$.
- The diagonal entry $D_2 = A_{2,2} - L_{2,1}^2D_1 > 0$ must be positive.
- General recursion (for $i > j$):

$$D_{j,j} = A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2 D_k \quad \text{and} \quad L_{i,j} = \frac{1}{D_j} \left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k} D_k \right) .$$

Cholesky Decomposition

- Cholesky's algorithm works reliably for positive definite matrices, $D_{ii} > 0$. We could also set

$$A = \tilde{L}^T \tilde{L} \quad \text{with} \quad \tilde{L} = D^{\frac{1}{2}} L .$$

This is the “original” Cholesky decomposition ($\tilde{L}$ can be interpreted as a “square-root” of A)

- Variants of the above $L^T D L$ algorithm are applicable for symmetric indefinite matrices, but require permutation of rows/columns
- For the symmetric non-positive-definite matrix

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

the $L^T D L$ algorithm fails (but we could permute rows).

Contents

- Problem Formulation
- Conditioning of Linear Equation Systems
- Gauss Elimination
- Linear Equations with Band-Structured Matrix
- Linear Equations with Positive Definite Matrices
- **QR-Decomposition**

Orthonormal Basis

- Let $A = (a_1, a_2, a_3, \dots, a_m) \in \mathbb{R}^{n \times m}$ be a given matrix
- The vectors $a_1, a_2, \dots, a_m \in \mathbb{R}^n$ are the columns of A
- The vectors $q_1, q_2, \dots, q_r \in \mathbb{R}^n$ with $r \leq n$ are called an orthonormal basis if

$$\text{span}(q_1, q_2, \dots, q_r) = \text{span}(a_1, a_2, \dots, a_m) = \text{Im}(A)$$

and

$$\forall i, j \in \{1, 2, \dots, r\}, \quad q_i^T q_j = \delta_{i,j} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

- Notice that we must have $r = \dim(\text{Im}(A))$

QR Decomposition

- Regard the orthormal basis vectors as columns of a matrix Q ,

$$Q = (q_1, q_2, \dots, q_r)$$

- The matrix Q satisfies $Q^T Q = I$
- And we have $\text{Im}(A) = \text{Im}(Q)$
- On the next slides we will prove that there exist an upper triangular matrix R such that

$$A = QR .$$

This is called a QR decomposition of A .

Gram-Schmidt Algorithm

Input: m vectors $a_1, \dots, a_m \in \mathbb{R}^n$.

Gram-Schmidt Algorithm:

For $i = 1, \dots, m$:

- Orthogonalization. $\bar{q}_i = a_i - (q_1^T a_i)q_1 - \dots - (q_{i-1}^T a_i)q_{i-1}$.
- Test for linear dependence. If $\bar{q}_i = 0$, set $q_i = 0$ and skip next step.
- Normalization. $q_i = \frac{\bar{q}_i}{\|\bar{q}_i\|_H}$.

Optional: delete all 0s from the sequence $q_1, q_2, \dots, q_m$. This saves memory but we should keep an index list (or use a sparse matrix format).

Gram-Schmidt Algorithm

Input: m vectors $a_1, \dots, a_m \in \mathbb{R}^n$.

Gram-Schmidt Algorithm:

For $i = 1, \dots, m$:

- Orthogonalization. $\bar{q}_i = a_i - (q_1^T a_i)q_1 - \dots - (q_{i-1}^T a_i)q_{i-1}$.
- Test for linear dependence. If $\bar{q}_i = 0$, set $q_i = 0$ and skip next step.
- Normalization. $q_i = \frac{\bar{q}_i}{\|\bar{q}_i\|_H}$.

Optional: delete all 0s from the sequence $q_1, q_2, \dots, q_m$. This saves memory but we should keep an index list (or use a sparse matrix format).

Properties of Gram-Schmidt Algorithm

Gram-Schmidt constructs the $q_1, q_2, \dots, q_m$ (with 0s) such that

$$\begin{aligned}a_1 &= (a_1^\top q_1)q_1 \\a_2 &= (a_2^\top q_1)q_1 + (a_2^\top q_2)q_2 \\&\vdots \\a_m &= (a_m^\top q_1)q_1 + (a_m^\top q_2)q_2 + \dots + (a_m^\top q_m)q_m\end{aligned}\tag{1}$$

Moreover, we have $q_i^\top q_j = \delta_{i,j}$ (Exercise: proof by induction!)

QR Decomposition

$$\text{Write } a_1 = r_{1,1}q_1$$

$$a_2 = r_{1,2}q_1 + r_{2,2}q_2$$

$$\vdots$$

$$a_m = r_{1,m}q_1 + r_{2,m}q_2 + \dots + r_{m,m}q_m$$

with $r_{i,j} = a_j^\top q_i$, which yields $A = QR$ with

$$\underbrace{(a_1, a_2, \dots, a_m)}_{=A} = \underbrace{(q_1, q_2, \dots, q_m)}_{=Q} \underbrace{\begin{pmatrix} r_{1,1} & r_{1,2} & \dots & r_{1,m} \\ & r_{2,2} & \dots & r_{2,m} \\ & & \ddots & \vdots \\ & & & r_{m,m} \end{pmatrix}}_{=R}$$

Complete QR decomposition

Problem:

- we might have $\dim(\text{Im}(A)) = r < n$.
- Thus, if $r < n$, we have $Q^T Q = I$ but $Q Q^T \neq I$.

Remedy:

- apply Gram-Schmidt algorithm to the augmented matrix $\tilde{A} = [A, I]$
- delete all “zero-columns” from Q (and corresponding rows of R)
- this yields Q and R such that $QR = \tilde{A}$ and

$$Q \in \mathbb{R}^{n \times n}, \quad R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix} \quad \text{with} \quad \begin{cases} R_{11} \in \mathbb{R}^{r \times m} \\ R_{22} \in \mathbb{R}^{n-r \times n}, \end{cases}$$

where Q is orthogonal, R_{11} and R_{22} upper triangular (with full rank).

QR Decomposition

- In summary, Gram-Schmidt algorithm yields the representation

$$A = \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \begin{bmatrix} R_{11} \\ 0 \end{bmatrix}$$

- The equation system $Ax = b$ admits a solution if and only if

$$\begin{bmatrix} Q_{12}^T & Q_{22}^T \end{bmatrix} b = 0$$

- If the above condition holds, all solutions of $Ax = b$ satisfy

$$R_{11}x = r_1 \quad \text{with} \quad r_1 = \begin{bmatrix} Q_{11}^T & Q_{21}^T \end{bmatrix} b$$

- Solve by backward substitution! (always works, as R_{11} has full rank!)

Householder's QR Algorithm

- Gram-Schmidt algorithm are sometimes affected by cancellation / numerical ill-conditioning: the vectors a_i are reduced in length after successive subtraction of its projections.
- Householder's method is an alternative to Gram-Schmidt algorithms
- Key idea: introduce reflecting matrices of the form

$$H_i = I - 2w_i w_i^T \quad \text{with unit vectors } w_i \in \mathbb{R}^n, \quad \|w_i\|_2 = 1$$

- The matrices H_i are symmetric and orthogonal, since

$$H_i^2 = I - 4w_i w_i^T + 4w_i \underbrace{w_i^T w_i}_{=1} w_i^T = I$$

Householder's QR Algorithm: Derivation

- Let $a \neq 0$ be the first column of a matrix A .
- Our first goal is find a unit vector w such that

$$(I - 2ww^T)a \stackrel{!}{=} \lambda e_1 \quad \text{for a scalar } \lambda \in \mathbb{R}$$

- We claim that this is true for

$$w = \frac{a \pm \|a\|e_1}{\|a \pm \|a\|e_1\|}$$

- Proof: either use your geometric intuition or directly show that

$$\|a \pm \|a\|e_1\|_2^2 = 2(a \pm \|a\|e_1)^T a \implies (I - 2ww^T)a = \pm(-1)\|a\|e_1$$

Householder's QR Algorithm: Summary

- We can find $H_1 = I - 2w_1w_1^T$ such that

$$H_1A = \begin{pmatrix} \star & \star & \dots & \star \\ 0 & \star & \dots & \star \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \star & \dots & \star \end{pmatrix}$$

- We repeat the same procedure for all sub-blocks until

$$H_{n-1}H_{n-2}\dots H_1A = R$$

is upper triangular. Finally, $A = QR$ with $Q = H_1H_2\dots H_{n-1}$.

Chapter 3

Iterative Methods

Iterative Methods

- Problem Formulation
- Gradient Methods
- Conjugent Gradient Methods

Contents

- Problem Formulation
- Gradient Methods
- Conjugent Gradient Methods

Problem Formulation

We are interested in solving “large” linear equation systems of the form

$$Ax = b ,$$

where $A \in \mathbb{R}^{n \times n}$ is a symmetric and positive definite matrix.

Iterative versus Direct Methods

Examples for direct LA methods are

- Gauss-elimination; LR decomposition)
- Gram-Schmidt methods; QR decomposition
- Cholesky factorization; tailored for symmetric matrices)

These and similar methods

1. have complexity $O(n^3)$ (if sparsity is not exploited)
2. find x up to small rounding errors, if A is well-conditioned.

Iterative versus Direct Methods

Examples for direct LA methods are

- Gauss-elimination; LR decomposition)
- Gram-Schmidt methods; QR decomposition
- Cholesky factorization; tailored for symmetric matrices)

These and similar methods

1. have complexity $O(n^3)$ (if sparsity is not exploited)
2. find x up to small rounding errors, if A is well-conditioned.

Iterative versus Direct Methods

In contrast to direct elimination/decomposition methods, we concentrate in this lecture on iterative algorithms that

- converge to a solution x of the equation $Ax = b$,
- improve an iterate for $x_k \approx x$ at every step,
- are stopped whenever a “sufficiently accurate solution” is found or if we run out of time.

Contents

- Problem Formulation
- Gradient Methods
- Conjugent Gradient Methods

An optimization perspective

Notice that solving the equation $Ax = b$, $A \succ 0$, is equivalent to solving the quadratic optimization problem

$$\min_x F(x) \quad \text{with} \quad F(x) = \frac{1}{2}x^T Ax - x^T b .$$

Proof: The function F is strictly convex (since $F''(x) = A \succ 0$) and the gradient $\nabla F(x) = Ax - b$ is zero if and only if x satisfies $Ax = b$.

An alternative approach could start by solving $\min_x \|Ax - b\|_2^2$, but then we would square the condition number.

An optimization perspective

Notice that solving the equation $Ax = b$, $A \succ 0$, is equivalent to solving the quadratic optimization problem

$$\min_x F(x) \quad \text{with} \quad F(x) = \frac{1}{2}x^T Ax - x^T b .$$

Proof: The function F is strictly convex (since $F''(x) = A \succ 0$) and the gradient $\nabla F(x) = Ax - b$ is zero if and only if x satisfies $Ax = b$.

An alternative approach could start by solving $\min_x \|Ax - b\|_2^2$, but then we would square the condition number.

An optimization perspective

Notice that solving the equation $Ax = b$, $A \succ 0$, is equivalent to solving the quadratic optimization problem

$$\min_x F(x) \quad \text{with} \quad F(x) = \frac{1}{2}x^T Ax - x^T b .$$

Proof: The function F is strictly convex (since $F''(x) = A \succ 0$) and the gradient $\nabla F(x) = Ax - b$ is zero if and only if x satisfies $Ax = b$.

An alternative approach could start by solving $\min_x \|Ax - b\|_2^2$, but then we would square the condition number.

Gradient Methods

Gradient methods compute iterate by search for the “steepest descent” of F :

$$x_{k+1} = x_k - \alpha_k \nabla F(x_k) = x_k - \alpha_k (Ax_k - b) .$$

The line search parameter α_k can be found “exact line search”, that is, by solving

$$\min_{\alpha_k} F(x_k - \alpha_k (Ax_k - b)) .$$

Gradient Methods

Gradient methods compute iterate by search for the “steepest descent” of F :

$$x_{k+1} = x_k - \alpha_k \nabla F(x_k) = x_k - \alpha_k (Ax_k - b) .$$

The line search parameter α_k can be found “exact line search”, that is, by solving

$$\min_{\alpha_k} F(x_k - \alpha_k (Ax_k - b)) .$$

Gradient Methods

In order to work out explicitly the solution of the line search problem

$$\min_{\alpha_k} F(x_k - \alpha_k(Ax_k - b)) ,$$

we denote with $d_k = Ax_k - b$ the search direction and eliminate α_k from the optimality conditions

$$0 = \nabla F(x_k - \alpha_k d_k)^T d_k = d_k^T d_k - \alpha_k d_k^T A d_k$$

which yields

$$\alpha_k = \frac{d_k^T d_k}{d_k^T A d_k} .$$

Summary: Gradient Method

1. **Input:** An initial guess x_0 and tolerance $\epsilon > 0$.
2. **Repeat:**
 - 2.1 compute the step direction $d_k = Ax_k - b$,
 - 2.2 if $\|d\| < \epsilon$, stop.
 - 2.3 compute the line search parameter $\alpha_k = \frac{d_k^T d_k}{d_k^T A d_k}$,
 - 2.4 set $x_{k+1} = x_k - \alpha_k d_k$ and increase the counter $k \leftarrow k + 1$.
3. **Output:** A numerical approximation $x_k \approx x$ of the solution vector.

Convergence Analysis of Gradient Methods

Most of the convergence proofs for gradient methods first show that we get in every step a sufficient descent of the “Lyapunov function”

$$L(y) = (y - x)^T A(y - x) .$$

One way to show this is by using the equation

$$\begin{aligned} L(x_{k+1}) &= (x_k - x - \alpha_k d_k)^T A (x_k - x - \alpha_k d_k)^T \\ &= L(x_k) - 2\alpha_k (x_k - x)^T A d_k + \alpha_k^2 d_k^T A d_k . \end{aligned}$$

Since $d_k = Ax_k - b = A(x_k - x)$ this can be simplified further to

$$L(x_{k+1}) = L(x_k) - 2\alpha_k d_k^T d_k + \alpha_k^2 d_k^T A d_k = L(x_k) - \underbrace{\frac{\|d_k\|_2^4}{d_k^T A d_k}}_{>0} ,$$

which proves that we get a strict descent.

Convergence Analysis of Gradient Methods

Most of the convergence proofs for gradient methods first show that we get in every step a sufficient descent of the “Lyapunov function”

$$L(y) = (y - x)^T A(y - x) .$$

One way to show this is by using the equation

$$\begin{aligned} L(x_{k+1}) &= (x_k - x - \alpha_k d_k)^T A (x_k - x - \alpha_k d_k)^T \\ &= L(x_k) - 2\alpha_k (x_k - x)^T A d_k + \alpha_k^2 d_k^T A d_k . \end{aligned}$$

Since $d_k = Ax_k - b = A(x_k - x)$ this can be simplified further to

$$L(x_{k+1}) = L(x_k) - 2\alpha_k d_k^T d_k + \alpha_k^2 d_k^T A d_k = L(x_k) - \underbrace{\frac{\|d_k\|_2^4}{d_k^T A d_k}}_{>0} ,$$

which proves that we get a strict descent.

Convergence Analysis of Gradient Methods

Most of the convergence proofs for gradient methods first show that we get in every step a sufficient descent of the “Lyapunov function”

$$L(y) = (y - x)^T A(y - x) .$$

One way to show this is by using the equation

$$\begin{aligned} L(x_{k+1}) &= (x_k - x - \alpha_k d_k)^T A (x_k - x - \alpha_k d_k)^T \\ &= L(x_k) - 2\alpha_k (x_k - x)^T A d_k + \alpha_k^2 d_k^T A d_k . \end{aligned}$$

Since $d_k = Ax_k - b = A(x_k - x)$ this can be simplified further to

$$L(x_{k+1}) = L(x_k) - 2\alpha_k d_k^T d_k + \alpha_k^2 d_k^T A d_k = L(x_k) - \underbrace{\frac{\|d_k\|_2^4}{d_k^T A d_k}}_{>0} ,$$

which proves that we get a strict descent.

Convergence Analysis of Gradient Methods

In order to finally prove convergence, we have to analyze the equation

$$L(x_{k+1}) = L(x_k) - \frac{\|d_k\|_2^4}{d_k^T A d_k},$$

a bit further. For this aim, we estimate the term

$$\frac{\|d_k\|_2^4}{d_k^T A d_k L(x_k)} = \frac{\|d_k\|_2^4}{d_k^T A d_k d_k^T A^{-1} d_k} \geq \frac{\lambda_{\min}(A)}{\lambda_{\max}(A)} = \frac{1}{\text{cond}_2(A)}.$$

This proves that the gradient method converges with linear rate

$$L(x_{k+1}) \leq \left(1 - \frac{\lambda_{\min}(A)}{\lambda_{\max}(A)}\right) L(x_k)$$

Convergence Analysis of Gradient Methods

In order to finally prove convergence, we have to analyze the equation

$$L(x_{k+1}) = L(x_k) - \frac{\|d_k\|_2^4}{d_k^T A d_k},$$

a bit further. For this aim, we estimate the term

$$\frac{\|d_k\|_2^4}{d_k^T A d_k L(x_k)} = \frac{\|d_k\|_2^4}{d_k^T A d_k d_k^T A^{-1} d_k} \geq \frac{\lambda_{\min}(A)}{\lambda_{\max}(A)} = \frac{1}{\text{cond}_2(A)}.$$

This proves that the gradient method converges with linear rate

$$L(x_{k+1}) \leq \left(1 - \frac{\lambda_{\min}(A)}{\lambda_{\max}(A)}\right) L(x_k)$$

Convergence Analysis of Gradient Methods

The convergence rate estimate $1 - \frac{\lambda_{\min}(A)}{\lambda_{\max}(A)}$ indicates that

- Gradient methods work very well, if $\text{cond}_2(A)$ is close to 1.
- The other way round, if A is ill-conditioned the gradient method converges very slowly.

These theoretical prediction are confirmed in numerical experiments.

Convergence of Gradient Methods in Practice

In practice, if we plot the iterates of a gradient method, we typically observe a “zig-zag” behavior. This is due to the fact that subsequent search directions of the gradient method are orthogonal to each other

$$d_{k+1}^T d_k = (d_k - \alpha_k A d_k)^T d_k = d_k^T d_k - \alpha_k d_k^T A d_k = 0 .$$

Remark: the convergence rate estimate of the gradient methods can be improved by using “Kantorovich’s inequality” which yields

$$\sqrt{L(x_{k+1})} \leq \frac{\lambda_{\max}(A) - \lambda_{\min}(A)}{\lambda_{\max}(A) + \lambda_{\min}(A)} \sqrt{L(x_k)} .$$

Convergence of Gradient Methods in Practice

In practice, if we plot the iterates of a gradient method, we typically observe a “zig-zag” behavior. This is due to the fact that subsequent search directions of the gradient method are orthogonal to each other

$$d_{k+1}^T d_k = (d_k - \alpha_k A d_k)^T d_k = d_k^T d_k - \alpha_k d_k^T A d_k = 0 .$$

Remark: the convergence rate estimate of the gradient methods can be improved by using “Kantorovich’s inequality” which yields

$$\sqrt{L(x_{k+1})} \leq \frac{\lambda_{\max}(A) - \lambda_{\min}(A)}{\lambda_{\max}(A) + \lambda_{\min}(A)} \sqrt{L(x_k)} .$$

Contents

- Problem Formulation
- Gradient Methods
- Conjugent Gradient Methods

Conjugent Vectors

Two vectors u and v are called conjugent (or “A-orthogonal”) with respect to a matrix A , if

$$u^T A v = 0 .$$

- for gradient methods only successive search directions are orthogonal
- main idea of conjugent gradient methods: maintain A-orthogonality of all search directions.

Conjugent Vectors

Two vectors u and v are called conjugent (or “A-orthogonal”) with respect to a matrix A , if

$$u^T A v = 0 .$$

- for gradient methods only successive search directions are orthogonal
- main idea of conjugent gradient methods: maintain A-orthogonality of all search directions.

Krylov subspaces

The affine vector spaces

$$K_i(A, d_0) = \text{span} (d_0, Ad_0, A^2d_0, \dots, A^i d_0)$$

are called Krylov subspaces.

Construction of Conjugent Gradient Methods

The main idea is to construct iterates of the form

$$x_i = x_0 + \sum_{j=0}^{i-1} \beta_j d_j$$

such that the coefficients β_j are minimizers of

$$\min_{\beta} F \left(x_0 + \sum_{j=0}^{j-1} \beta_j d_j \right) \quad \text{with} \quad F(x) = \frac{1}{2} x^T A x - b^T x ,$$

which gives the optimality conditions $(Ax_i - b)^T d_j = 0$ for $j = 1, \dots, i - 1$.

Construction of Conjugent Gradient Methods

Let the previous search directions $d_0 = Ax_0 - b, d_1, \dots, d_{i-1}$ be an A -orthogonal basis of the Krylov space $K_i(A, d_0)$. We may assume $Ax_i - b \notin K_i(A, d_0)$ as we would have $x_i = x$ otherwise. This motivates to construct the next search direction $d_i \in K_{i+1}(A, d_0)$ from

$$d_i = -(Ax_i - b) + \beta_{i-1}d_{i-1} .$$

This direction satisfies the orthogonality condition

$$d_i^T Ad_j = -(Ax_i - b)^T Ad_j + \beta_{i-1}d_{i-1}^T Ad_j = 0$$

for $j = 1, \dots, i-2$ by construction.

Construction of Conjugent Gradient Methods

The parameter β_{i-1} is then constructed in such a way that we have

$$\begin{aligned} 0 &= d_i^T A d_{i-1} = -(Ax_i - b)^T A d_{i-1} + \beta_{i-1} d_{i-1}^T A d_{i-1} \\ \implies \beta_{i-1} &= \frac{(Ax_i - b)^T A d_{i-1}}{d_{i-1}^T A d_{i-1}} . \end{aligned}$$

Thus yields the recursion law for the conjugent gradient method

$$g_{k+1} = g_k + \alpha_k A g_k$$

$$x_{k+1} = x_k + \alpha_k d_k$$

$$d_{k+1} = -g_k + \beta_k d_k$$

with $\alpha_k = \frac{g_k^T d_k}{d_k^T A d_k} = \frac{\|g_k\|_2^2}{d_k^T A d_k}$ and $\beta_k = \frac{\|g_{k+1}\|_2^2}{\|g_k\|_2^2}$. The method is started with $d_0 = -g_0 = b - Ax_0$.

Construction of Conjugent Gradient Methods

The parameter β_{i-1} is then constructed in such a way that we have

$$\begin{aligned} 0 &= d_i^T A d_{i-1} = -(Ax_i - b)^T A d_{i-1} + \beta_{i-1} d_{i-1}^T A d_{i-1} \\ \implies \quad \beta_{i-1} &= \frac{(Ax_i - b)^T A d_{i-1}}{d_{i-1}^T A d_{i-1}} . \end{aligned}$$

Thus yields the recursion law for the conjugent gradient method

$$g_{k+1} = g_k + \alpha_k A g_k$$

$$x_{k+1} = x_k + \alpha_k d_k$$

$$d_{k+1} = -g_k + \beta_k d_k$$

with $\alpha_k = \frac{g_k^T d_k}{d_k^T A d_k} = \frac{\|g_k\|_2^2}{d_k^T A d_k}$ and $\beta_k = \frac{\|g_{k+1}\|_2^2}{\|g_k\|_2^2}$. The method is started with $d_0 = -g_0 = b - Ax_0$.

Convergence of Conjugent Gradient Methods

Since we are constructing an A -orthogonal basis, the conjugent gradient method terminates after at most n steps.

- If we run the conjugent gradient method for n steps, it is a “direct method”.
- In practice, the conjugent gradient method is terminated whenever sufficient accuracy is achieved, e.g., if $\|g_k\| \leq \epsilon$.

Convergence of Conjugent Gradient Methods

Since we are constructing an A -orthogonal basis, the conjugent gradient method terminates after at most n steps.

- If we run the conjugent gradient method for n steps, it is a “direct method”.
- In practice, the conjugent gradient method is terminated whenever sufficient accuracy is achieved, e.g., if $\|g_k\| \leq \epsilon$.

Chapter 4

Polynomial Interpolation

Polynomial Interpolation

- Problem Formulation
- Divided Differences
- Interpolating Functions
- Hermite Interpolation

Contents

- Problem Formulation
- Divided Differences
- Interpolating Functions
- Hermite Interpolation

Polynomial Interpolation

We have $n + 1$ data point $(x_0, y_0), \dots, (x_n, y_n)$. We are interested in finding a polynomial

$$p(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$$

such that $p(x_i) = y_i$ for all $i \in \{0, \dots, n\}$.

Application Examples:

- We have a (smooth) function $f : \mathbb{R} \rightarrow \mathbb{R}$. Evaluating f at one point takes, say 1h. We need to evaluate f at 10^6 points $x \in [0, 1]$ within 6h. What can we do?
- We measure very accurately the lift force of a wing for 11 different angles of attack in $[0^\circ, 10^\circ]$. We want to predict the lift force of the wing at intermediate angles (but have no physical model at hand).

Polynomial Interpolation

We have $n + 1$ data point $(x_0, y_0), \dots, (x_n, y_n)$. We are interested in finding a polynomial

$$p(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$$

such that $p(x_i) = y_i$ for all $i \in \{0, \dots, n\}$.

Application Examples:

- We have a (smooth) function $f : \mathbb{R} \rightarrow \mathbb{R}$. Evaluating f at one point takes, say 1h. We need to evaluate f at 10^6 points $x \in [0, 1]$ within 6h. What can we do?
- We measure very accurately the lift force of a wing for 11 different angles of attack in $[0^\circ, 10^\circ]$. We want to predict the lift force of the wing at intermediate angles (but have no physical model at hand).

Existence and Uniqueness of Solutions

Theorem If none of the points $x_0, \dots, x_n$ are equal, there exists a unique sequence of coefficients $a_0, \dots, a_n$ such that $p(x_i) = y_i$ for all $i \in \{0, \dots, n\}$, where

$$p(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n .$$

Proof: The proof of this theorem proceeds in two parts:

- *Existence:* construct a polynomial satisfying all requirements,
- *Uniqueness:* prove that if we have two interpolating polynomials, then they are equal.

Existence and Uniqueness of Solutions

Theorem If none of the points $x_0, \dots, x_n$ are equal, there exists a unique sequence of coefficients $a_0, \dots, a_n$ such that $p(x_i) = y_i$ for all $i \in \{0, \dots, n\}$, where

$$p(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n .$$

Proof: The proof of this theorem proceeds in two parts:

- *Existence:* construct a polynomial satisfying all requirements,
- *Uniqueness:* prove that if we have two interpolating polynomials, then they are equal.

Existence and Uniqueness of Solutions

Theorem If none of the points $x_0, \dots, x_n$ are equal, there exists a unique sequence of coefficients $a_0, \dots, a_n$ such that $p(x_i) = y_i$ for all $i \in \{0, \dots, n\}$, where

$$p(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n .$$

Proof: The proof of this theorem proceeds in two parts:

- *Existence:* construct a polynomial satisfying all requirements,
- *Uniqueness:* prove that if we have two interpolating polynomials, then they are equal.

Lagrange Polynomials

Lagrange's idea is to define auxiliary polynomials of the form

$$L_i(x) := \prod_{j=0, j \neq i}^n \frac{x - x_j}{x_i - x_j}$$

for all $i \in \{0, \dots, n\}$.

Important Property:

$$L_i(x_k) = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{otherwise} \end{cases} = \delta_{i,k}$$

Thus, $p(x) = \sum_{k=0}^n y_k L_k(x)$ satisfies $p(x_i) = y_i$ for all $i \in \{0, \dots, n\}$.

Lagrange Polynomials

Lagrange's idea is to define auxiliary polynomials of the form

$$L_i(x) := \prod_{j=0, j \neq i}^n \frac{x - x_j}{x_i - x_j}$$

for all $i \in \{0, \dots, n\}$.

Important Property:

$$L_i(x_k) = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{otherwise} \end{cases} = \delta_{i,k}$$

Thus, $p(x) = \sum_{k=0}^n y_k L_k(x)$ satisfies $p(x_i) = y_i$ for all $i \in \{0, \dots, n\}$.

Lagrange Polynomials

Lagrange's idea is to define auxiliary polynomials of the form

$$L_i(x) := \prod_{j=0, j \neq i}^n \frac{x - x_j}{x_i - x_j}$$

for all $i \in \{0, \dots, n\}$.

Important Property:

$$L_i(x_k) = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{otherwise} \end{cases} = \delta_{i,k}$$

Thus, $p(x) = \sum_{k=0}^n y_k L_k(x)$ satisfies $p(x_i) = y_i$ for all $i \in \{0, \dots, n\}$.

Example: Linear Interpolation

For $n = 1$ the problem reduces to finding a line (= a polynomial with degree 1) passing through two given points

$$(x_0, y_0) \quad \text{and} \quad (x_1, y_1) .$$

The corresponding Lagrange polynomials are

$$L_0(x) = \frac{x - x_1}{x_0 - x_1} \quad \text{and} \quad L_1(x) = \frac{x - x_0}{x_1 - x_0}$$

Thus, the affine given function passing through the points is

$$\begin{aligned} p(x) &= y_0 L_0(x) + y_1 L_1(x) = y_0 \frac{x - x_1}{x_0 - x_1} + y_1 \frac{x - x_0}{x_1 - x_0} \\ &= \underbrace{\frac{y_0 - y_1}{x_0 - x_1}}_{a_1} x + \underbrace{\frac{y_1 x_0 - y_0 x_1}{x_0 - x_1}}_{a_0} = a_0 + a_1 x \end{aligned}$$

The function p satisfies $p(x_0) = y_0$ and $p(x_1) = y_1$.

Example: Linear Interpolation

For $n = 1$ the problem reduces to finding a line (= a polynomial with degree 1) passing through two given points

$$(x_0, y_0) \quad \text{and} \quad (x_1, y_1) .$$

The corresponding Lagrange polynomials are

$$L_0(x) = \frac{x - x_1}{x_0 - x_1} \quad \text{and} \quad L_1(x) = \frac{x - x_0}{x_1 - x_0}$$

Thus, the affine given function passing through the points is

$$\begin{aligned} p(x) &= y_0 L_0(x) + y_1 L_1(x) = y_0 \frac{x - x_1}{x_0 - x_1} + y_1 \frac{x - x_0}{x_1 - x_0} \\ &= \underbrace{\frac{y_0 - y_1}{x_0 - x_1}}_{a_1} x + \underbrace{\frac{y_1 x_0 - y_0 x_1}{x_0 - x_1}}_{a_0} = a_0 + a_1 x \end{aligned}$$

The function p satisfies $p(x_0) = y_0$ and $p(x_1) = y_1$.

Example: Linear Interpolation

For $n = 1$ the problem reduces to finding a line (= a polynomial with degree 1) passing through two given points

$$(x_0, y_0) \quad \text{and} \quad (x_1, y_1) .$$

The corresponding Lagrange polynomials are

$$L_0(x) = \frac{x - x_1}{x_0 - x_1} \quad \text{and} \quad L_1(x) = \frac{x - x_0}{x_1 - x_0}$$

Thus, the affine given function passing through the points is

$$\begin{aligned} p(x) &= y_0 L_0(x) + y_1 L_1(x) = y_0 \frac{x - x_1}{x_0 - x_1} + y_1 \frac{x - x_0}{x_1 - x_0} \\ &= \underbrace{\frac{y_0 - y_1}{x_0 - x_1}}_{a_1} x + \underbrace{\frac{y_1 x_0 - y_0 x_1}{x_0 - x_1}}_{a_0} = a_0 + a_1 x \end{aligned}$$

The function p satisfies $p(x_0) = y_0$ and $p(x_1) = y_1$.

Existence and Uniqueness of Solutions

Theorem If none of the points $x_0, \dots, x_n$ are equal, there exists a unique sequence of coefficients $a_0, \dots, a_n$ such that $p(x_i) = y_i$ for all $i \in \{0, \dots, n\}$, where

$$p(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n .$$

Proof (Part I: *Existence*). The Lagrange polynomials

$$L_i(x) := \prod_{j=0, j \neq i}^n \frac{x - x_j}{x_i - x_j}$$

are well-defined (we never divide by zero), since $x_i \neq x_j$ for all $i \neq j$.

The polynomial $p(x) = \sum_{k=0}^n y_k L_k(x)$ satisfies the requirements; that is, we have found (at least one) solution for p that is guaranteed to exist.

Existence and Uniqueness of Solutions

Theorem If none of the points $x_0, \dots, x_n$ are equal, there exists a unique sequence of coefficients $a_0, \dots, a_n$ such that $p(x_i) = y_i$ for all $i \in \{0, \dots, n\}$, where

$$p(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n .$$

Proof (Part I: *Existence*). The Lagrange polynomials

$$L_i(x) := \prod_{j=0, j \neq i}^n \frac{x - x_j}{x_i - x_j}$$

are well-defined (we never divide by zero), since $x_i \neq x_j$ for all $i \neq j$.

The polynomial $p(x) = \sum_{k=0}^n y_k L_k(x)$ satisfies the requirements; that is, we have found (at least one) solution for p that is guaranteed to exist.

Existence and Uniqueness of Solutions

Theorem If none of the points $x_0, \dots, x_n$ are equal, there exists a unique sequence of coefficients $a_0, \dots, a_n$ such that $p(x_i) = y_i$ for all $i \in \{0, \dots, n\}$, where

$$p(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n .$$

Proof (Part II: *Uniqueness*). Assume that we can find two polynomials p, q with degree $\leq n$ which satisfy $p(x_i) = q(x_i) = y_i$ for $i \in \{0, \dots, n\}$. The function $r(x) = p(x) - q(x)$ satisfies

$$r(x_i) = 0 \quad \text{for all } i \in \{0, \dots, n\}. \quad (n+1 \text{ roots})$$

Thus, $r(x) = 0$, since r is a polynomial of degree $\leq n$, i.e., $p = q$.

Existence and Uniqueness of Solutions

Theorem If none of the points $x_0, \dots, x_n$ are equal, there exists a unique sequence of coefficients $a_0, \dots, a_n$ such that $p(x_i) = y_i$ for all $i \in \{0, \dots, n\}$, where

$$p(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n .$$

Proof (Part II: *Uniqueness*). Assume that we can find two polynomials p, q with degree $\leq n$ which satisfy $p(x_i) = q(x_i) = y_i$ for $i \in \{0, \dots, n\}$. The function $r(x) = p(x) - q(x)$ satisfies

$$r(x_i) = 0 \quad \text{for all } i \in \{0, \dots, n\}. \quad (n+1 \text{ roots})$$

Thus, $r(x) = 0$, since r is a polynomial of degree $\leq n$, i.e., $p = q$.

Contents

- Problem Formulation
- Divided Differences
- Interpolating Functions
- Hermite Interpolation

Disadvantages of Lagrange Polynomials

In practice, Lagrange polynomials are almost never used for interpolation. The two main reasons are:

1. Evaluating the expression $p(x) = \sum_{i=0}^n y_i \prod_{j=0, j \neq i}^n \frac{x-x_j}{x_i-x_j}$ is often not well-conditioned, i.e., we have to expect large numerical errors.
2. Say, we have already a polynomial passing through n data points $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$ but now get a new data point (x_n, y_n) . If we use Lagrange polynomials for computing a polynomial that passes through all data points, we have to compute this new polynomial from scratch.

Disadvantages of Lagrange Polynomials

In practice, Lagrange polynomials are almost never used for interpolation. The two main reasons are:

1. Evaluating the expression $p(x) = \sum_{i=0}^n y_i \prod_{j=0, j \neq i}^n \frac{x-x_j}{x_i-x_j}$ is often not well-conditioned, i.e., we have to expect large numerical errors.
2. Say, we have already a polynomial passing through n data points $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$ but now get a new data point (x_n, y_n) . If we use Lagrange polynomials for computing a polynomial that passes through all data points, we have to compute this new polynomial from scratch.

Disadvantages of Lagrange Polynomials

In practice, Lagrange polynomials are almost never used for interpolation. The two main reasons are:

1. Evaluating the expression $p(x) = \sum_{i=0}^n y_i \prod_{j=0, j \neq i}^n \frac{x-x_j}{x_i-x_j}$ is often not well-conditioned, i.e., we have to expect large numerical errors.
2. Say, we have already a polynomial passing through n data points $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$ but now get a new data point (x_n, y_n) . If we use Lagrange polynomials for computing a polynomial that passes through all data points, we have to compute this new polynomial from scratch.

Newton Polynomials

Newton's basis polynomials are given by

$$N_i(x) := \prod_{j=0}^{i-1} (x - x_j) .$$

The coefficients of the interpolating polynomial $p(x) = \sum_{i=0}^n b_i N_i(x)$ can be found by solving the equation system

$$y_0 = p(x_0) = b_0$$

$$y_1 = p(x_1) = b_0 + b_1(x_1 - x_0)$$

$$\vdots$$

$$y_n = p(x_n) = b_0 + b_1(x_n - x_0) + \dots + b_n(x_n - x_0) \dots (x_n - x_{n-1})$$

recursively with respect to $b_0, b_1, \dots, b_n$.

Newton Polynomials

Newton's basis polynomials are given by

$$N_i(x) := \prod_{j=0}^{i-1} (x - x_j) .$$

The coefficients of the interpolating polynomial $p(x) = \sum_{i=0}^n b_i N_i(x)$ can be found by solving the equation system

$$y_0 = p(x_0) = b_0$$

$$y_1 = p(x_1) = b_0 + b_1(x_1 - x_0)$$

$$\vdots$$

$$y_n = p(x_n) = b_0 + b_1(x_n - x_0) + \dots + b_n(x_n - x_0) \dots (x_n - x_{n-1})$$

recursively with respect to $b_0, b_1, \dots, b_n$.

Neville's Recursion Idea

Neville suggested a numerically stable way to find the coefficients of the interpolating polynomial using Newton's basis.

Key observation: if $(x_0, y_0), \dots, (x_n, y_n)$ are given data points and f and g functions that satisfy

- $f(x_i) = y_i$ for all $i \in \{0, \dots, n-1\}$ and
- $g(x_i) = y_i$ for all $i \in \{1, \dots, n\}$,

then we can construct the new “divided-difference” function

$$h(x) = f(x) + \frac{g(x) - f(x)}{x_n - x_0}(x - x_0) ,$$

which satisfies $h(x_i) = y_i$ for all $i \in \{0, \dots, n\}$.

Neville's Recursion Idea

Neville suggested a numerically stable way to find the coefficients of the interpolating polynomial using Newton's basis.

Key observation: if $(x_0, y_0), \dots, (x_n, y_n)$ are given data points and f and g functions that satisfy

- $f(x_i) = y_i$ for all $i \in \{0, \dots, n-1\}$ and
- $g(x_i) = y_i$ for all $i \in \{1, \dots, n\}$,

then we can construct the new “divided-difference” function

$$h(x) = f(x) + \frac{g(x) - f(x)}{x_n - x_0}(x - x_0) ,$$

which satisfies $h(x_i) = y_i$ for all $i \in \{0, \dots, n\}$.

Neville's Recursion Idea

Neville suggested a numerically stable way to find the coefficients of the interpolating polynomial using Newton's basis.

Key observation: if $(x_0, y_0), \dots, (x_n, y_n)$ are given data points and f and g functions that satisfy

- $f(x_i) = y_i$ for all $i \in \{0, \dots, n-1\}$ and
- $g(x_i) = y_i$ for all $i \in \{1, \dots, n\}$,

then we can construct the new “divided-difference” function

$$h(x) = f(x) + \frac{g(x) - f(x)}{x_n - x_0}(x - x_0) ,$$

which satisfies $h(x_i) = y_i$ for all $i \in \{0, \dots, n\}$.

Neville's Recursion Idea: example for $n = 2$

In order to understand Neville's recursion, we consider the case $n = 2$.

- The constant functions $p_{0,0}(x) = y_0$, $p_{1,1}(x) = y_1$, and $p_{2,2}(x) = y_2$ interpolate the first, second, and third data point, respectively.
- The polynomial $p_{0,1}(x) = p_{0,0}(x) + \frac{p_{1,1}(x) - p_{0,0}(x)}{x_1 - x_0}(x - x_0)$ satisfies $p_{0,1}(x_i) = y_i$ for $i \in \{0, 1\}$.
- The polynomial $p_{1,2}(x) = p_{1,1}(x) + \frac{p_{2,2}(x) - p_{1,1}(x)}{x_2 - x_1}(x - x_1)$ satisfies $p_{1,2}(x_i) = y_i$ for $i \in \{1, 2\}$.
- The polynomial $p(x) = p_{0,1} + \frac{p_{1,2}(x) - p_{0,1}(x)}{x_2 - x_0}(x - x_0)$ satisfies $p(x_i) = y_i$ for $i \in \{0, 1, 2\}$.

Neville's Recursion Idea: example for $n = 2$

In order to understand Neville's recursion, we consider the case $n = 2$.

- The constant functions $p_{0,0}(x) = y_0$, $p_{1,1}(x) = y_1$, and $p_{2,2}(x) = y_2$ interpolate the first, second, and third data point, respectively.
- The polynomial $p_{0,1}(x) = p_{0,0}(x) + \frac{p_{1,1}(x) - p_{0,0}(x)}{x_1 - x_0}(x - x_0)$ satisfies $p_{0,1}(x_i) = y_i$ for $i \in \{0, 1\}$.
- The polynomial $p_{1,2}(x) = p_{1,1}(x) + \frac{p_{2,2}(x) - p_{1,1}(x)}{x_2 - x_1}(x - x_1)$ satisfies $p_{1,2}(x_i) = y_i$ for $i \in \{1, 2\}$.
- The polynomial $p(x) = p_{0,1} + \frac{p_{1,2}(x) - p_{0,1}(x)}{x_2 - x_0}(x - x_0)$ satisfies $p(x_i) = y_i$ for $i \in \{0, 1, 2\}$.

Neville's Recursion Idea: example for $n = 2$

In order to understand Neville's recursion, we consider the case $n = 2$.

- The constant functions $p_{0,0}(x) = y_0$, $p_{1,1}(x) = y_1$, and $p_{2,2}(x) = y_2$ interpolate the first, second, and third data point, respectively.
- The polynomial $p_{0,1}(x) = p_{0,0}(x) + \frac{p_{1,1}(x) - p_{0,0}(x)}{x_1 - x_0}(x - x_0)$ satisfies $p_{0,1}(x_i) = y_i$ for $i \in \{0, 1\}$.
- The polynomial $p_{1,2}(x) = p_{1,1}(x) + \frac{p_{2,2}(x) - p_{1,1}(x)}{x_2 - x_1}(x - x_1)$ satisfies $p_{1,2}(x_i) = y_i$ for $i \in \{1, 2\}$.
- The polynomial $p(x) = p_{0,1} + \frac{p_{1,2}(x) - p_{0,1}(x)}{x_2 - x_0}(x - x_0)$ satisfies $p(x_i) = y_i$ for $i \in \{0, 1, 2\}$.

Neville's Recursion Idea: example for $n = 2$

In order to understand Neville's recursion, we consider the case $n = 2$.

- The constant functions $p_{0,0}(x) = y_0$, $p_{1,1}(x) = y_1$, and $p_{2,2}(x) = y_2$ interpolate the first, second, and third data point, respectively.
- The polynomial $p_{0,1}(x) = p_{0,0}(x) + \frac{p_{1,1}(x) - p_{0,0}(x)}{x_1 - x_0}(x - x_0)$ satisfies $p_{0,1}(x_i) = y_i$ for $i \in \{0, 1\}$.
- The polynomial $p_{1,2}(x) = p_{1,1}(x) + \frac{p_{2,2}(x) - p_{1,1}(x)}{x_2 - x_1}(x - x_1)$ satisfies $p_{1,2}(x_i) = y_i$ for $i \in \{1, 2\}$.
- The polynomial $p(x) = p_{0,1} + \frac{p_{1,2}(x) - p_{0,1}(x)}{x_2 - x_0}(x - x_0)$ satisfies $p(x_i) = y_i$ for $i \in \{0, 1, 2\}$.

Divided Differences

In general, Neville's recursion is initialized with

$$p_{i,i}(x) = y_i \quad \text{f.a.} \quad i \in \{1, \dots, n\}$$

and applies the recursion rule

$$p_{i,i+k}(x) = p_{i,i+k-1}(x) + \frac{p_{i+1,i+k}(x) - p_{i,i+k-1}(x)}{x_{i+k} - x_i}(x - x_i)$$

for $k \in \{1, \dots, n - i\}$ to finally compute $p(x) = p_{0,n}(x)$. This formula can be used directly, if $p(x)$ should be evaluated at a given point x .

Divided Differences

In general, Neville's recursion is initialized with

$$p_{i,i}(x) = y_i \quad \text{f.a.} \quad i \in \{1, \dots, n\}$$

and applies the recursion rule

$$p_{i,i+k}(x) = p_{i,i+k-1}(x) + \frac{p_{i+1,i+k}(x) - p_{i,i+k-1}(x)}{x_{i+k} - x_i}(x - x_i)$$

for $k \in \{1, \dots, n - i\}$ to finally compute $p(x) = p_{0,n}(x)$. This formula can be used directly, if $p(x)$ should be evaluated at a given point x .

Divided Differences

The divided differences are defined by the recursion

$$d_{i,i} = y_i \quad \text{and} \quad d_{i,i+k} = \frac{d_{i+1,i+k} - d_{i,i+k-1}}{x_{i+k} - x_i}$$

for $i \in \{0, \dots, n\}$ and $k \in \{0, \dots, n - i\}$.

Theorem The functions $p_{i,i+k}(x)$ can be written in the form

$$p_{i,i+k}(x) = d_{i,i} + d_{i,i+1}(x - x_i) + \dots + d_{i,i+k}(x - x_i) \dots (x - x_{i+k-1}) .$$

for $i \in \{0, \dots, n\}$ and $k \in \{0, \dots, n - i\}$.

Proof: We have

$p_{i,i+k}(x) = p_{i,i+k-1}(x) + d_{i,i+k}(x - x_i) \dots (x - x_{i+k-1})$ by construction. The proof follows by induction over k .

Divided Differences

The divided differences are defined by the recursion

$$d_{i,i} = y_i \quad \text{and} \quad d_{i,i+k} = \frac{d_{i+1,i+k} - d_{i,i+k-1}}{x_{i+k} - x_i}$$

for $i \in \{0, \dots, n\}$ and $k \in \{0, \dots, n - i\}$.

Theorem The functions $p_{i,i+k}(x)$ can be written in the form

$$p_{i,i+k}(x) = d_{i,i} + d_{i,i+1}(x - x_i) + \dots + d_{i,i+k}(x - x_i) \dots (x - x_{i+k-1}) .$$

for $i \in \{0, \dots, n\}$ and $k \in \{0, \dots, n - i\}$.

Proof: We have

$p_{i,i+k}(x) = p_{i,i+k-1}(x) + d_{i,i+k}(x - x_i) \dots (x - x_{i+k-1})$ by construction. The proof follows by induction over k .

Divided Differences

The divided differences are defined by the recursion

$$d_{i,i} = y_i \quad \text{and} \quad d_{i,i+k} = \frac{d_{i+1,i+k} - d_{i,i+k-1}}{x_{i+k} - x_i}$$

for $i \in \{0, \dots, n\}$ and $k \in \{0, \dots, n - i\}$.

Theorem The functions $p_{i,i+k}(x)$ can be written in the form

$$p_{i,i+k}(x) = d_{i,i} + d_{i,i+1}(x - x_i) + \dots + d_{i,i+k}(x - x_i) \dots (x - x_{i+k-1}) .$$

for $i \in \{0, \dots, n\}$ and $k \in \{0, \dots, n - i\}$.

Proof: We have

$p_{i,i+k}(x) = p_{i,i+k-1}(x) + d_{i,i+k}(x - x_i) \dots (x - x_{i+k-1})$ by construction. The proof follows by induction over k .

Visualization of Divided Differences

Divided differences can be computed recursively as visualized in the table below:

$$x_0 \mid y_0 \mid$$

For one data point, the constant polynomial $p(x) = y_0$ is the solution.

Visualization of Divided Differences

Divided differences can be computed recursively as visualized in the table below:

$$\begin{array}{c|c} x_0 & y_0 \\ x_1 & y_1 \end{array}$$

Let's assume a second data point becomes available.

Visualization of Divided Differences

Divided differences can be computed recursively as visualized in the table below:

$$\begin{array}{c|c|c} x_0 & y_0 & d_{01} \\ x_1 & y_1 & \end{array}$$

We compute $d_{01} = \frac{y_1 - y_0}{x_1 - x_0}$ and find the interpolating polynomial

$$p(x) = y_0 + d_{01}(x - x_0) .$$

Visualization of Divided Differences

Divided differences can be computed recursively as visualized in the table below:

$$\begin{array}{c|c|c} x_0 & y_0 & d_{01} \\ x_1 & y_1 & \\ x_2 & y_2 & \end{array}$$

Once the third data point is available...

Visualization of Divided Differences

Divided differences can be computed recursively as visualized in the table below:

$$\begin{array}{c|c|c} x_0 & y_0 & d_{01} \\ x_1 & y_1 & d_{12} \\ x_2 & y_2 & \end{array}$$

... we compute $d_{12} = \frac{y_2 - y_1}{x_2 - x_1}$ and ...

Visualization of Divided Differences

Divided differences can be computed recursively as visualized in the table below:

x_0	y_0	d_{01}	d_{02}
x_1	y_1	d_{12}	
x_2	y_2		

... $d_{02} = \frac{d_{12} - d_{01}}{x_2 - x_0}$. The interpolating polynomial is

$$p(x) = y_0 + d_{01}(x - x_0) + d_{02}(x - x_0)(x - x_1) .$$

Visualization of Divided Differences

Divided differences can be computed recursively as visualized in the table below:

x_0	y_0	d_{01}	d_{02}
x_1	y_1	d_{12}	
x_2	y_2		

... $d_{02} = \frac{d_{12} - d_{01}}{x_2 - x_0}$. The interpolating polynomial is

$$p(x) = y_0 + d_{01}(x - x_0) + d_{02}(x - x_0)(x - x_1) .$$

Visualization of Divided Differences

Divided differences can be computed recursively as visualized in the table below:

x_0	y_0	d_{01}	d_{02}	$\mathbf{d_{03}}$
x_1	y_1	d_{12}	$\mathbf{d_{13}}$	
x_2	y_2	$\mathbf{d_{23}}$		
x_3	$\mathbf{y_3}$			

In general, the complexity of adding one data point is $\mathbf{O}(n)$.

The complexity for computing all coefficients is $\mathbf{O}(n^2)$.

Visualization of Divided Differences

Divided differences can be computed recursively as visualized in the table below:

x_0	y_0	d_{01}	d_{02}	$\mathbf{d_{03}}$
x_1	y_1	d_{12}	$\mathbf{d_{13}}$	
x_2	y_2	$\mathbf{d_{23}}$		
x_3	$\mathbf{y_3}$			

In general, the complexity of adding one data point is $\mathbf{O}(n)$.

The complexity for computing all coefficients is $\mathbf{O}(n^2)$.

Visualization of Divided Differences

Divided differences can be computed recursively as visualized in the table below:

x_0	y_0	d_{01}	d_{02}	d_{03}	d_{04}
x_1	y_1	d_{12}	d_{13}	d_{14}	
x_2	y_2	d_{23}	d_{24}		
x_3	y_3	d_{34}			
x_4	y_4				

If we are only interested in computing p_{0n} we don't have to store all coefficients. Thus, the memory requirement scales with $O(n)$.

Visualization of Divided Differences

Divided differences can be computed recursively as visualized in the table below:

x_0	y_0	d_{01}	d_{02}	d_{03}	d_{04}	d_{05}
x_1	y_1	d_{12}	d_{13}	d_{14}	d_{15}	
x_2	y_2	d_{23}	d_{24}	d_{25}		
x_3	y_3	d_{34}	d_{35}			
x_4	y_4	d_{45}				
x_5	y_5					

We can keep on refining the scheme whenever new data points are available; $p(x) = \sum_{i=0}^n d_{0i} N_i(x)$.

Evaluation based on Horner's Scheme

Once the divided differences are computed the polynomial

$p(x) = \sum_{i=0}^n d_{0i} N_i(x)$ can be evaluated at any given point x based on

Horner's algorithm:

$$b_n = d_{0n}$$

$$b_k = d_{0k} + (x - x_k) b_{k+1} \quad k = n - 1, \dots, 0$$

$$p(x) = b_0.$$

Contents

- Problem Formulation
- Divided Differences
- **Interpolating Functions**
- Hermite Interpolation

Polynomial Approximation Error

Polynomial interpolation can be applied to any set of data points.

However, often we are interested in approximating functions, i.e., the data points are

$$y_i = f(x_i), \quad i \in \{0, \dots, n\},$$

where f is a $(n + 1)$ -times continuously differentiable function.

The difference between f and the interpolating polynomial can in this case be bounded by

$$|f(x) - p(x)| \leq \frac{1}{(n+1)!} \left| \frac{\partial^{n+1} f(\xi_x)}{\partial x^{n+1}} \prod_{j=0}^n (x - x_j) \right|$$

for a $\xi_x \in [\min_i x_i, \max_i x_i] = [\underline{x}, \overline{x}]$.

Polynomial Approximation Error

Polynomial interpolation can be applied to any set of data points.

However, often we are interested in approximating functions, i.e., the data points are

$$y_i = f(x_i), \quad i \in \{0, \dots, n\},$$

where f is a $(n + 1)$ -times continuously differentiable function.

The difference between f and the interpolating polynomial can in this case be bounded by

$$|f(x) - p(x)| \leq \frac{1}{(n+1)!} \left| \frac{\partial^{n+1} f(\xi_x)}{\partial x^{n+1}} \prod_{j=0}^n (x - x_j) \right|$$

for a $\xi_x \in [\min_i x_i, \max_i x_i] = [\underline{x}, \overline{x}]$.

Proof of Interpolation Error Bound

Analysis of divided differences:

1. Define the notation $f[x_i] \stackrel{\text{def}}{=} f(x_i)$.
2. Define the general divided differences recursively:

$$f[x_i, x_{i+1}, \dots, x_{i+k}] \stackrel{\text{def}}{=} \frac{f[x_{i+1}, \dots, x_{i+k}] - f[x_i, \dots, x_{i+k-1}]}{x_{i+k} - x_i}$$

3. Polynomial approximation in Newton basis:

$$p(x) = \sum_{i=0}^n f[x_0, \dots, x_i] N_i(x) \implies \frac{\partial^n}{\partial x^n} p(x) = n! f[x_0, \dots, x_n]$$

4. Rolle's theorem: $\frac{\partial^n}{\partial x^n} p(\xi) = \frac{\partial^n}{\partial x^n} f(\xi)$ for at least one $\xi \in [\underline{x}, \bar{x}]$.
5. All relations together yield the error bound from the previous slide!

Polynomial Approximation Error

Example 1: For the function $f(x) = \sin(x)$ all derivatives are uniformly bounded by 1 on the interval $[\underline{x}, \overline{x}]$. Thus, we have

$$|f(x) - p(x)| \leq \frac{1}{(n+1)!} \left| \prod_{j=1}^n (x - x_j) \right| \leq \frac{1}{(n+1)!} [\overline{x} - \underline{x}]^n,$$

which converges to 0 for $n \rightarrow \infty$ as long as $x_i \in [\underline{x}, \overline{x}]$.

Example 2: For the function $f(x) = \frac{1}{1+x^2}$ the n -th derivative satisfies

$$|f^{(n)}(x)| \approx 2^n n! \mathbf{O}(|x|^{-2-n})$$

Here, a uniform convergence of polynomial interpolation cannot be expected.

Polynomial Approximation Error

Example 1: For the function $f(x) = \sin(x)$ all derivatives are uniformly bounded by 1 on the interval $[\underline{x}, \overline{x}]$. Thus, we have

$$|f(x) - p(x)| \leq \frac{1}{(n+1)!} \left| \prod_{j=1}^n (x - x_j) \right| \leq \frac{1}{(n+1)!} [\overline{x} - \underline{x}]^n,$$

which converges to 0 for $n \rightarrow \infty$ as long as $x_i \in [\underline{x}, \overline{x}]$.

Example 2: For the function $f(x) = \frac{1}{1+x^2}$ the n -th derivative satisfies

$$|f^{(n)}(x)| \approx 2^n n! \mathbf{O}(|x|^{-2-n})$$

Here, a uniform convergence of polynomial interpolation cannot be expected.

Contents

- Problem Formulation
- Divided Differences
- Interpolating Functions
- Hermite Interpolation

Numerical Differentiation Revisited

What happens if we interpolate the points $(x, f(x))$ and $(x + h, f(x + h))$ for very small $h > 0$?

The slope of the interpolating polynomial approximates $f'(x)$:

$$\begin{array}{c|c|c} x & f(x) & \\ \hline x+h & f(x+h) & \frac{f(x+h)-f(x)}{h} \end{array}$$

For $h \rightarrow 0$ this divided difference table becomes

$$\begin{array}{c|c|c} x & f(x) & f'(x) \\ \hline x & f(x) & \end{array}$$

Numerical Differentiation Revisited

What happens if we interpolate the points $(x, f(x))$ and $(x + h, f(x + h))$ for very small $h > 0$?

The slope of the interpolating polynomial approximates $f'(x)$:

$$\begin{array}{c|c|c} x & f(x) & \\ \hline x+h & f(x+h) & \frac{f(x+h)-f(x)}{h} \end{array}$$

For $h \rightarrow 0$ this divided difference table becomes

$$\begin{array}{c|c|c} x & f(x) & f'(x) \\ \hline x & f(x) & \end{array}$$

Numerical Differentiation Revisited

What happens if we interpolate the points $(x, f(x))$ and $(x + h, f(x + h))$ for very small $h > 0$?

The slope of the interpolating polynomial approximates $f'(x)$:

$$\begin{array}{c|c|c} x & f(x) & \\ x+h & f(x+h) & \frac{f(x+h)-f(x)}{h} \end{array}$$

For $h \rightarrow 0$ this divided difference table becomes

$$\begin{array}{c|c|c} x & f(x) & f'(x) \\ x & f(x) & \end{array}$$

Numerical Differentiation Revisited

The same principle can be used to approximate higher order derivatives, for example

$$\begin{array}{c|c|c|c}
 x-h & f(x-h) & \frac{f(x)-f(x-h)}{h} & \frac{f(x-h)-2f(x)+f(x+h)}{2h^2} \\
 x & f(x) & \frac{f(x+h)-f(x)}{h} & \\
 x+h & f(x+h) & &
 \end{array}$$

For $h \rightarrow 0$ this divided difference table becomes

$$\begin{array}{c|c|c|c}
 x & f(x) & f'(x) & \frac{1}{2}f''(x) \\
 x & f(x) & f'(x) & \\
 x & f(x) & &
 \end{array}$$

Numerical Differentiation Revisited

The same principle can be used to approximate higher order derivatives, for example

$$\begin{array}{c|c|c|c}
 x-h & f(x-h) & \frac{f(x)-f(x-h)}{h} & \frac{f(x-h)-2f(x)+f(x+h)}{2h^2} \\
 x & f(x) & \frac{f(x+h)-f(x)}{h} & \\
 x+h & f(x+h) & &
 \end{array}$$

For $h \rightarrow 0$ this divided difference table becomes

$$\begin{array}{c|c|c}
 x & f(x) & f'(x) \\
 x & f(x) & f'(x) \\
 x & f(x) &
 \end{array}
 \quad \frac{1}{2}f''(x)$$

Hermite Interpolation

Hermite's interpolation problem is to find a polynomial of degree

$\sum_{i=0}^n m_i$ satisfying the condition

$$\frac{\partial^k p_i}{\partial x^k}(x_i) = y_i^k, \quad k \in \{0, \dots, m_i - 1\}$$

for all $i \in \{0, \dots, n\}$ and data $y_i^k \in \mathbb{R}$.

The solution polynomial can be found in analogy to the standard interpolation problem with the only difference that the points x_i are added m_i times to the divided difference table. The divided differences are then replaced with the corresponding derivative terms.

Hermite Interpolation

Hermite's interpolation problem is to find a polynomial of degree

$\sum_{i=0}^n m_i$ satisfying the condition

$$\frac{\partial^k p_i}{\partial x^k}(x_i) = y_i^k, \quad k \in \{0, \dots, m_i - 1\}$$

for all $i \in \{0, \dots, n\}$ and data $y_i^k \in \mathbb{R}$.

The solution polynomial can be found in analogy to the standard interpolation problem with the only difference that the points x_i are added m_i times to the divided difference table. The divided differences are then replaced with the corresponding derivative terms.

Hermite Interpolation

Example: we want to find a polynomial of degree 3, which satisfies

$$p(a) = f(a), p'(a) = f'(a), p(b) = f(b), \text{ and } p'(b) = f'(b)$$

for given points a, b and a continuously differentiable function f . The corresponding divided difference table is

a	$f(a)$	$f'(a)$	$\frac{f(b)-f(a)-f'(a)(b-a)}{(b-a)^2}$	$\frac{2(f(a)-f(b))+(f'(a)+f'(b))(b-a)}{(b-a)^3}$
a	$f(a)$	$\frac{f(b)-f(a)}{b-a}$	$\frac{f(a)-f(b)+f'(b)(b-a)}{(b-a)^2}$	
b	$f(b)$	$f'(b)$		
b	$f(b)$			

This table yields the solution polynomial w.r.t. the Newton basis.

Hermite Interpolation

Example: we want to find a polynomial of degree 3, which satisfies

$$p(a) = f(a), p'(a) = f'(a), p(b) = f(b), \text{ and } p'(b) = f'(b)$$

for given points a, b and a continuously differentiable function f . The corresponding divided difference table is

a	$f(a)$	$f'(a)$	$\frac{f(b)-f(a)-f'(a)(b-a)}{(b-a)^2}$	$\frac{2(f(a)-f(b))+(f'(a)+f'(b))(b-a)}{(b-a)^3}$
a	$f(a)$	$\frac{f(b)-f(a)}{b-a}$	$\frac{f(a)-f(b)+f'(b)(b-a)}{(b-a)^2}$	
b	$f(b)$	$f'(b)$		
b	$f(b)$			

This table yields the solution polynomial w.r.t. the Newton basis.

Approximation Error of Hermite Interpolation

For the general Hermite interpolation, the difference between f and the interpolating polynomial p is bounded by

$$|f(x) - p(x)| \leq \frac{1}{(m+1)!} \left| \frac{\partial^{m+1} f(\xi_x)}{\partial x^{m+1}} \prod_{j=1}^n (x - x_j)^{m_j} \right|$$

for a $\xi_x \in [\min_i x_i, \max_i x_i]$ and $m = \sum_{i=0}^n m_i$.

(for a proof see, e.g., the Numerical Analysis book by Burden and Faires)

Summary

- There exists a unique polynomial of order n , which interpolates the data points $(x_0, y_0), \dots, (x_n, y_n)$, if $x_i \neq x_j$ for all $i \neq j$.
- The polynomial is given by $p(x) = \sum_{i=0}^n y_i L_i(x)$, but this representation is not used in practice.
- We have discussed how to use divided differences for computing interpolating polynomials.
- If the derivatives of f are uniformly bounded, the polynomial interpolation converges to the exact function for $n \rightarrow \infty$.
- Hermite interpolation additionally interpolates derivatives.

Chapter 5

Applications of Polynomial Interpolation

Applications of Polynomial Interpolation

- Extrapolation
- Splines

Contents

- Extrapolation

- Splines

Problem Formulation

We have a function $f : \mathbb{R} \rightarrow \mathbb{R}$ that can be evaluated for all $h > 0$, but f cannot be evaluated easily at $h = 0$. We are interested in computing

$$\lim_{h \rightarrow 0} f(h) .$$

Example: We want to evaluate the expression

$$\lim_{h \rightarrow 0} \frac{\exp(h) - 1}{\sin(h)} .$$

Here, we cannot simply substitute $h = 0$, as we would have to divide by $\sin(0) = 0$.

Problem Formulation

We have a function $f : \mathbb{R} \rightarrow \mathbb{R}$ that can be evaluated for all $h > 0$, but f cannot be evaluated easily at $h = 0$. We are interested in computing

$$\lim_{h \rightarrow 0} f(h) .$$

Example: We want to evaluate the expression

$$\lim_{h \rightarrow 0} \frac{\exp(h) - 1}{\sin(h)} .$$

Here, we cannot simply substitute $h = 0$, as we would have to divide by $\sin(0) = 0$.

L'Hospital's rule

There are several solution strategies for computing limites whose applicability depends on the situation. For example L'Hospital's rule gives

$$\lim_{h \rightarrow 0} \frac{\exp(h) - 1}{\sin(h)} = \frac{\exp(0)}{\cos(0)} = 1 .$$

Thus, if we use algorithmic differentiation, we may be able to implement L'Hospital's rule for expressions of the form

$$\lim_{h \rightarrow 0} \frac{f_1(h)}{f_2(h)} = \lim_{h \rightarrow 0} \frac{f_1'(h)}{f_2'(h)} ,$$

assuming $f_1(0) = f_2(0) = 0$ and that the derivatives of f_1 and f_2 can be computed easily. However, in general L'Hospital's rule may not be applicable.

L'Hospital's rule

There are several solution strategies for computing limites whose applicability depends on the situation. For example L'Hospital's rule gives

$$\lim_{h \rightarrow 0} \frac{\exp(h) - 1}{\sin(h)} = \frac{\exp(0)}{\cos(0)} = 1 .$$

Thus, if we use algorithmic differentiation, we may be able to implement L'Hospital's rule for expressions of the form

$$\lim_{h \rightarrow 0} \frac{f_1(h)}{f_2(h)} = \lim_{h \rightarrow 0} \frac{f_1'(h)}{f_2'(h)} ,$$

assuming $f_1(0) = f_2(0) = 0$ and that the derivatives of f_1 and f_2 can be computed easily. However, in general L'Hospital's rule may not be applicable.

L'Hospital's rule

There are several solution strategies for computing limites whose applicability depends on the situation. For example L'Hospital's rule gives

$$\lim_{h \rightarrow 0} \frac{\exp(h) - 1}{\sin(h)} = \frac{\exp(0)}{\cos(0)} = 1 .$$

Thus, if we use algorithmic differentiation, we may be able to implement L'Hospital's rule for expressions of the form

$$\lim_{h \rightarrow 0} \frac{f_1(h)}{f_2(h)} = \lim_{h \rightarrow 0} \frac{f_1'(h)}{f_2'(h)} ,$$

assuming $f_1(0) = f_2(0) = 0$ and that the derivatives of f_1 and f_2 can be computed easily. However, in general L'Hospital's rule may not be applicable.

Extrapolation

An alternative to L'Hospital's rule is to use extrapolation. For this aim, we evaluate f at a decreasing sequence of points $h_i > 0$, e.g., $h_1 = \frac{1}{4}$, $h_2 = \frac{1}{8}$, and $h_3 = \frac{1}{16}$.

Next, we compute a polynomial $p(x)$ interpolating the evaluation points

$$(h_1, f(h_1)), (h_2, f(h_2)), (h_3, f(h_3)), \dots$$

and use the approximation $\lim_{h \rightarrow 0} \frac{f_1(h)}{f_2(h)} \approx p(0)$.

Extrapolation

An alternative to L'Hospital's rule is to use extrapolation. For this aim, we evaluate f at a decreasing sequence of points $h_i > 0$, e.g., $h_1 = \frac{1}{4}$, $h_2 = \frac{1}{8}$, and $h_3 = \frac{1}{16}$.

Next, we compute a polynomial $p(x)$ interpolating the evaluation points

$$(h_1, f(h_1)), (h_2, f(h_2)), (h_3, f(h_3)), \dots$$

and use the approximation $\lim_{h \rightarrow 0} \frac{f_1(h)}{f_2(h)} \approx p(0)$.

Extrapolation Error

If the function f is $(n + 1)$ -times continuously differentiable in a small neighborhood of 0 and $p(x)$ a polynomial of degree $\leq n$ which interpolates the points

$$(h_1, f(h_1)), (h_2, f(h_2)), \dots, (h_n, f(h_n))$$

for small $h \geq h_1 > h_2 > \dots > h_n > 0$, then we have

$$|p(0) - f(0)| \leq \mathbf{O}(h^{n+1}) .$$

(the proof follows from Taylor's theorem)

Contents

- Extrapolation

- Splines

Limitations of Polynomial Interpolation

We have learned in the previous lecture that the interpolation polynomial may not converge uniformly for $n \rightarrow \infty$.

Example: [Runge's phenomenon] Interpolation of

$$f(x) = \frac{1}{x^2 + 1}$$

on equidistant grid on the interval $[-5, 5]$ with high order polynomials leads to (unwanted) highly oscillatory interpolation.

Splines

One strategy to overcome this problem is to use splines. Here, we break up the whole interval $[x_{\min}, x_{\max}]$ into sub-intervals and interpolate the function f on each of these sub-intervals with a polynomial of moderate degree (often $n = 3$).

The most common splines are “piecewise linear interpolation” and “cubic splines”.

In this lecture we have a closer look at cubic splines.

Splines

One strategy to overcome this problem is to use splines. Here, we break up the whole interval $[x_{\min}, x_{\max}]$ into sub-intervals and interpolate the function f on each of these sub-intervals with a polynomial of moderate degree (often $n = 3$).

The most common splines are “piecewise linear interpolation” and “cubic splines”.

In this lecture we have a closer look at cubic splines.

Splines

One strategy to overcome this problem is to use splines. Here, we break up the whole interval $[x_{\min}, x_{\max}]$ into sub-intervals and interpolate the function f on each of these sub-intervals with a polynomial of moderate degree (often $n = 3$).

The most common splines are “piecewise linear interpolation” and “cubic splines”.

In this lecture we have a closer look at cubic splines.

Cubic Splines using Hermite Interpolation

Let f be continuously differentiable. We divide the whole interval $[x_{\min}, x_{\max}]$ into n sub-intervals

$$x_{\min} = x_0 < x_1 < \dots < x_n = x_{\max} .$$

Now, we search for a piecewise cubic approximation of the form

$$p(x) = \left\{ \begin{array}{ll} x_{\min} + f'(x_{\min})(x - x_{\min}) & \text{if } x < x_{\min} \\ p_i(x) & \text{if } x \in [x_i, x_{i+1}] \\ x_{\max} + f'(x_{\max})(x - x_{\max}) & \text{if } x > x_{\max} \end{array} \right\}$$

with $i \in \{0, \dots, n-1\}$), where the cubic polynomials p_i satisfy

$$p_i(x_i) = f(x_i) \quad \text{and} \quad p'_i(x_i) = f'(x_i)$$

$$p_i(x_{i+1}) = f(x_{i+1}) \quad \text{and} \quad p'_i(x_{i+1}) = f'(x_{i+1}) \quad (\text{Hermite interpolation!})$$

for all $i \in \{0, \dots, n\}$. The function p is called a cubic spline.

Cubic Splines using Hermite Interpolation

Let f be continuously differentiable. We divide the whole interval $[x_{\min}, x_{\max}]$ into n sub-intervals

$$x_{\min} = x_0 < x_1 < \dots < x_n = x_{\max} .$$

Now, we search for a piecewise cubic approximation of the form

$$p(x) = \left\{ \begin{array}{ll} x_{\min} + f'(x_{\min})(x - x_{\min}) & \text{if } x < x_{\min} \\ p_i(x) & \text{if } x \in [x_i, x_{i+1}] \\ x_{\max} + f'(x_{\max})(x - x_{\max}) & \text{if } x > x_{\max} \end{array} \right\}$$

with $i \in \{0, \dots, n-1\}$), where the cubic polynomials p_i satisfy

$$p_i(x_i) = f(x_i) \quad \text{and} \quad p'_i(x_i) = f'(x_i)$$

$$p_i(x_{i+1}) = f(x_{i+1}) \quad \text{and} \quad p'_i(x_{i+1}) = f'(x_{i+1}) \quad (\text{Hermite interpolation!})$$

for all $i \in \{0, \dots, n\}$. The function p is called a cubic spline.

Cubic Splines using Hermite Interpolation

Let f be continuously differentiable. We divide the whole interval $[x_{\min}, x_{\max}]$ into n sub-intervals

$$x_{\min} = x_0 < x_1 < \dots < x_n = x_{\max} .$$

Now, we search for a piecewise cubic approximation of the form

$$p(x) = \left\{ \begin{array}{ll} x_{\min} + f'(x_{\min})(x - x_{\min}) & \text{if } x < x_{\min} \\ p_i(x) & \text{if } x \in [x_i, x_{i+1}] \\ x_{\max} + f'(x_{\max})(x - x_{\max}) & \text{if } x > x_{\max} \end{array} \right\}$$

with $i \in \{0, \dots, n-1\}$), where the cubic polynomials p_i satisfy

$$p_i(x_i) = f(x_i) \quad \text{and} \quad p'_i(x_i) = f'(x_i)$$

$$p_i(x_{i+1}) = f(x_{i+1}) \quad \text{and} \quad p'_i(x_{i+1}) = f'(x_{i+1}) \quad (\text{Hermite interpolation!})$$

for all $i \in \{0, \dots, n\}$. The function p is called a cubic spline.

Natural Cubic Splines

Another way to construct cubic splines is by imposing the following conditions on the piecewise cubic function p :

1. The function p satisfies $p(x_i) = f(x_i)$, $i \in \{0, \dots, n\}$, ($2n$ conditions)
2. The function p is twice continuously differentiable, ($2(n - 1)$ conditions)
3. We have $p''(x_{\min}) = p''(x_{\max}) = 0$, (2 conditions)

In total, we have $4n$ conditions determining the coefficients of the n cubic polynomials.

Natural Cubic Splines

Notation: $p_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$, assume $x_{i+1} - x_i = h$.

1. Interpolation ($i \in \{1, \dots, n\}$):

$$a_i = f(x_i) \quad \text{and} \quad a_i - b_i h + c_i h^2 - d_i h^3 = f(x_{i-1})$$

2. First derivatives ($i \in \{1, \dots, n-1\}$):

$$b_i = b_{i+1} - 2c_{i+1}h + 3d_{i+1}h^2$$

3. Second derivatives ($i \in \{1, \dots, n-1\}$):

$$c_i = c_{i+1} - 3d_{i+1}h$$

4. Boundaries: $c_n = 0 \quad c_1 - 3d_1h = 0$.

Natural Cubic Splines

Notation: $p_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$, assume $x_{i+1} - x_i = h$.

1. Interpolation ($i \in \{1, \dots, n\}$):

$$a_i = f(x_i) \quad \text{and} \quad a_i - b_i h + c_i h^2 - d_i h^3 = f(x_{i-1})$$

2. First derivatives ($i \in \{1, \dots, n-1\}$):

$$b_i = b_{i+1} - 2c_{i+1}h + 3d_{i+1}h^2$$

3. Second derivatives ($i \in \{1, \dots, n-1\}$):

$$c_i = c_{i+1} - 3d_{i+1}h$$

4. Boundaries: $c_n = 0 \quad c_1 - 3d_1h = 0$.

Natural Cubic Splines

Notation: $p_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$, assume $x_{i+1} - x_i = h$.

1. Interpolation ($i \in \{1, \dots, n\}$):

$$a_i = f(x_i) \quad \text{and} \quad a_i - b_i h + c_i h^2 - d_i h^3 = f(x_{i-1})$$

2. First derivatives ($i \in \{1, \dots, n-1\}$):

$$b_i = b_{i+1} - 2c_{i+1}h + 3d_{i+1}h^2$$

3. Second derivatives ($i \in \{1, \dots, n-1\}$):

$$c_i = c_{i+1} - 3d_{i+1}h$$

4. Boundaries: $c_n = 0 \quad c_1 - 3d_1h = 0$.

Natural Cubic Splines

Notation: $p_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$, assume $x_{i+1} - x_i = h$.

1. Interpolation ($i \in \{1, \dots, n\}$):

$$a_i = f(x_i) \quad \text{and} \quad a_i - b_i h + c_i h^2 - d_i h^3 = f(x_{i-1})$$

2. First derivatives ($i \in \{1, \dots, n-1\}$):

$$b_i = b_{i+1} - 2c_{i+1}h + 3d_{i+1}h^2$$

3. Second derivatives ($i \in \{1, \dots, n-1\}$):

$$c_i = c_{i+1} - 3d_{i+1}h$$

4. Boundaries: $c_n = 0 \quad c_1 - 3d_1h = 0$.

Natural Cubic Splines

The equation system can be simplified as follows:

1. We know the coefficients $a_i = f(x_i)$.
2. From $c_i = c_{i+1} - 3d_{i+1}h$ we conclude $d_i = \frac{c_i - c_{i-1}}{3h}$ (define $c_0 = 0$).
3. From $a_i - b_i h + c_i h^2 - d_i h^3 = f(x_{i-1})$ we conclude

$$b_i = \frac{f(x_i) - f(x_{i-1})}{h} + \frac{2c_i + c_{i-1}}{3}h.$$

So, in summary, we can express the coefficients a_i , b_i , and d_i in dependence on the sequence $c_0, \dots, c_n$.

From the boundary conditions we know that $c_0 = c_n = 0$.

Natural Cubic Splines

The equation system can be simplified as follows:

1. We know the coefficients $a_i = f(x_i)$.
2. From $c_i = c_{i+1} - 3d_{i+1}h$ we conclude $d_i = \frac{c_i - c_{i-1}}{3h}$ (define $c_0 = 0$).
3. From $a_i - b_i h + c_i h^2 - d_i h^3 = f(x_{i-1})$ we conclude

$$b_i = \frac{f(x_i) - f(x_{i-1}))}{h} + \frac{2c_i + c_{i-1}}{3}h .$$

So, in summary, we can express the coefficients a_i , b_i , and d_i in dependence on the sequence $c_0, \dots, c_n$.

From the boundary conditions we know that $c_0 = c_n = 0$.

Natural Cubic Splines

The equation system can be simplified as follows:

1. We know the coefficients $a_i = f(x_i)$.
2. From $c_i = c_{i+1} - 3d_{i+1}h$ we conclude $d_i = \frac{c_i - c_{i-1}}{3h}$ (define $c_0 = 0$).
3. From $a_i - b_i h + c_i h^2 - d_i h^3 = f(x_{i-1})$ we conclude

$$b_i = \frac{f(x_i) - f(x_{i-1}))}{h} + \frac{2c_i + c_{i-1}}{3}h .$$

So, in summary, we can express the coefficients a_i , b_i , and d_i in dependence on the sequence $c_0, \dots, c_n$.

From the boundary conditions we know that $c_0 = c_n = 0$.

Natural Cubic Splines

The equation system can be simplified as follows:

1. We know the coefficients $a_i = f(x_i)$.
2. From $c_i = c_{i+1} - 3d_{i+1}h$ we conclude $d_i = \frac{c_i - c_{i-1}}{3h}$ (define $c_0 = 0$).
3. From $a_i - b_i h + c_i h^2 - d_i h^3 = f(x_{i-1})$ we conclude

$$b_i = \frac{f(x_i) - f(x_{i-1})}{h} + \frac{2c_i + c_{i-1}}{3}h.$$

So, in summary, we can express the coefficients a_i , b_i , and d_i in dependence on the sequence $c_0, \dots, c_n$.

From the boundary conditions we know that $c_0 = c_n = 0$.

Natural Cubic Splines

The next step is a bit cumbersome: we have to substitute our expression for a_i , b_i , and d_i into $b_i = b_{i+1} - 2c_{i+1}h + 3d_{i+1}h^2$. This gives the recursion:

$$\begin{aligned} & \frac{f(x_i) - f(x_{i-1}))}{h} + \frac{2c_i + c_{i-1}}{3}h \\ &= \frac{f(x_{i+1}) - f(x_i)}{h} + \frac{2c_{i+1} + c_i}{3}h - 2c_{i+1}h + (c_{i+1} - c_i)h \\ &\implies h(c_{i-1} + 4c_i + c_{i+1}) = r_i \end{aligned}$$

for $i = 1, \dots, n-1$ and $c_0 = c_n = 0$ as well as the short-hand

$$r_i = \frac{3}{h} (f(x_{i+1}) - 2f(x_i) + f(x_{i-1})).$$

Natural Cubic Splines

The next step is a bit cumbersome: we have to substitute our expression for a_i , b_i , and d_i into $b_i = b_{i+1} - 2c_{i+1}h + 3d_{i+1}h^2$. This gives the recursion:

$$\begin{aligned} & \frac{f(x_i) - f(x_{i-1}))}{h} + \frac{2c_i + c_{i-1}}{3}h \\ &= \frac{f(x_{i+1}) - f(x_i)}{h} + \frac{2c_{i+1} + c_i}{3}h - 2c_{i+1}h + (c_{i+1} - c_i)h \\ &\implies h(c_{i-1} + 4c_i + c_{i+1}) = r_i \end{aligned}$$

for $i = 1, \dots, n-1$ and $c_0 = c_n = 0$ as well as the short-hand

$$r_i = \frac{3}{h} (f(x_{i+1}) - 2f(x_i) + f(x_{i-1})).$$

Natural Cubic Splines

Thus, the equation system for $c_1, \dots, c_{n-1}$ can be written in the form

$$\begin{pmatrix} 4h & h & & \dots & 0 \\ h & 4h & h & & \\ & \ddots & \ddots & \ddots & \\ \vdots & & h & 4h & h \\ 0 & & & h & 4h \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n-2} \\ c_{n-1} \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_{n-2} \\ r_{n-1} \end{pmatrix}$$

This equation system has a unique solution and can be solved using tridiagonal matrix inversion algorithms. (for non-equidistant points x_i a similar result can be obtained)

Properties of Natural Cubic Splines

The natural cubic splines satisfy the inequality

$$\int_{x_{\min}}^{x_{\max}} |p''(x)|^2 dx \leq \int_{x_{\min}}^{x_{\max}} |f''(x)|^2 dx .$$

“The natural cubic spline p never oscillates more than the function f .”

Proof: Any twice continuously differentiable function w with

$w(x_0) = w(x_1) = \dots w(x_n) = 0$ satisfies

$$\begin{aligned} \int_{x_{\min}}^{x_{\max}} p''(x)w''(x)dx &= \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} p''(x)w''(x)dx \\ &= \sum_{i=0}^{n-1} p''w'|_{x_i}^{x_{i+1}} - \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} p'''(x)w'(x)dx \\ &= \sum_{i=0}^{n-1} \{ p''w'|_{x_i}^{x_{i+1}} - p'''w|_{x_i}^{x_{i+1}} \} \\ &\quad + \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} p'''(x)w(x)dx . \end{aligned}$$

Properties of Natural Cubic Splines

The natural cubic splines satisfy the inequality

$$\int_{x_{\min}}^{x_{\max}} |p''(x)|^2 dx \leq \int_{x_{\min}}^{x_{\max}} |f''(x)|^2 dx .$$

“The natural cubic spline p never oscillates more than the function f .”

Proof: Any twice continuously differentiable function w with

$w(x_0) = w(x_1) = \dots w(x_n) = 0$ satisfies

$$\begin{aligned} \int_{x_{\min}}^{x_{\max}} p''(x)w''(x)dx &= \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} p''(x)w''(x)dx \\ &= \sum_{i=0}^{n-1} p''w'|_{x_i}^{x_{i+1}} - \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} p'''(x)w'(x)dx \\ &= \sum_{i=0}^{n-1} \{ p''w'|_{x_i}^{x_{i+1}} - p'''w|_{x_i}^{x_{i+1}} \} \\ &\quad + \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} p''''(x)w(x)dx . \end{aligned}$$

Proof (continued)

As p is a cubic polynomial, we have $p'''' = 0$ which implies

$$\begin{aligned}\int_{x_{\min}}^{x_{\max}} p''(x)w''(x)dx &= \sum_{i=0}^{n-1} \{p''w'|_{x_i}^{x_{i+1}} - p'''w|_{x_i}^{x_{i+1}}\} \\ &= \sum_{i=0}^{n-1} p''w'|_{x_i}^{x_{i+1}}\end{aligned}$$

as we have $w(x_i) = w(x_{i+1}) = 0$. Finally, we find

$$\begin{aligned}\int_{x_{\min}}^{x_{\max}} p''(x)w''(x)dx &= \sum_{i=0}^{n-1} p''w'|_{x_i}^{x_{i+1}} \\ &= p''(x_n)w'(x_n) - p''(x_0)w'(x_0) = 0 ,\end{aligned}$$

since $p''(x_0) = p''(x_n) = 0$. For $w = f - p$ we find

$$\int_{x_{\min}}^{x_{\max}} |f''(x)|^2 dx = \int_{x_{\min}}^{x_{\max}} |p''(x) + w''(x)|^2 dx \geq \int_{x_{\min}}^{x_{\max}} |p''(x)|^2 dx .$$

Proof (continued)

As p is a cubic polynomial, we have $p'''' = 0$ which implies

$$\begin{aligned}\int_{x_{\min}}^{x_{\max}} p''(x)w''(x)dx &= \sum_{i=0}^{n-1} \{ p''w'|_{x_i}^{x_{i+1}} - p'''w|_{x_i}^{x_{i+1}} \} \\ &= \sum_{i=0}^{n-1} p''w'|_{x_i}^{x_{i+1}}\end{aligned}$$

as we have $w(x_i) = w(x_{i+1}) = 0$. Finally, we find

$$\begin{aligned}\int_{x_{\min}}^{x_{\max}} p''(x)w''(x)dx &= \sum_{i=0}^{n-1} p''w'|_{x_i}^{x_{i+1}} \\ &= p''(x_n)w'(x_n) - p''(x_0)w'(x_0) = 0 ,\end{aligned}$$

since $p''(x_0) = p''(x_n) = 0$. For $w = f - p$ we find

$$\int_{x_{\min}}^{x_{\max}} |f''(x)|^2 dx = \int_{x_{\min}}^{x_{\max}} |p''(x) + w''(x)|^2 dx \geq \int_{x_{\min}}^{x_{\max}} |p''(x)|^2 dx .$$

Summary

- Extrapolation can be used to compute limit values of functions with high accuracy (only needed if L'Hospital's rule in combination with algorithmic differentiation is not applicable).
- We have discussed two ways to construct cubic splines:
 - Hermite interpolation (continuously differentiable interpolation)
 - Natural Splines (twice continuously differentiable interpolation)
- Natural splines do not require us to evaluate derivatives of f .
- The average of the square of the second derivative of a natural spline p is bounded by the average of the square of the second derivative of the original function f .

Chapter 6

Gauss Approximation

Gauss Approximation

- Problem Formulation
- Gram-Schmidt Algorithm
- Orthogonal Polynomials
- Solution of Gauss' Approximation Problem

Contents

- Problem Formulation
- Gram-Schmidt Algorithm
- Orthogonal Polynomials
- Solution of Gauss' Approximation Problem

Problem Formulation

Gauss' approximation problem is to construct a polynomial p of degree $\leq n$ which solves

$$\min_{p \in P_n} \|f - p\| \quad \text{with} \quad \|g\| = \sqrt{\int_a^b g(x)^2 dx} .$$

denoting the L_2 -norm. Here P_n denotes the set of polynomials $p : \mathbb{R} \rightarrow \mathbb{R}$ with degree $\leq n$ and $f : \mathbb{R} \rightarrow \mathbb{R}$ is a given function.

L_2 -Scalar Products

Recall that the L_2 -scalar product of two functions $f, g : [a, b] \rightarrow \mathbb{R}$ on an interval $[a, b]$ is given by

$$\langle f, g \rangle = \int_a^b f(x)g(x) \, dx .$$

In this notation, the L_2 -norm can be written in the form

$$\|f\| = \sqrt{\langle f, f \rangle}$$

In particular, the Cauchy-Schwartz inequality can be written in the form

$$\langle f, g \rangle \leq \|f\| \cdot \|g\| .$$

L_2 -Scalar Products

Recall that the L_2 -scalar product of two functions $f, g : [a, b] \rightarrow \mathbb{R}$ on an interval $[a, b]$ is given by

$$\langle f, g \rangle = \int_a^b f(x)g(x) \, dx .$$

In this notation, the L_2 -norm can be written in the form

$$\|f\| = \sqrt{\langle f, f \rangle}$$

In particular, the Cauchy-Schwartz inequality can be written in the form

$$\langle f, g \rangle \leq \|f\| \cdot \|g\| .$$

Optimality Conditions

Theorem The polynomial p is a solution of the minimization problem

$$\min_{p \in P_n} \|f - p\| \ .$$

if and only if we have $\langle f - p, q \rangle = 0$ for all $q \in P_n$.

Proof:

Optimality Conditions

Theorem The polynomial p is a solution of the minimization problem

$$\min_{p \in P_n} \|f - p\| .$$

if and only if we have $\langle f - p, q \rangle = 0$ for all $q \in P_n$.

Proof:

Step 1: If $p \in P_n$ is an optimal approximation, the function

$F(t) := \|f - p - tq\|^2$ must have a minimizer at $t = 0$ for all $q \in P_n$.

Thus, we must have

$$0 = \left. \frac{\partial}{\partial t} \|f - p - tq\|^2 \right|_{t=0} = \langle f - p, q \rangle .$$

Optimality Conditions

Theorem The polynomial p is a solution of the minimization problem

$$\min_{p \in P_n} \|f - p\| \text{ .}$$

if and only if we have $\langle f - p, q \rangle = 0$ for all $q \in P_n$.

Proof:

Step 2: The other way around, if p satisfies $\langle f - p, q \rangle = 0$ for all $q \in P_n$, we have

$$\|f - p\|^2 = \langle f - p, f - q \rangle + \underbrace{\langle f - p, q - p \rangle}_{=0} \leq \|f - p\| \|f - q\|$$

and thus $\|f - p\| \leq \min_{q \in P_n} \|f - q\|$, i.e., p is a minimizer. □

Uniqueness of Solutions

In the following, we check that the Gauss problem has at most one solution:

If two functions $p_1, p_2 \in P_n$ satisfy the optimality condition

$$\langle f - p_1, q \rangle = \langle f - p_2, q \rangle = 0 \quad \text{for all } q \in P_n ,$$

we also have $\langle p_1 - p_2, q \rangle = 0$. Thus, for $q = p_1 - p_2$, we find

$$\|p_1 - p_2\| = 0 ,$$

which implies $p_1 = p_2$.

Proving existence is a bit more difficult; we will come back to it later...

Uniqueness of Solutions

In the following, we check that the Gauss problem has at most one solution:

If two functions $p_1, p_2 \in P_n$ satisfy the optimality condition

$$\langle f - p_1, q \rangle = \langle f - p_2, q \rangle = 0 \quad \text{for all } q \in P_n ,$$

we also have $\langle p_1 - p_2, q \rangle = 0$. Thus, for $q = p_1 - p_2$, we find

$$\|p_1 - p_2\| = 0 ,$$

which implies $p_1 = p_2$.

Proving existence is a bit more difficult; we will come back to it later...

Uniqueness of Solutions

In the following, we check that the Gauss problem has at most one solution:

If two functions $p_1, p_2 \in P_n$ satisfy the optimality condition

$$\langle f - p_1, q \rangle = \langle f - p_2, q \rangle = 0 \quad \text{for all } q \in P_n ,$$

we also have $\langle p_1 - p_2, q \rangle = 0$. Thus, for $q = p_1 - p_2$, we find

$$\|p_1 - p_2\| = 0 ,$$

which implies $p_1 = p_2$.

Proving existence is a bit more difficult; we will come back to it later...

Uniqueness of Solutions

In the following, we check that the Gauss problem has at most one solution:

If two functions $p_1, p_2 \in P_n$ satisfy the optimality condition

$$\langle f - p_1, q \rangle = \langle f - p_2, q \rangle = 0 \quad \text{for all } q \in P_n ,$$

we also have $\langle p_1 - p_2, q \rangle = 0$. Thus, for $q = p_1 - p_2$, we find

$$\|p_1 - p_2\| = 0 ,$$

which implies $p_1 = p_2$.

Proving existence is a bit more difficult; we will come back to it later...

Contents

- Problem Formulation
- Gram-Schmidt Algorithm
- Orthogonal Polynomials
- Solution of Gauss' Approximation Problem

Gram-Schmidt Algorithm

Let's recall some basic linear algebra:

Assume we have k vectors $a_1, \dots, a_k \in \mathbb{R}^n$. Gram-Schmidt's algorithm can be used to check for linear independence:

Gram-Schmidt Algorithm:

For $i = 1, \dots, k$:

- Orthogonalization. $\bar{q}_i = a_i - \langle q_1, a_i \rangle q_1 - \dots - \langle q_{i-1}, a_i \rangle q_{i-1}$.
- Test for dependence. If $\bar{q}_i = 0$, quit.
- Normalization. $q_i = \frac{\bar{q}_i}{\|\bar{q}_i\|}$.

If the algorithm does not quit, the vectors a_i are linearly independent.

Gram-Schmidt Algorithm

Let's recall some basic linear algebra:

Assume we have k vectors $a_1, \dots, a_k \in \mathbb{R}^n$. Gram-Schmidt's algorithm can be used to check for linear independence:

Gram-Schmidt Algorithm:

For $i = 1, \dots, k$:

- Orthogonalization. $\bar{q}_i = a_i - \langle q_1, a_i \rangle q_1 - \dots - \langle q_{i-1}, a_i \rangle q_{i-1}$.
- Test for dependence. If $\bar{q}_i = 0$, quit.
- Normalization. $q_i = \frac{\bar{q}_i}{\|\bar{q}_i\|}$.

If the algorithm does not quit, the vectors a_i are linearly independent.

Gram-Schmidt Algorithm

The Gram-Schmidt Algorithm computes the vectors $q_1, \dots, q_k$. These vectors are orthonormal. This can be proven by induction:

- The vector $q_1 = \frac{a_1}{\|a_1\|}$ is normalized.
- Assume the vectors $q_1, \dots, q_{i-1}$ are already orthonormal. Then, the vector $\bar{q}_i$ satisfies

$$\begin{aligned}\langle \bar{q}_i, q_j \rangle &= \langle a_i, q_j \rangle - \sum_{k=1}^{i-1} \langle q_k, a_i \rangle \langle q_k, q_j \rangle \\ &= \langle a_i, q_j \rangle - \sum_{k=1}^{i-1} \langle q_k, a_i \rangle \delta_{k,j} = 0\end{aligned}$$

for all $j \in \{1, \dots, i-1\}$, i.e., the vectors $q_1, \dots, q_i$ are orthonormal.

Gram-Schmidt Algorithm

The Gram-Schmidt Algorithm computes the vectors $q_1, \dots, q_k$. These vectors are orthonormal. This can be proven by induction:

- The vector $q_1 = \frac{a_1}{\|a_1\|}$ is normalized.
- Assume the vectors $q_1, \dots, q_{i-1}$ are already orthonormal. Then, the vector $\bar{q}_i$ satisfies

$$\begin{aligned}\langle \bar{q}_i, q_j \rangle &= \langle a_i, q_j \rangle - \sum_{k=1}^{i-1} \langle q_k, a_i \rangle \langle q_k, q_j \rangle \\ &= \langle a_i, q_j \rangle - \sum_{k=1}^{i-1} \langle q_k, a_i \rangle \delta_{k,j} = 0\end{aligned}$$

for all $j \in \{1, \dots, i-1\}$, i.e., the vectors $q_1, \dots, q_i$ are orthonormal.

Gram-Schmidt Algorithm

The Gram-Schmidt Algorithm computes the vectors $q_1, \dots, q_k$. These vectors are orthonormal. This can be proven by induction:

- The vector $q_1 = \frac{a_1}{\|a_1\|}$ is normalized.
- Assume the vectors $q_1, \dots, q_{i-1}$ are already orthonormal. Then, the vector $\bar{q}_i$ satisfies

$$\begin{aligned}\langle \bar{q}_i, q_j \rangle &= \langle a_i, q_j \rangle - \sum_{k=1}^{i-1} \langle q_k, a_i \rangle \langle q_k, q_j \rangle \\ &= \langle a_i, q_j \rangle - \sum_{k=1}^{i-1} \langle q_k, a_i \rangle \delta_{k,j} = 0\end{aligned}$$

for all $j \in \{1, \dots, i-1\}$, i.e., the vectors $q_1, \dots, q_i$ are orthonormal.

Contents

- Problem Formulation
- Gram-Schmidt Algorithm
- **Orthogonal Polynomials**
- Solution of Gauss' Approximation Problem

Gram-Schmidt Algorithm for Functions

The Gram-Schmidt Algorithm can be generalized for any Hilbert space. In particular, we can apply it to functions, where $\langle \cdot, \cdot \rangle$ denotes the L_2 scalar product on the interval $[-1, 1]$.

Example: Start with $a_0(x) = 1$, $a_1(x) = x$, $a_2(x) = x^2$, ..., $a_n = x^n$:

$$\bullet q_0(x) = \sqrt{\frac{1}{2}}.$$

$$\bullet q_1(x) = \sqrt{\frac{3}{2}}x.$$

$$\bullet q_2(x) = \sqrt{\frac{5}{8}}(3x^2 - 1).$$

$$\bullet \dots$$

$$\bullet q_n(x) = \sqrt{\frac{2n+1}{2}} \frac{1}{2^n n!} \frac{\partial^n}{\partial x^n} (x^2 - 1)^n. \text{ (Exercise)}$$

Gram-Schmidt Algorithm for Functions

The Gram-Schmidt Algorithm can be generalized for any Hilbert space. In particular, we can apply it to functions, where $\langle \cdot, \cdot \rangle$ denotes the L_2 scalar product on the interval $[-1, 1]$.

Example: Start with $a_0(x) = 1$, $a_1(x) = x$, $a_2(x) = x^2$, ..., $a_n = x^n$:

- $q_0(x) = \sqrt{\frac{1}{2}}.$
- $q_1(x) = \sqrt{\frac{3}{2}}x.$
- $q_2(x) = \sqrt{\frac{5}{8}}(3x^2 - 1).$
- ...
- $q_n(x) = \sqrt{\frac{2n+1}{2}} \frac{1}{2^n n!} \frac{\partial^n}{\partial x^n} (x^2 - 1)^n.$ (Exercise)

Gram-Schmidt Algorithm for Functions

The Gram-Schmidt Algorithm can be generalized for any Hilbert space. In particular, we can apply it to functions, where $\langle \cdot, \cdot \rangle$ denotes the L_2 scalar product on the interval $[-1, 1]$.

Example: Start with $a_0(x) = 1$, $a_1(x) = x$, $a_2(x) = x^2$, ..., $a_n = x^n$:

- $q_0(x) = \sqrt{\frac{1}{2}}.$
- $q_1(x) = \sqrt{\frac{3}{2}}x.$
- $q_2(x) = \sqrt{\frac{5}{8}}(3x^2 - 1).$
- ...
- $q_n(x) = \sqrt{\frac{2n+1}{2}} \frac{1}{2^n n!} \frac{\partial^n}{\partial x^n} (x^2 - 1)^n.$ (Exercise)

Gram-Schmidt Algorithm for Functions

The Gram-Schmidt Algorithm can be generalized for any Hilbert space. In particular, we can apply it to functions, where $\langle \cdot, \cdot \rangle$ denotes the L_2 scalar product on the interval $[-1, 1]$.

Example: Start with $a_0(x) = 1$, $a_1(x) = x$, $a_2(x) = x^2$, ..., $a_n = x^n$:

- $q_0(x) = \sqrt{\frac{1}{2}}.$
- $q_1(x) = \sqrt{\frac{3}{2}}x.$
- $q_2(x) = \sqrt{\frac{5}{8}}(3x^2 - 1).$
- ...
- $q_n(x) = \sqrt{\frac{2n+1}{2}} \frac{1}{2^n n!} \frac{\partial^n}{\partial x^n} (x^2 - 1)^n.$ (Exercise)

Gram-Schmidt Algorithm for Functions

The Gram-Schmidt Algorithm can be generalized for any Hilbert space. In particular, we can apply it to functions, where $\langle \cdot, \cdot \rangle$ denotes the L_2 scalar product on the interval $[-1, 1]$.

Example: Start with $a_0(x) = 1$, $a_1(x) = x$, $a_2(x) = x^2$, ..., $a_n = x^n$:

- $q_0(x) = \sqrt{\frac{1}{2}}.$
- $q_1(x) = \sqrt{\frac{3}{2}}x.$
- $q_2(x) = \sqrt{\frac{5}{8}}(3x^2 - 1).$
- ...
- $q_n(x) = \sqrt{\frac{2n+1}{2}} \frac{1}{2^n n!} \frac{\partial^n}{\partial x^n} (x^2 - 1)^n.$ (Exercise)

Gram-Schmidt Algorithm for Functions

The Gram-Schmidt Algorithm can be generalized for any Hilbert space. In particular, we can apply it to functions, where $\langle \cdot, \cdot \rangle$ denotes the L_2 scalar product on the interval $[-1, 1]$.

Example: Start with $a_0(x) = 1$, $a_1(x) = x$, $a_2(x) = x^2$, ..., $a_n = x^n$:

- $q_0(x) = \sqrt{\frac{1}{2}}.$
- $q_1(x) = \sqrt{\frac{3}{2}}x.$
- $q_2(x) = \sqrt{\frac{5}{8}}(3x^2 - 1).$
- ...
- $q_n(x) = \sqrt{\frac{2n+1}{2}} \frac{1}{2^n n!} \frac{\partial^n}{\partial x^n} (x^2 - 1)^n.$ (Exercise)

Gram-Schmidt Algorithm for Functions

The Gram-Schmidt Algorithm can be generalized for any Hilbert space. In particular, we can apply it to functions, where $\langle \cdot, \cdot \rangle$ denotes the L_2 scalar product on the interval $[-1, 1]$.

Example: Start with $a_0(x) = 1$, $a_1(x) = x$, $a_2(x) = x^2$, ..., $a_n = x^n$:

- $q_0(x) = \sqrt{\frac{1}{2}}.$
- $q_1(x) = \sqrt{\frac{3}{2}}x.$
- $q_2(x) = \sqrt{\frac{5}{8}}(3x^2 - 1).$
- ...
- $q_n(x) = \sqrt{\frac{2n+1}{2}} \frac{1}{2^n n!} \frac{\partial^n}{\partial x^n} (x^2 - 1)^n.$ (Exercise)

Legendre Polynomials

The orthogonal polynomials

$$L_n(x) = \frac{1}{2^n n!} \frac{\partial^n}{\partial x^n} (x^2 - 1)^n.$$

are called Legendre polynomials. They satisfy

$$\langle L_i, L_j \rangle = \frac{2}{2i + 1} \delta_{i,j}$$

by construction.

Contents

- Problem Formulation
- Gram-Schmidt Algorithm
- Orthogonal Polynomials
- Solution of Gauss' Approximation Problem

Solution of Gauss' Approximation Problem

We represent the polynomial p with respect to orthonal basis functions

$q_0, \dots, q_n,$

$$p(x) = \sum_{i=0}^n c_i q_i(x) .$$

The coefficients $c_0, \dots, c_n$ can be found by substituting the orthogonal polynomials in the optimality condition

$$\forall q \in P_n, \quad \langle f - p, q \rangle = 0 .$$

This yields

$$c_i = \langle p, q_i \rangle = \langle f, q_i \rangle$$

for all $i \in \{1, \dots, n\}$.

Summary

- Gauss' approximation problem is to find polynomials $p \in P_n$, which solve

$$\min_{p \in P_n} \|f - p\|$$

for a given (L_2 -integrable) function f .

- Gram Schmidt algorithm can be used to construct orthogonal polynomials $q_0, \dots, q_n \in P_n$, which satisfy

$$\langle p_i, p_j \rangle = \delta_{i,j} .$$

- The solution polynomial p is unique and can be written in the form $p(x) = \sum_{i=0}^n c_i q_i(x)$. Here, the coefficients $c_0, \dots, c_n$ are given by

$$\forall i \in \{0, \dots, n\}, \quad c_i = \langle p, q_i \rangle = \langle f, q_i \rangle .$$

Chapter 7

Numerical Integration

Numerical Integration

- Problem Formulation
- Lagrange Quadrature
- Gauss Quadrature

Contents

- Problem Formulation
- Lagrange Quadrature
- Gauss Quadrature

Problem Formulation

We are interested in computing integrals of the form

$$\int_a^b f(x) \, dx$$

for given functions $f : \mathbb{R} \rightarrow \mathbb{R}$.

In this lecture, we assume (for simplicity) that f is sufficiently often differentiable.

Problem Formulation

We are interested in computing integrals of the form

$$\int_a^b f(x) \, dx$$

for given functions $f : \mathbb{R} \rightarrow \mathbb{R}$.

In this lecture, we assume (for simplicity) that f is sufficiently often differentiable.

Integration of Polynomials

For the special case that the function f is polynomial, the integral

$$\int_a^b f(x) \, dx = \int_a^b \sum_{i=0}^n c_i x^i \, dx$$

can be computed explicitly. We find

$$\int_a^b f(x) \, dx = \int_a^b \sum_{i=0}^n c_i x^i \, dx = \sum_{i=0}^n \frac{c_i}{i+1} (b^{i+1} - a^{i+1}) ,$$

which can be evaluated with Horner's algorithm.

Integration of Polynomials

For the special case that the function f is polynomial, the integral

$$\int_a^b f(x) \, dx = \int_a^b \sum_{i=0}^n c_i x^i \, dx$$

can be computed explicitly. We find

$$\int_a^b f(x) \, dx = \int_a^b \sum_{i=0}^n c_i x^i \, dx = \sum_{i=0}^n \frac{c_i}{i+1} (b^{i+1} - a^{i+1}) ,$$

which can be evaluated with Horner's algorithm.

Contents

- Problem Formulation
- Lagrange Quadrature
- Gauss Quadrature

Integration using Lagrange Interpolation

Since we know how to integrate polynomials, one strategy to approximate the general integral

$$\int_a^b f(x) \, dx$$

is to first approximate f by a polynomial and then use the integral over this polynomial as an approximation for the integral over f .

One way to do this is by Lagrange interpolation. For this aim, we choose points $x_0, \dots, x_n \in [a, b]$ and compute

$$p(x) = \sum_{i=0}^n f(x_i) L_i(x) = \sum_{i=0}^n f(x_i) \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}.$$

Integration using Lagrange Interpolation

Since we know how to integrate polynomials, one strategy to approximate the general integral

$$\int_a^b f(x) \, dx$$

is to first approximate f by a polynomial and then use the integral over this polynomial as an approximation for the integral over f .

One way to do this is by Lagrange interpolation. For this aim, we choose points $x_0, \dots, x_n \in [a, b]$ and compute

$$p(x) = \sum_{i=0}^n f(x_i) L_i(x) = \sum_{i=0}^n f(x_i) \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}.$$

Newton-Cotes Formulas

An important special case of Lagrange interpolation is obtained for equidistant points:

- Closed Newton-Cotes methods choose:

$$x_i = a + i * H, \quad i = 0, \dots, n, \quad \text{with} \quad H = \frac{b - a}{n} .$$

- Open Newton-Cotes methods choose:

$$x_i = a + (i + 1) * H, \quad i = 0, \dots, n, \quad \text{with} \quad H = \frac{b - a}{n + 2} .$$

Newton-Cotes Formulas

Using the coordinate transformation $x = a + tH$ with $t \in [0, n]$ we can compute the Lagrange polynomials

$$L_i(t) = \prod_{j=0, j \neq i}^n \frac{x - x_j}{x_i - x_j} = \prod_{j=0, j \neq i}^n \frac{t - j}{i - j}$$

The so-called Newton-Cotes coefficients

$$\alpha_i = \int_0^n \prod_{j=0, j \neq i}^n \frac{t - j}{i - j} dt$$

can be computed “once and forever”. The integral approximation is then given by

$$\int_a^b f(x) dx \approx \sum_{i=0}^n H \alpha_i f(x_i) .$$

Newton-Cotes Formulas

Using the coordinate transformation $x = a + tH$ with $t \in [0, n]$ we can compute the Lagrange polynomials

$$L_i(t) = \prod_{j=0, j \neq i}^n \frac{x - x_j}{x_i - x_j} = \prod_{j=0, j \neq i}^n \frac{t - j}{i - j}$$

The so-called Newton-Cotes coefficients

$$\alpha_i = \int_0^n \prod_{j=0, j \neq i}^n \frac{t - j}{i - j} dt$$

can be computed “once and forever”. The integral approximation is then given by

$$\int_a^b f(x) dx \approx \sum_{i=0}^n H \alpha_i f(x_i) .$$

Newton-Cotes Formulas

Using the coordinate transformation $x = a + tH$ with $t \in [0, n]$ we can compute the Lagrange polynomials

$$L_i(t) = \prod_{j=0, j \neq i}^n \frac{x - x_j}{x_i - x_j} = \prod_{j=0, j \neq i}^n \frac{t - j}{i - j}$$

The so-called Newton-Cotes coefficients

$$\alpha_i = \int_0^n \prod_{j=0, j \neq i}^n \frac{t - j}{i - j} dt$$

can be computed “once and forever”. The integral approximation is then given by

$$\int_a^b f(x) dx \approx \sum_{i=0}^n H \alpha_i f(x_i) .$$

Example: Simpson's rule

For $n = 2$ the coefficients of the closed Newton-Cotes methods are

- $\alpha_0 = \int_0^2 \frac{t-1}{0-1} \frac{t-2}{0-2} dt = \frac{1}{3}.$

- $\alpha_1 = \int_0^2 \frac{t-0}{1-0} \frac{t-2}{1-2} dt = \frac{4}{3}.$

- $\alpha_2 = \int_0^2 \frac{t-0}{2-0} \frac{t-1}{2-1} dt = \frac{1}{3}.$

- The corresponding approximation formula

$$\int_a^b f(x) dx \approx \frac{H}{3} \left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right)$$

with $H = \frac{b-a}{2}$ is called Simpson's rule.

Example: Simpson's rule

For $n = 2$ the coefficients of the closed Newton-Cotes methods are

- $\alpha_0 = \int_0^2 \frac{t-1}{0-1} \frac{t-2}{0-2} dt = \frac{1}{3}.$

- $\alpha_1 = \int_0^2 \frac{t-0}{1-0} \frac{t-2}{1-2} dt = \frac{4}{3}.$

- $\alpha_2 = \int_0^2 \frac{t-0}{2-0} \frac{t-1}{2-1} dt = \frac{1}{3}.$

- The corresponding approximation formula

$$\int_a^b f(x) dx \approx \frac{H}{3} \left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right)$$

with $H = \frac{b-a}{2}$ is called Simpson's rule.

Example: Simpson's rule

For $n = 2$ the coefficients of the closed Newton-Cotes methods are

- $\alpha_0 = \int_0^2 \frac{t-1}{0-1} \frac{t-2}{0-2} dt = \frac{1}{3}.$

- $\alpha_1 = \int_0^2 \frac{t-0}{1-0} \frac{t-2}{1-2} dt = \frac{4}{3}.$

- $\alpha_2 = \int_0^2 \frac{t-0}{2-0} \frac{t-1}{2-1} dt = \frac{1}{3}.$

- The corresponding approximation formula

$$\int_a^b f(x) dx \approx \frac{H}{3} \left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right)$$

with $H = \frac{b-a}{2}$ is called Simpson's rule.

Example: Simpson's rule

For $n = 2$ the coefficients of the closed Newton-Cotes methods are

- $\alpha_0 = \int_0^2 \frac{t-1}{0-1} \frac{t-2}{0-2} dt = \frac{1}{3}.$

- $\alpha_1 = \int_0^2 \frac{t-0}{1-0} \frac{t-2}{1-2} dt = \frac{4}{3}.$

- $\alpha_2 = \int_0^2 \frac{t-0}{2-0} \frac{t-1}{2-1} dt = \frac{1}{3}.$

- The corresponding approximation formula

$$\int_a^b f(x) dx \approx \frac{H}{3} \left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right)$$

with $H = \frac{b-a}{2}$ is called Simpson's rule.

Integration Error

We use the divided difference notation

$$\begin{aligned}f[x_i] &= f(x_i) \\f[x_i, x_{i+1}, \dots, x_{i+k}] &= \frac{f[x_{i+1}, x_{i+2}, \dots, x_{i+k}] - f[x_i, x_{i+1}, \dots, x_{i+k-1}]}{x_{i+k} - x_i},\end{aligned}$$

which is defined recursively. If $x_{i+k} = x_i$, replace the divided difference by the corresponding derivative of f , see Hermite interpolation.

The integration error is now given by

$$\int_a^b f(x) dx - \left(\frac{b-a}{n} \sum_{i=0}^n f(x_i) \alpha_i \right) = \int_a^b f[x_0, \dots, x_n, x] \prod_{j=0}^n (x - x_j) dx$$

Integration Error

We use the divided difference notation

$$\begin{aligned}f[x_i] &= f(x_i) \\f[x_i, x_{i+1}, \dots, x_{i+k}] &= \frac{f[x_{i+1}, x_{i+2}, \dots, x_{i+k}] - f[x_i, x_{i+1}, \dots, x_{i+k-1}]}{x_{i+k} - x_i},\end{aligned}$$

which is defined recursively. If $x_{i+k} = x_i$, replace the divided difference by the corresponding derivative of f , see Hermite interpolation.

The integration error is now given by

$$\int_a^b f(x) dx - \left(\frac{b-a}{n} \sum_{i=0}^n f(x_i) \alpha_i \right) = \int_a^b f[x_0, \dots, x_n, x] \prod_{j=0}^n (x - x_j) dx$$

Example

The error of Simpson's formula is bounded by

$$\begin{aligned}
 & \int_a^b f \left[a, \frac{a+b}{2}, b, x \right] (x-a) \left(x - \frac{a+b}{2} \right) (x-b) dx \\
 = & \int_a^b \frac{f \left[a, \frac{a+b}{2}, b, x \right] - f \left[a, \frac{a+b}{2}, b, \frac{a+b}{2} \right]}{x - \frac{a+b}{2}} \underbrace{\left(x-a \right) \left(x - \frac{a+b}{2} \right)^2 (x-b)}_{\leq 0} dx \\
 & + f \left[a, \frac{a+b}{2}, b, \frac{a+b}{2} \right] \underbrace{\int_a^b (x-a) \left(x - \frac{a+b}{2} \right) (x-b) dx}_{=0} \\
 \leq & \frac{\max_{x \in [a,b]} |f^{(4)}(x)|}{4!} \underbrace{\left| \int_a^b (x-a) \left(x - \frac{a+b}{2} \right)^2 (x-b) dx \right|}_{= \frac{(b-a)^5}{120}}
 \end{aligned}$$

Example

Summary: the numerical error of Simpson's formula is bounded by

$$\frac{\max_{x \in [a,b]} |f^{(4)}(x)|}{4!} \frac{(b-a)^5}{120} = \frac{(b-a)^5}{2880} \max_{x \in [a,b]} |f^{(4)}(x)|$$

The error bounds for other Newton-Cotes methods can be worked out analogously.

Contents

- Problem Formulation
- Lagrange Quadrature
- Gauss Quadrature

Order of Integration Formulas

Let f be a smooth function and $I(f) = \int_a^b f(x) \, dx$. We say that an integration formula of the form

$$I_n(f) = \sum_{i=0}^n \alpha_i f(x_i)$$

has order $q \in \mathbb{N}$ if there exists a constant $C < \infty$ such that

$$|I(f) - I_n(f)| \leq C \max_{\xi \in [a,b]} |f^{(q)}(\xi)| |b - a|^q$$

for any q -times differentiable function f .

Maximum Order of Integration Formulas

What is the maximum “order” of integration formulas of the form

$$I(f) = \int_a^b f(x) \, dx \approx I_n(f) = \sum_{i=0}^n \alpha_i f(x_i) ?$$

Can we choose the points $x_0, \dots, x_n \in [a, b]$ in a smart way?

An answer to this question was given by Gauss: the best order we can achieve is

$$|I(f) - I_n(f)| \leq \mathbf{O} \left((b-a)^{2n+2} \right) .$$

Maximum Order of Integration Formulas

What is the maximum “order” of integration formulas of the form

$$I(f) = \int_a^b f(x) \, dx \approx I_n(f) = \sum_{i=0}^n \alpha_i f(x_i) ?$$

Can we choose the points $x_0, \dots, x_n \in [a, b]$ in a smart way?

An answer to this question was given by Gauss: the best order we can achieve is

$$|I(f) - I_n(f)| \leq \mathbf{O} \left((b - a)^{2n+2} \right) .$$

Maximum Order of Integration Formulas

Let us first show that $m = 2n + 2$ is an upper bound on the order. For this aim, we consider the polynomial

$$p(x) = \frac{1}{b-a} \prod_{i=0}^n (x - x_i)^2 .$$

If we had a interpolation formula with order larger than $2n + 2$ it would be exact for $p(x)$, i.e.,

$$0 = I_n(p) = I(p) = \underbrace{\int_a^b p(x) \, dx}_{O((b-a)^{2n+2})} > 0 .$$

This is a contradiction! Thus, $m = 2n + 2$ is an upper bound.

Maximum Order of Integration Formulas

Let us first show that $m = 2n + 2$ is an upper bound on the order. For this aim, we consider the polynomial

$$p(x) = \frac{1}{b-a} \prod_{i=0}^n (x - x_i)^2 .$$

If we had a interpolation formula with order larger than $2n + 2$ it would be exact for $p(x)$, i.e.,

$$0 = I_n(p) = I(p) = \underbrace{\int_a^b p(x) \, dx}_{O((b-a)^{2n+2})} > 0 .$$

This is a contradiction! Thus, $m = 2n + 2$ is an upper bound.

Gauss Quadrature Rule: Main Idea

How can we construct an interpolation formula with order $2n + 2$?

In the following, we use the divided difference notation

$$f[x_0, \dots, x_n] = \sum_{i=0}^n f(x_i) \prod_{j=0, j \neq i}^n \frac{1}{x_i - x_j}.$$

The interpolation based integration formula for $2n + 2$ points,

$x_0, \dots, x_n, x_{n+1}, \dots, x_{2n+1}$ can now be written as

$$\begin{aligned} I_{2n+1}(f) &= \sum_{i=0}^{2n+1} f[x_0, \dots, x_i] \int_a^b \prod_{j=0}^{i-1} (x - x_j) \, dx \\ &= I_n(f) + \sum_{i=n+1}^{2n+1} f[x_0, \dots, x_i] \int_a^b \prod_{j=0}^{i-1} (x - x_j) \, dx \end{aligned}$$

Gauss Quadrature Rule: Main Idea

How can we construct an interpolation formula with order $2n + 2$?

In the following, we use the divided difference notation

$$f[x_0, \dots, x_n] = \sum_{i=0}^n f(x_i) \prod_{j=0, j \neq i}^n \frac{1}{x_i - x_j}.$$

The interpolation based integration formula for $2n + 2$ points,

$x_0, \dots, x_n, x_{n+1}, \dots, x_{2n+1}$ can now be written as

$$\begin{aligned} I_{2n+1}(f) &= \sum_{i=0}^{2n+1} f[x_0, \dots, x_i] \int_a^b \prod_{j=0}^{i-1} (x - x_j) dx \\ &= I_n(f) + \sum_{i=n+1}^{2n+1} f[x_0, \dots, x_i] \int_a^b \prod_{j=0}^{i-1} (x - x_j) dx \end{aligned}$$

Gauss Quadrature Rule: Main Idea

The integral term in the equation

$$I_{2n+1}(f) = I_n(f) + \sum_{i=n+1}^{2n+1} f[x_0, \dots, x_i] \int_a^b \prod_{j=0}^{i-1} (x - x_j) dx$$

can be written in the form

$$\int_a^b \prod_{j=0}^{i-1} (x - x_j) dx = \int_a^b \underbrace{\prod_{j=0}^n (x - x_j)}_{\in P_{n+1}} \underbrace{\prod_{j=n+1}^{i-1} (x - x_j)}_{\in P_n} dx .$$

Thus, if we succeed in choosing $x_0, \dots, x_n$ such that

$$\int_a^b \prod_{j=0}^n (x - x_j) q(x) dx = 0 \quad \text{for all } q \in P_n ,$$

we would have $I_{2n+1}(f) = I_n(f)$.

Gauss Quadrature Rule: Main Idea

The integral term in the equation

$$I_{2n+1}(f) = I_n(f) + \sum_{i=n+1}^{2n+1} f[x_0, \dots, x_i] \int_a^b \prod_{j=0}^{i-1} (x - x_j) dx$$

can be written in the form

$$\int_a^b \prod_{j=0}^{i-1} (x - x_j) dx = \int_a^b \underbrace{\prod_{j=0}^n (x - x_j)}_{\in P_{n+1}} \underbrace{\prod_{j=n+1}^{i-1} (x - x_j)}_{\in P_n} dx .$$

Thus, if we succeed in choosing $x_0, \dots, x_n$ such that

$$\int_a^b \prod_{j=0}^n (x - x_j) q(x) dx = 0 \quad \text{for all } q \in P_n ,$$

we would have $I_{2n+1}(f) = I_n(f)$.

Gauss Quadrature Rule: Main Idea

Let us assume $a = -1$ and $b = 1$. The main idea is to choose the points $x_0, x_1, \dots, x_n$ such that we have

$$L_{n+1}(x_i) = 0$$

with L_{n+1} being the $(n + 1)$ -th Legendre polynomial we must have

$$\int_{-1}^1 \underbrace{\prod_{j=0}^n (x - x_j)}_{\sim L_{n+1}(x)} q(x) \, dx = 0 \quad \text{for all } q \in P_n ,$$

since L_{n+1} is by construction orthogonal on P_n .

Roots of the Legendre Polynomials

Theorem:

The Legendre polynomial L_{n+1} has $n + 1$ distinct real roots on the interval $[-1, 1]$.

Proof: We define the set

$$S = \{\lambda \in (-1, 1) \mid \lambda \text{ is a real root of } L_{n+1} \text{ with odd multiplicity}\}$$

and the polynomial $q(x) = \prod_{\lambda \in S} (x - \lambda)$. Now, the polynomial $q(x) \cdot L_{n+1}(x)$ must be either positive or negative; that is,

$$\langle q, L_{n+1} \rangle \neq 0 .$$

For $|S| < n + 1$ this is a contradiction to $q \perp L_{n+1}$.

Roots of the Legendre Polynomials

Theorem:

The Legendre polynomial L_{n+1} has $n + 1$ distinct real roots on the interval $[-1, 1]$.

Proof: We define the set

$$S = \{\lambda \in (-1, 1) \mid \lambda \text{ is a real root of } L_{n+1} \text{ with odd multiplicity}\}$$

and the polynomial $q(x) = \prod_{\lambda \in S} (x - \lambda)$. Now, the polynomial $q(x) \cdot L_{n+1}(x)$ must be either positive or negative; that is,

$$\langle q, L_{n+1} \rangle \neq 0 .$$

For $|S| < n + 1$ this is a contradiction to $q \perp L_{n+1}$.

Roots of the Legendre Polynomials

Theorem:

The Legendre polynomial L_{n+1} has $n + 1$ distinct real roots on the interval $[-1, 1]$.

Proof: We define the set

$$S = \{\lambda \in (-1, 1) \mid \lambda \text{ is a real root of } L_{n+1} \text{ with odd multiplicity}\}$$

and the polynomial $q(x) = \prod_{\lambda \in S} (x - \lambda)$. Now, the polynomial $q(x) \cdot L_{n+1}(x)$ must be either positive or negative; that is,

$$\langle q, L_{n+1} \rangle \neq 0 .$$

For $|S| < n + 1$ this is a contradiction to $q \perp L_{n+1}$.

Gauss Quadrature

If $x_0, \dots, x_n$ are the $n + 1$ roots of the Legendre polynomial L_{n+1} on the interval $[a, b]$, then the corresponding quadrature formula

$$I_n(f) = \sum_{i=0}^n \alpha_i f_i(x_i)$$

with $\alpha_i = \int_a^b \prod_{j=0, j \neq i}^n \frac{x - x_j}{x_i - x_j} dt$ has order $2n + 2$, i.e.,

$$|I(f) - I_n(f)| \leq \mathbf{O}((a - b)^{2n+2}) .$$

Example

For the case $n = 1$, the Legendre polynomial $L_2(x) = \frac{1}{2}(3x^2 - 1)$ has the roots

$$x_{1,2} = \pm \sqrt{\frac{1}{3}}$$

Thus, the first Gauss quadrature formula is given by

$$\int_{-1}^1 f(x) \, dx \approx f\left(-\sqrt{\frac{1}{3}}\right) + f\left(\sqrt{\frac{1}{3}}\right),$$

which is exact for polynomials of order less or equal than $2n + 1 = 3$.

Summary

- The main idea of numerical integration is to first approximate the function f with a polynomial p and then integrate the polynomial.
- For equidistant interpolation points, we obtain the so-called Newton Codes formulas. The coefficients $\alpha_i = \int_0^n \prod_{j=0, j \neq i}^n \frac{t-j}{i-j} dt$ can be worked out “once and forever”.
- The maximum order of polynomial interpolation based integration schemes is $2n + 2$ (for $n + 1$ evaluation points). This order can be achieved by using Gauss quadrature rules.

Chapter 8

Newton's Method

Newton's Method

- Problem Formulation
- Newton's method
- Local Convergence Analysis
- Unconstrained Optimization
- Globalization Techniques

Contents

- Problem Formulation
- Newton's method
- Local Convergence Analysis
- Unconstrained Optimization
- Globalization Techniques

Problem Formulation

Given a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ we are searching for solutions of the nonlinear equation

$$f(x) = 0 .$$

Examples:

- For $f(x) = Ax - b$ this amounts to solving a linear equation system,
- For $f(x) = x^2 + 1$: no solution can be found,
- For $f(x) = x^3 - x$: three solutions exist.

Problem Formulation

Given a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ we are searching for solutions of the nonlinear equation

$$f(x) = 0 .$$

Examples:

- For $f(x) = Ax - b$ this amounts to solving a linear equation system,
- For $f(x) = x^2 + 1$: no solution can be found,
- For $f(x) = x^3 - x$: three solutions exist.

Problem Formulation

Given a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ we are searching for solutions of the nonlinear equation

$$f(x) = 0 .$$

Examples:

- For $f(x) = Ax - b$ this amounts to solving a linear equation system,
- For $f(x) = x^2 + 1$: no solution can be found,
- For $f(x) = x^3 - x$: three solutions exist.

Problem Formulation

Given a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ we are searching for solutions of the nonlinear equation

$$f(x) = 0 .$$

Examples:

- For $f(x) = Ax - b$ this amounts to solving a linear equation system,
- For $f(x) = x^2 + 1$: no solution can be found,
- For $f(x) = x^3 - x$: three solutions exist.

Contents

- Problem Formulation
- **Newton's method**
- Local Convergence Analysis
- Unconstrained Optimization
- Globalization Techniques

Main Idea

In order to solve the nonlinear equation $f(x)$, we start with an initial guess x_0 and solve the linear equation systems

$$f(x_k) + M(x_k)(x_{k+1} - x_k) = 0 ,$$

for $k \in \{0, 1, 2, \dots\}$. Here, the matrix $M(x_k) \in \mathbb{R}^{n \times n}$ is chosen in such a way that

$$f(x_k) + M(x_k)(x - x_k) \approx f(x) ,$$

is an approximation of the function f . For example, if f is differentiable, we might choose $M(x_k) = f'(x_k)$, which corresponds to the so called Newton method.

Main Idea

In order to solve the nonlinear equation $f(x)$, we start with an initial guess x_0 and solve the linear equation systems

$$f(x_k) + M(x_k)(x_{k+1} - x_k) = 0 ,$$

for $k \in \{0, 1, 2, \dots\}$. Here, the matrix $M(x_k) \in \mathbb{R}^{n \times n}$ is chosen in such a way that

$$f(x_k) + M(x_k)(x - x_k) \approx f(x) ,$$

is an approximation of the function f . For example, if f is differentiable, we might choose $M(x_k) = f'(x_k)$, which corresponds to the so called Newton method.

Main Idea

In order to solve the nonlinear equation $f(x)$, we start with an initial guess x_0 and solve the linear equation systems

$$f(x_k) + M(x_k)(x_{k+1} - x_k) = 0 ,$$

for $k \in \{0, 1, 2, \dots\}$. Here, the matrix $M(x_k) \in \mathbb{R}^{n \times n}$ is chosen in such a way that

$$f(x_k) + M(x_k)(x - x_k) \approx f(x) ,$$

is an approximation of the function f . For example, if f is differentiable, we might choose $M(x_k) = f'(x_k)$, which corresponds to the so called Newton method.

Newton (or Newton-type) method

If the matrix $M(x_k)$ is invertible, the method can also be written in the form

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

for $k \in \{0, 1, 2, \dots\}$.

- In practice, we usually work with approximations $M(x_k) \approx f'(x_k)$.
- If $M(x_k)$ is independent of x_k , we only need to decompose M once (e.g., using LR or QR decomposition)
- Some methods try to update M at every step without re-computing the Jacobian.

Newton (or Newton-type) method

If the matrix $M(x_k)$ is invertible, the method can also be written in the form

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

for $k \in \{0, 1, 2, \dots\}$.

- In practice, we usually work with approximations $M(x_k) \approx f'(x_k)$.
- If $M(x_k)$ is independent of x_k , we only need to decompose M once (e.g., using LR or QR decomposition)
- Some methods try to update M at every step without re-computing the Jacobian.

Newton (or Newton-type) method

If the matrix $M(x_k)$ is invertible, the method can also be written in the form

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

for $k \in \{0, 1, 2, \dots\}$.

- In practice, we usually work with approximations $M(x_k) \approx f'(x_k)$.
- If $M(x_k)$ is independent of x_k , we only need to decompose M once (e.g., using LR or QR decomposition)
- Some methods try to update M at every step without re-computing the Jacobian.

Newton (or Newton-type) method

If the matrix $M(x_k)$ is invertible, the method can also be written in the form

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

for $k \in \{0, 1, 2, \dots\}$.

- In practice, we usually work with approximations $M(x_k) \approx f'(x_k)$.
- If $M(x_k)$ is independent of x_k , we only need to decompose M once (e.g., using LR or QR decomposition)
- Some methods try to update M at every step without re-computing the Jacobian.

Scaling Properties

If x^* satisfies $f(x^*) = 0$ it also satisfies $S \cdot f(x^*) = 0$, where $S \in \mathbb{R}^{n \times n}$ can be any (invertible) scaling matrix. If we apply the above recursion to the scaled equation

$$\tilde{f}(x) = S \cdot f(x) = 0$$

we obtain the iterates $x_{k+1} = x_k - M(x_k)^{-1} S \cdot f(x_k)$, which do in general not coincide with the iterates that are obtained without scaling f . However, if we use exact Jacobians, we have

$$M(x_k) = \tilde{f}'(x) = S \cdot f'(x) \quad \implies \quad x_{k+1} = x_k - f'(x_k)^{-1} f(x_k) .$$

This implies that Newton's methods with exact Jacobians is invariant under scaling.

Scaling Properties

If x^* satisfies $f(x^*) = 0$ it also satisfies $S \cdot f(x^*) = 0$, where $S \in \mathbb{R}^{n \times n}$ can be any (invertible) scaling matrix. If we apply the above recursion to the scaled equation

$$\tilde{f}(x) = S \cdot f(x) = 0$$

we obtain the iterates $x_{k+1} = x_k - M(x_k)^{-1} S \cdot f(x_k)$, which do in general not coincide with the iterates that are obtained without scaling f . However, if we use exact Jacobians, we have

$$M(x_k) = \tilde{f}'(x) = S \cdot f'(x) \quad \implies \quad x_{k+1} = x_k - f'(x_k)^{-1} f(x_k) .$$

This implies that Newton's methods with exact Jacobians is invariant under scaling.

Scaling Properties

If x^* satisfies $f(x^*) = 0$ it also satisfies $S \cdot f(x^*) = 0$, where $S \in \mathbb{R}^{n \times n}$ can be any (invertible) scaling matrix. If we apply the above recursion to the scaled equation

$$\tilde{f}(x) = S \cdot f(x) = 0$$

we obtain the iterates $x_{k+1} = x_k - M(x_k)^{-1} S \cdot f(x_k)$, which do in general not coincide with the iterates that are obtained without scaling f . However, if we use exact Jacobians, we have

$$M(x_k) = \tilde{f}'(x) = S \cdot f'(x) \quad \implies \quad x_{k+1} = x_k - f'(x_k)^{-1} f(x_k) .$$

This implies that Newton's methods with exact Jacobians is invariant under scaling.

Contents

- Problem Formulation
- Newton's method
- **Local Convergence Analysis**
- Unconstrained Optimization
- Globalization Techniques

Local Convergence Analysis

Assumptions:

- There exists a point x^* with $f(x^*) = 0$.
- The point x_0 is already in a small neighborhood of x^* .
- The scaled Jacobian matrix $M(x_k)^{-1}f'(x)$ is Lipschitz continuous w.r.t. x in a neighborhood of x^* with Lipschitz constant $\omega \geq 0$.

The basic idea is to estimate the distance of the iterates to x^* :

$$\begin{aligned} & \|x_{k+1} - x^*\| \\ &= \|x_k - x^* - M(x_k)^{-1}f(x_k)\| \\ &= \left\| x_k - x^* - M(x_k)^{-1} \int_0^1 J(x^* + s(x_k - x^*))(x_k - x^*) ds \right\| \\ &\leq \|x_k - x^* - M(x_k)^{-1}J(x_k)(x_k - x^*)\| + \frac{\omega}{2} \|x_k - x^*\|_2^2. \end{aligned}$$

Local Convergence Analysis

Assumptions:

- There exists a point x^* with $f(x^*) = 0$.
- The point x_0 is already in a small neighborhood of x^* .
- The scaled Jacobian matrix $M(x_k)^{-1}f'(x)$ is Lipschitz continuous w.r.t. x in a neighborhood of x^* with Lipschitz constant $\omega \geq 0$.

The basic idea is to estimate the distance of the iterates to x^* :

$$\begin{aligned} & \|x_{k+1} - x^*\| \\ &= \|x_k - x^* - M(x_k)^{-1}f(x_k)\| \\ &= \left\| x_k - x^* - M(x_k)^{-1} \int_0^1 J(x^* + s(x_k - x^*))(x_k - x^*) ds \right\| \\ &\leq \|x_k - x^* - M(x_k)^{-1}J(x_k)(x_k - x^*)\| + \frac{\omega}{2} \|x_k - x^*\|_2^2. \end{aligned}$$

Local Convergence Analysis

In summary, we find the estimate

$$\|x_{k+1} - x^*\| \leq \kappa \|x_k - x^*\| + \frac{\omega}{2} \|x_k - x^*\|_2^2 .$$

as long as $\|I - M(x_k)^{-1}J(x_k)\| \leq \kappa$. Here, κ can be interpreted as a bound on the accuracy of the Jacobian approximation M . If we have $\kappa < 1$ and $\|x_0 - x^*\| < \frac{2}{\omega}(1 - \kappa)$ the iterates contract and we have

$$\lim_{k \rightarrow \infty} x_k \rightarrow x^* .$$

Local Convergence Analysis

In summary, we find the estimate

$$\|x_{k+1} - x^*\| \leq \kappa \|x_k - x^*\| + \frac{\omega}{2} \|x_k - x^*\|_2^2 .$$

as long as $\|I - M(x_k)^{-1}J(x_k)\| \leq \kappa$. Here, κ can be interpreted as a bound on the accuracy of the Jacobian approximation M . If we have $\kappa < 1$ and $\|x_0 - x^*\| < \frac{2}{\omega}(1 - \kappa)$ the iterates contract and we have

$$\lim_{k \rightarrow \infty} x_k \rightarrow x^* .$$

Local Convergence Analysis

In summary, we find the estimate

$$\|x_{k+1} - x^*\| \leq \kappa \|x_k - x^*\| + \frac{\omega}{2} \|x_k - x^*\|_2^2 .$$

as long as $\|I - M(x_k)^{-1}J(x_k)\| \leq \kappa$. Here, κ can be interpreted as a bound on the accuracy of the Jacobian approximation M . If we have $\kappa < 1$ and $\|x_0 - x^*\| < \frac{2}{\omega}(1 - \kappa)$ the iterates contract and we have

$$\lim_{k \rightarrow \infty} x_k \rightarrow x^* .$$

Convergence Rate

The convergence rate estimate

$$\|x_{k+1} - x^*\| \leq \kappa \|x_k - x^*\| + \frac{\omega}{2} \|x_k - x^*\|_2^2 .$$

implies that

- if we have $\kappa \neq 0$ the convergence rate is in general linear.
- if we choose $M(x_k) = J(x_k)$ (Newton's method), we have $\kappa = 0$ and

$$\|x_{k+1} - x^*\| \leq \frac{\omega}{2} \|x_k - x^*\|_2^2 .$$

In this case, the convergence rate is called quadratic. (the number of correct internal decimal places roughly doubles in every step).

Convergence Rate

The convergence rate estimate

$$\|x_{k+1} - x^*\| \leq \kappa \|x_k - x^*\| + \frac{\omega}{2} \|x_k - x^*\|_2^2 .$$

implies that

- if we have $\kappa \neq 0$ the convergence rate is in general linear.
- if we choose $M(x_k) = J(x_k)$ (Newton's method), we have $\kappa = 0$ and

$$\|x_{k+1} - x^*\| \leq \frac{\omega}{2} \|x_k - x^*\|_2^2 .$$

In this case, the convergence rate is called quadratic. (the number of correct internal decimal places roughly doubles in every step).

Convergence Rate

The convergence rate estimate

$$\|x_{k+1} - x^*\| \leq \kappa \|x_k - x^*\| + \frac{\omega}{2} \|x_k - x^*\|_2^2 .$$

implies that

- if we have $\kappa \neq 0$ the convergence rate is in general linear.
- if we choose $M(x_k) = J(x_k)$ (Newton's method), we have $\kappa = 0$ and

$$\|x_{k+1} - x^*\| \leq \frac{\omega}{2} \|x_k - x^*\|_2^2 .$$

In this case, the convergence rate is called quadratic. (the number of correct internal decimal places roughly doubles in every step).

Degeneracy Handling

If the exact Jacobian $J(x^*)$ is singular (has eigenvalues that are equal to zero), Newton's method is not applicable, since the matrices $J(x_k)$ converge to a singular matrix that cannot be inverted.

If the matrix $M(x_k)$ is chosen in such a way that the convergence condition

$$\|I - M(x_k)^{-1}J(x_k)\| < 1$$

is maintained. This is possible even if J is singular, although special care has to be taken, if M is ill-conditioned. If we choose M such that

$$\|I - M(x_k)^{-1}J(x_k)\| \leq \mathbf{O}(\|x_k - x^*\|) ,$$

a (locally) quadratic convergence rate can be recovered.

Degeneracy Handling

If the exact Jacobian $J(x^*)$ is singular (has eigenvalues that are equal to zero), Newton's method is not applicable, since the matrices $J(x_k)$ converge to a singular matrix that cannot be inverted.

If the matrix $M(x_k)$ is chosen in such a way that the convergence condition

$$\|I - M(x_k)^{-1}J(x_k)\| < 1$$

is maintained. This is possible even if J is singular, although special care has to be taken, if M is ill-conditioned. If we choose M such that

$$\|I - M(x_k)^{-1}J(x_k)\| \leq \mathbf{O}(\|x_k - x^*\|) ,$$

a (locally) quadratic convergence rate can be recovered.

Degeneracy Handling

If the exact Jacobian $J(x^*)$ is singular (has eigenvalues that are equal to zero), Newton's method is not applicable, since the matrices $J(x_k)$ converge to a singular matrix that cannot be inverted.

If the matrix $M(x_k)$ is chosen in such a way that the convergence condition

$$\|I - M(x_k)^{-1}J(x_k)\| < 1$$

is maintained. This is possible even if J is singular, although special care has to be taken, if M is ill-conditioned. If we choose M such that

$$\|I - M(x_k)^{-1}J(x_k)\| \leq \mathbf{O}(\|x_k - x^*\|) ,$$

a (locally) quadratic convergence rate can be recovered.

Contents

- Problem Formulation
- Newton's method
- Local Convergence Analysis
- **Unconstrained Optimization**
- Globalization Techniques

Nonlinear Least-Squares Problems

The unconstrained nonlinear least-squares problem is given by

$$\min_x \|f(x)\|_2^2 .$$

- If we can find a $x^* \in \mathbb{R}^n$ with $f(x^*) = 0$, then x^* is minimizer of the above problem.
- For $f(x) = Ax - b$ this problem is a least-squares problem in standard form.
- Makes sense for any function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ with $m \neq n$ in general.

Nonlinear Least-Squares Problems

The unconstrained nonlinear least-squares problem is given by

$$\min_x \|f(x)\|_2^2 .$$

- If we can find a $x^* \in \mathbb{R}^n$ with $f(x^*) = 0$, then x^* is minimizer of the above problem.
- For $f(x) = Ax - b$ this problem is a least-squares problem in standard form.
- Makes sense for any function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ with $m \neq n$ in general.

Nonlinear Least-Squares Problems

The unconstrained nonlinear least-squares problem is given by

$$\min_x \|f(x)\|_2^2 .$$

- If we can find a $x^* \in \mathbb{R}^n$ with $f(x^*) = 0$, then x^* is minimizer of the above problem.
- For $f(x) = Ax - b$ this problem is a least-squares problem in standard form.
- Makes sense for any function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ with $m \neq n$ in general.

Nonlinear Least-Squares Problems

The unconstrained nonlinear least-squares problem is given by

$$\min_x \|f(x)\|_2^2 .$$

- If we can find a $x^* \in \mathbb{R}^n$ with $f(x^*) = 0$, then x^* is minimizer of the above problem.
- For $f(x) = Ax - b$ this problem is a least-squares problem in standard form.
- Makes sense for any function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ with $m \neq n$ in general.

Unconstrained Optimization Problems

An even more general class of problems are the unconstrained optimization problems

$$\min_x F(x) .$$

This contains the nonlinear least-squares problems as a special case, since we can choose $F(x) = \|f(x)\|_2^2$.

If F is twice Lipschitz-continuously differentiable, a minimizer can be found by applying Newton's method to

$$F'(x) = 0$$

If a solution x^* satisfies $F''(x) \succ 0$, it must be a local minimizer.

Unconstrained Optimization Problems

An even more general class of problems are the unconstrained optimization problems

$$\min_x F(x) .$$

This contains the nonlinear least-squares problems as a special case, since we can choose $F(x) = \|f(x)\|_2^2$.

If F is twice Lipschitz-continuously differentiable, a minimizer can be found by applying Newton's method to

$$F'(x) = 0$$

If a solution x^* satisfies $F''(x) \succ 0$, it must be a local minimizer.

Unconstrained Optimization Problems

An even more general class of problems are the unconstrained optimization problems

$$\min_x F(x) .$$

This contains the nonlinear least-squares problems as a special case, since we can choose $F(x) = \|f(x)\|_2^2$.

If F is twice Lipschitz-continuously differentiable, a minimizer can be found by applying Newton's method to

$$F'(x) = 0$$

If a solution x^* satisfies $F''(x) \succ 0$, it must be a local minimizer.

Newton-Type Methods for Optimization

In detail, Newton-type methods for unconstrained optimization problems can be written in the form

$$x_{k+1} = x_k - M(x_k)^{-1} F'(x_k)^T ,$$

where $M(x_k) \approx F''(x_k)$ is a suitable Hessian approximation.

- In practice, we often choose a symmetric Hessian approximation M , since F'' is symmetric.
- If $M(x_k)$ is symmetric and positive definite, the iterate x_{k+1} is the minimizer of the quadratic function

$$\min_{x_{k+1}} F(x_k) + F'(x_k)(x_{k+1} - x_k) + \frac{1}{2} (x_{k+1} - x_k)^T M(x_k) (x_{k+1} - x_k) ,$$

which can be interpreted as a quadratic model of F .

Newton-Type Methods for Optimization

In detail, Newton-type methods for unconstrained optimization problems can be written in the form

$$x_{k+1} = x_k - M(x_k)^{-1} F'(x_k)^T ,$$

where $M(x_k) \approx F''(x_k)$ is a suitable Hessian approximation.

- In practice, we often choose a symmetric Hessian approximation M , since F'' is symmetric.
- If $M(x_k)$ is symmetric and positive definite, the iterate x_{k+1} is the minimizer of the quadratic function

$$\min_{x_{k+1}} F(x_k) + F'(x_k)(x_{k+1} - x_k) + \frac{1}{2} (x_{k+1} - x_k)^T M(x_k) (x_{k+1} - x_k) ,$$

which can be interpreted as a quadratic model of F .

Newton-Type Methods for Optimization

In detail, Newton-type methods for unconstrained optimization problems can be written in the form

$$x_{k+1} = x_k - M(x_k)^{-1} F'(x_k)^T ,$$

where $M(x_k) \approx F''(x_k)$ is a suitable Hessian approximation.

- In practice, we often choose a symmetric Hessian approximation M , since F'' is symmetric.
- If $M(x_k)$ is symmetric and positive definite, the iterate x_{k+1} is the minimizer of the quadratic function

$$\min_{x_{k+1}} F(x_k) + F'(x_k)(x_{k+1} - x_k) + \frac{1}{2} (x_{k+1} - x_k)^T M(x_k) (x_{k+1} - x_k) ,$$

which can be interpreted as a quadratic model of F .

Contents

- Problem Formulation
- Newton's method
- Local Convergence Analysis
- Unconstrained Optimization
- Globalization Techniques

Line Search Methods

So far, we have only analyzed the local convergence properties of Newton-type methods. If we start far from a local solution, Newton type methods are often take “too big” steps and are divergent.

One way to fix this problem is to first compute a step-direction by solving

$$\min_{x_{k+1}} F(x_k) + F'(x_k)\Delta x_k + \frac{1}{2}\Delta x_k^T M(x_k)\Delta x_k ,$$

and update the iterate as

$$x_{k+1} = x_k + \alpha_k \Delta x_k .$$

Here, $\alpha_k \in (0, 1]$ is a so-called line-search parameter, which is found by (approximately) solving the scalar optimization problem

$$\min_{\alpha_k \in [0,1]} F(x_k + \alpha_k \Delta x_k) .$$

Line Search Methods

So far, we have only analyzed the local convergence properties of Newton-type methods. If we start far from a local solution, Newton type methods are often take “too big” steps and are divergent.

One way to fix this problem is to first compute a step-direction by solving

$$\min_{x_{k+1}} F(x_k) + F'(x_k)\Delta x_k + \frac{1}{2}\Delta x_k^T M(x_k)\Delta x_k ,$$

and update the iterate as

$$x_{k+1} = x_k + \alpha_k \Delta x_k .$$

Here, $\alpha_k \in (0, 1]$ is a so-called line-search parameter, which is found by (approximately) solving the scalar optimization problem

$$\min_{\alpha_k \in [0, 1]} F(x_k + \alpha_k \Delta x_k) .$$

Line Search Methods

So far, we have only analyzed the local convergence properties of Newton-type methods. If we start far from a local solution, Newton type methods are often take “too big” steps and are divergent.

One way to fix this problem is to first compute a step-direction by solving

$$\min_{x_{k+1}} F(x_k) + F'(x_k)\Delta x_k + \frac{1}{2}\Delta x_k^T M(x_k)\Delta x_k ,$$

and update the iterate as

$$x_{k+1} = x_k + \alpha_k \Delta x_k .$$

Here, $\alpha_k \in (0, 1]$ is a so-called line-search parameter, which is found by (approximately) solving the scalar optimization problem

$$\min_{\alpha_k \in [0,1]} F(x_k + \alpha_k \Delta x_k) .$$

Armijo Line Search Conditions

In practice the line search optimization problem

$$\min_{\alpha_k \in [0,1]} F(x_k + \alpha_k \Delta x_k) .$$

is not solved exactly (too expensive), but only approximately.

One way to implement this is by using back-tracking until the Armijo condition

$$F(x_k + \alpha_k \Delta x_k) \leq F(x_k) + c\alpha_k F'(x_k) \Delta x_k$$

for a constant $c \ll 1$ is satisfied. This condition ensures that the line search parameter is not excessively large, although it is not sufficient to prove convergence in general.

Armijo Line Search Conditions

In practice the line search optimization problem

$$\min_{\alpha_k \in [0,1]} F(x_k + \alpha_k \Delta x_k) .$$

is not solved exactly (too expensive), but only approximately.

One way to implement this is by using back-tracking until the Armijo condition

$$F(x_k + \alpha_k \Delta x_k) \leq F(x_k) + c\alpha_k F'(x_k) \Delta x_k$$

for a constant $c \ll 1$ is satisfied. This condition ensures that the line search parameter is not excessively large, although it is not sufficient to prove convergence in general.

Armijo Line Search Conditions

If we substitute $\Delta x_k = -M(x_k)^{-1}F'(x_k)^T$ the Armijo line search condition can alternatively be written in the form

$$F(x_k + \alpha_k \Delta x_k) \leq F(x_k) - c\alpha_k F'(x_k)M(x_k)^{-1}F'(x_k)^T .$$

Thus, if M is positive definite, the Armijo condition ensures that we get a strict descent of the objective function whenever we apply a (damped) Newton step.

Positive definite approximations M in combination with Armijo line search work extremely well in practice, but other variants exist.

Armijo Line Search Conditions

If we substitute $\Delta x_k = -M(x_k)^{-1}F'(x_k)^T$ the Armijo line search condition can alternatively be written in the form

$$F(x_k + \alpha_k \Delta x_k) \leq F(x_k) - c\alpha_k F'(x_k)M(x_k)^{-1}F'(x_k)^T .$$

Thus, if M is positive definite, the Armijo condition ensures that we get a strict descent of the objective function whenever we apply a (damped) Newton step.

Positive definite approximations M in combination with Armijo line search work extremely well in practice, but other variants exist.

Chapter 9

Gauss-Newton Methods

Gauss-Newton Methods

- Problem Formulation
- Gauss-Newton Methods
- Local Convergence Analysis

Contents

- Problem Formulation
- Gauss-Newton Methods
- Local Convergence Analysis

Problem Formulation

Given a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ we are searching for solutions of the nonlinear least-squares problem

$$\min_x \frac{1}{2} \|f(x)\|_2^2 .$$

Examples:

- For $f(x) = Ax - b$ this amounts to solving a standard least-squares problem,
- If we want to estimate the parameters x of a nonlinear model $h : \mathbb{R}^n \rightarrow \mathbb{R}^m$ from given measurements, we often set

$$f(x) = h(x) - \eta .$$

Problem Formulation

Given a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ we are searching for solutions of the nonlinear least-squares problem

$$\min_x \frac{1}{2} \|f(x)\|_2^2 .$$

Examples:

- For $f(x) = Ax - b$ this amounts to solving a standard least-squares problem,
- If we want to estimate the parameters x of a nonlinear model $h : \mathbb{R}^n \rightarrow \mathbb{R}^m$ from given measurements, we often set

$$f(x) = h(x) - \eta .$$

Problem Formulation

Given a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ we are searching for solutions of the nonlinear least-squares problem

$$\min_x \frac{1}{2} \|f(x)\|_2^2 .$$

Examples:

- For $f(x) = Ax - b$ this amounts to solving a standard least-squares problem,
- If we want to estimate the parameters x of a nonlinear model $h : \mathbb{R}^n \rightarrow \mathbb{R}^m$ from given measurements, we often set

$$f(x) = h(x) - \eta .$$

Contents

- Problem Formulation
- Gauss-Newton Methods
- Local Convergence Analysis

Gauss-Newton method

In order to solve the nonlinear least-squares optimization problem $f(x)$, we start with an initial guess x_0 and solve the standard least-squares problem

$$\min_{\Delta x_k} \frac{1}{2} \|f(x_k) + f'(x_k)\Delta x_k\|_2^2 ,$$

and update $x_{k+1} = x_k + \alpha_k \Delta x_k$ for $k \in \{0, 1, 2, \dots\}$, where $\alpha_k \in (0, 1]$ is a line search parameter.

If the Jacobian matrix $f'(x_k)$ has full-rank, the step direction can alternatively be written in the form

$$\Delta x_k = - \left(f'(x_k)^T f'(x_k) \right)^{-1} f'(x_k)^T f(x_k) .$$

Gauss-Newton method

In order to solve the nonlinear least-squares optimization problem $f(x)$, we start with an initial guess x_0 and solve the standard least-squares problem

$$\min_{\Delta x_k} \frac{1}{2} \|f(x_k) + f'(x_k)\Delta x_k\|_2^2 ,$$

and update $x_{k+1} = x_k + \alpha_k \Delta x_k$ for $k \in \{0, 1, 2, \dots\}$, where $\alpha_k \in (0, 1]$ is a line search parameter.

If the Jacobian matrix $f'(x_k)$ has full-rank, the step direction can alternatively be written in the form

$$\Delta x_k = - \left(f'(x_k)^T f'(x_k) \right)^{-1} f'(x_k)^T f(x_k) .$$

Interpretation as Newton-type methods

If we apply a Newton-type method to solve the minimization problem

$$\min_x F(x) \quad \text{with} \quad F(x) = \frac{1}{2} \|f(x)\|_2^2$$

we obtain a step direction of the form

$$\Delta x_k = -M(x_k)^{-1} F'(x_k)^T = -M(x_k)^{-1} f'(x_k)^T f(x_k) .$$

Thus, Gauss-Newton methods are special class of Newton type methods which employ the Hessian approximation

$$F''(x_k) \approx M(x_k) = f'(x_k)^T f'(x_k) .$$

Properties of Gauss-Newton Hessian approximations

The exact Hessian matrix is given by

$$F''(x_k) = f''(x_k)^T f(x_k) + f'(x_k)^T f'(x_k) = f''(x_k)^T f(x_k) + M(x_k) .$$

- The approximation error $f''(x_k)^T f(x_k)$ is small if either the residuum $f(x_k)$ or the second derivative $f''(x_k)$ is small.
- The Gauss-Newton Hessian approximation $M(x_k) = f'(x_k)^T f'(x_k)$ is always positive semi-definite.
- Similar to exact Newton methods, Gauss-Newton methods are invariant under scaling.
- $M(x_k)$ depends on first order derivatives only.

Properties of Gauss-Newton Hessian approximations

The exact Hessian matrix is given by

$$F''(x_k) = f''(x_k)^T f(x_k) + f'(x_k)^T f'(x_k) = f''(x_k)^T f(x_k) + M(x_k) .$$

- The approximation error $f''(x_k)^T f(x_k)$ is small if either the residuum $f(x_k)$ or the second derivative $f''(x_k)$ is small.
- The Gauss-Newton Hessian approximation $M(x_k) = f'(x_k)^T f'(x_k)$ is always positive semi-definite.
- Similar to exact Newton methods, Gauss-Newton methods are invariant under scaling.
- $M(x_k)$ depends on first order derivatives only.

Properties of Gauss-Newton Hessian approximations

The exact Hessian matrix is given by

$$F''(x_k) = f''(x_k)^T f(x_k) + f'(x_k)^T f'(x_k) = f''(x_k)^T f(x_k) + M(x_k) .$$

- The approximation error $f''(x_k)^T f(x_k)$ is small if either the residuum $f(x_k)$ or the second derivative $f''(x_k)$ is small.
- The Gauss-Newton Hessian approximation $M(x_k) = f'(x_k)^T f'(x_k)$ is always positive semi-definite.
- Similar to exact Newton methods, Gauss-Newton methods are invariant under scaling.
- $M(x_k)$ depends on first order derivatives only.

Properties of Gauss-Newton Hessian approximations

The exact Hessian matrix is given by

$$F''(x_k) = f''(x_k)^T f(x_k) + f'(x_k)^T f'(x_k) = f''(x_k)^T f(x_k) + M(x_k) .$$

- The approximation error $f''(x_k)^T f(x_k)$ is small if either the residuum $f(x_k)$ or the second derivative $f''(x_k)$ is small.
- The Gauss-Newton Hessian approximation $M(x_k) = f'(x_k)^T f'(x_k)$ is always positive semi-definite.
- Similar to exact Newton methods, Gauss-Newton methods are invariant under scaling.
- $M(x_k)$ depends on first order derivatives only.

Properties of Gauss-Newton Hessian approximations

The exact Hessian matrix is given by

$$F''(x_k) = f''(x_k)^T f(x_k) + f'(x_k)^T f'(x_k) = f''(x_k)^T f(x_k) + M(x_k) .$$

- The approximation error $f''(x_k)^T f(x_k)$ is small if either the residuum $f(x_k)$ or the second derivative $f''(x_k)$ is small.
- The Gauss-Newton Hessian approximation $M(x_k) = f'(x_k)^T f'(x_k)$ is always positive semi-definite.
- Similar to exact Newton methods, Gauss-Newton methods are invariant under scaling.
- $M(x_k)$ depends on first order derivatives only.

Contents

- Problem Formulation
- Gauss-Newton Methods
- Local Convergence Analysis

Local Convergence Analysis

As Gauss-Newton methods are special Newton-type methods the local convergence bound has the form

$$\|x_{k+1} - x^*\| \leq \kappa \|x_k - x^*\| + \frac{\omega}{2} \|x_k - x^*\|_2^2 ,$$

where

$$\|I - M(x_k)^{-1} F''(x_k)\| \leq \|M(x_k)^{-1}\| \|f(x_k)\| \|f''(x_k)\| \leq \kappa$$

- If we have $f''(x) = 0$, the method converges in one step.
- If we have $f(x^*) = 0$ at the limit point x^* , we have

$$\kappa \leq \mathbf{O}(\|x_k - x^*\|)$$

and the method converges with locally quadratic rate.

Local Convergence Analysis

As Gauss-Newton methods are special Newton-type methods the local convergence bound has the form

$$\|x_{k+1} - x^*\| \leq \kappa \|x_k - x^*\| + \frac{\omega}{2} \|x_k - x^*\|_2^2 ,$$

where

$$\|I - M(x_k)^{-1} F''(x_k)\| \leq \|M(x_k)^{-1}\| \|f(x_k)\| \|f''(x_k)\| \leq \kappa$$

- If we have $f''(x) = 0$, the method converges in one step.
- If we have $f(x^*) = 0$ at the limit point x^* , we have

$$\kappa \leq \mathbf{O}(\|x_k - x^*\|)$$

and the method converges with locally quadratic rate.

Local Convergence Analysis

As Gauss-Newton methods are special Newton-type methods the local convergence bound has the form

$$\|x_{k+1} - x^*\| \leq \kappa \|x_k - x^*\| + \frac{\omega}{2} \|x_k - x^*\|_2^2 ,$$

where

$$\|I - M(x_k)^{-1} F''(x_k)\| \leq \|M(x_k)^{-1}\| \|f(x_k)\| \|f''(x_k)\| \leq \kappa$$

- If we have $f''(x) = 0$, the method converges in one step.
- If we have $f(x^*) = 0$ at the limit point x^* , we have

$$\kappa \leq \mathbf{O}(\|x_k - x^*\|)$$

and the method converges with locally quadratic rate.

Chapter 10

Nonlinear Differential Equations

Nonlinear Differential Equations

- Nonlinear Differential Equations
- Existence and uniqueness of solutions
- Taylor-Model Based Integrators
- Runge-Kutta Integrators
- Stability Analysis
- Multi-Step Methods

Contents

- Nonlinear Differential Equations
- Existence and uniqueness of solutions
- Taylor-Model Based Integrators
- Runge-Kutta Integrators
- Stability Analysis
- Multi-Step Methods

Problem Formulation

The focus of this lecture is on ordinary differential equations (ODEs),

$$\forall t \in [0, T], \quad \dot{x}(t) = f(t, x(t)) \quad \text{with} \quad x(0) = x_0 .$$

Here, $x : [0, T] \rightarrow \mathbb{R}^n$ is the state trajectory.

Assumptions:

- The function $f : \mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R}^n$ may be nonlinear.
- The initial value $x_0 \in \mathbb{R}^n$ is given.

Problem Formulation

The focus of this lecture is on ordinary differential equations (ODEs),

$$\forall t \in [0, T], \quad \dot{x}(t) = f(t, x(t)) \quad \text{with} \quad x(0) = x_0 .$$

Here, $x : [0, T] \rightarrow \mathbb{R}^n$ is the state trajectory.

Assumptions:

- The function $f : \mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R}^n$ may be nonlinear.
- The initial value $x_0 \in \mathbb{R}^n$ is given.

Problem Formulation

The focus of this lecture is on ordinary differential equations (ODEs),

$$\forall t \in [0, T], \quad \dot{x}(t) = f(t, x(t)) \quad \text{with} \quad x(0) = x_0 .$$

Here, $x : [0, T] \rightarrow \mathbb{R}^n$ is the state trajectory.

Assumptions:

- The function $f : \mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R}^n$ may be nonlinear.
- The initial value $x_0 \in \mathbb{R}^n$ is given.

Explicit solution

- In general: no explicit solution possible
- But in some special cases, we can solve the nonlinear differential equation by using the concept of separation of variables.

Separation of variables:

- Assumption: f is scalar separable; that is,

$$f(t, x) = f_1(x)f_2(t) .$$

- Strategy: integrate the equation

$$\frac{\dot{x}(t)}{f_1(x(t))} = f_2(t) ,$$

with respect to t on both sides and eliminate $x(t)$.

Explicit solution

- In general: no explicit solution possible
- But in some special cases, we can solve the nonlinear differential equation by using the concept of separation of variables.

Separation of variables:

- Assumption: f is scalar separable; that is,

$$f(t, x) = f_1(x)f_2(t) .$$

- Strategy: integrate the equation

$$\frac{\dot{x}(t)}{f_1(x(t))} = f_2(t) ,$$

with respect to t on both sides and eliminate $x(t)$.

Example: quadratic differential equation

Nonlinear ODE:

$$\dot{x}(t) = -x^2(t) \quad \text{with} \quad x(0) = 1 .$$

Separation of variables:

$$-\frac{\dot{x}(t)}{x(t)^2} = 1 \quad \xRightarrow{\text{integrate}} \quad \frac{1}{x(t)} - \frac{1}{x(0)} = t$$

Elimination of $x(t)$:

$$x(t) = \frac{1}{1+t} \quad \text{for all} \quad t \geq 0 .$$

Example: quadratic differential equation

Nonlinear ODE:

$$\dot{x}(t) = -x^2(t) \quad \text{with} \quad x(0) = 1 .$$

Separation of variables:

$$-\frac{\dot{x}(t)}{x(t)^2} = 1 \quad \xRightarrow{\text{integrate}} \quad \frac{1}{x(t)} - \frac{1}{x(0)} = t$$

Elimination of $x(t)$:

$$x(t) = \frac{1}{1+t} \quad \text{for all} \quad t \geq 0 .$$

Example: quadratic differential equation

Nonlinear ODE:

$$\dot{x}(t) = -x^2(t) \quad \text{with} \quad x(0) = 1 .$$

Separation of variables:

$$-\frac{\dot{x}(t)}{x(t)^2} = 1 \quad \xRightarrow{\text{integrate}} \quad \frac{1}{x(t)} - \frac{1}{x(0)} = t$$

Elimination of $x(t)$:

$$x(t) = \frac{1}{1+t} \quad \text{for all} \quad t \geq 0 .$$

Example: Gauss' differential equation

ODE:

$$\dot{x}(t) = -tx(t) \quad \text{with} \quad x(0) = 1 .$$

Separation of variables:

$$\frac{\dot{x}(t)}{x(t)} = -t \quad \implies \quad \log(x(t)) = -\frac{1}{2}t^2$$

Elimination of $x(t)$:

$$x(t) = e^{-\frac{t^2}{2}} ,$$

Example: Gauss' differential equation

ODE:

$$\dot{x}(t) = -tx(t) \quad \text{with} \quad x(0) = 1 .$$

Separation of variables:

$$\frac{\dot{x}(t)}{x(t)} = -t \quad \Longrightarrow \quad \log(x(t)) = -\frac{1}{2}t^2$$

Elimination of $x(t)$:

$$x(t) = e^{-\frac{t^2}{2}} ,$$

Example: Gauss' differential equation

ODE:

$$\dot{x}(t) = -tx(t) \quad \text{with} \quad x(0) = 1 .$$

Separation of variables:

$$\frac{\dot{x}(t)}{x(t)} = -t \quad \Longrightarrow \quad \log(x(t)) = -\frac{1}{2}t^2$$

Elimination of $x(t)$:

$$x(t) = e^{-\frac{t^2}{2}} ,$$

Contents

- Nonlinear Differential Equations
- Existence and uniqueness of solutions
- Taylor-Model Based Integrators
- Runge-Kutta Integrators
- Stability Analysis
- Multi-Step Methods

Integral Form

The ordinary differential equation (ODE)

$$\forall t \in [0, T], \quad \dot{x}(t) = f(x(t)) \quad \text{with} \quad x(0) = x_0 .$$

can be equivalently be written in its integral form

$$\forall t \in [0, T], \quad x(t) = x_0 + \int_0^t f(x(s)) \, ds .$$

Lipschitz continuity

Recall:

- A function $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ is called (globally) Lipschitz continuous, if there exist a constant $L < \infty$ with

$$\forall x, y \in \mathbb{R}, \quad \|f(x) - f(y)\| \leq L\|x - y\| .$$

Existence and Uniqueness

Theorem (Picard-Lindelöf):

- If f is globally Lipschitz continuous, the ODE has a unique solution.

Proof: (main idea, rough sketch only)

1) Start with any continuous function $y_1 : [0, T] \rightarrow \mathbb{R}$ and iterate

$$y_{i+1}(t) = x_0 + \int_0^t f(y_i(s)) \, ds \quad [\text{Picard iteration}]$$

2) Show that $y_1, y_2, y_3, \dots$ is a Cauchy sequence, $y^* = \lim_{k \rightarrow \infty} y_i$.

3) Conclude that the (unique) limit point y^* satisfies the ODE.

Existence and Uniqueness

Theorem (Picard-Lindelöf):

- If f is globally Lipschitz continuous, the ODE has a unique solution.

Proof: (main idea, rough sketch only)

- 1) Start with any continuous function $y_1 : [0, T] \rightarrow \mathbb{R}$ and iterate

$$y_{i+1}(t) = x_0 + \int_0^t f(y_i(s)) \, ds \quad [\text{Picard iteration}]$$

- 2) Show that $y_1, y_2, y_3, \dots$ is a Cauchy sequence, $y^* = \lim_{k \rightarrow \infty} y_i$.
- 3) Conclude that the (unique) limit point y^* satisfies the ODE.

Existence and Uniqueness

Theorem (Picard-Lindelöf):

- If f is globally Lipschitz continuous, the ODE has a unique solution.

Proof: (main idea, rough sketch only)

- 1) Start with any continuous function $y_1 : [0, T] \rightarrow \mathbb{R}$ and iterate

$$y_{i+1}(t) = x_0 + \int_0^t f(y_i(s)) \, ds \quad [\text{Picard iteration}]$$

- 2) Show that $y_1, y_2, y_3, \dots$ is a Cauchy sequence, $y^* = \lim_{k \rightarrow \infty} y_i$.

- 3) Conclude that the (unique) limit point y^* satisfies the ODE.

Existence and Uniqueness

Theorem (Picard-Lindelöf):

- If f is globally Lipschitz continuous, the ODE has a unique solution.

Proof: (main idea, rough sketch only)

- 1) Start with any continuous function $y_1 : [0, T] \rightarrow \mathbb{R}$ and iterate

$$y_{i+1}(t) = x_0 + \int_0^t f(y_i(s)) \, ds \quad [\text{Picard iteration}]$$

- 2) Show that $y_1, y_2, y_3, \dots$ is a Cauchy sequence, $y^* = \lim_{k \rightarrow \infty} y_i$.
- 3) Conclude that the (unique) limit point y^* satisfies the ODE.

Some technical details

- Define $\Delta(t) = \max_{s \in [0, t]} |y_2(s) - y_1(s)|$.
- If $|y_{i+1}(t) - y_i(t)| \leq \frac{(tL)^{i-1}}{(i-1)!} \Delta(t)$, then

$$\begin{aligned} |y_{i+2}(t) - y_{i+1}(t)| &\leq L \left| \int_0^t [y_{i+1}(\tau) - y_i(\tau)] d\tau \right| \\ &\leq \int_0^t L \frac{(\tau L)^{i-1}}{(i-1)!} \Delta(t) d\tau = \frac{(tL)^i}{i!} \Delta(t) . \end{aligned}$$

- Thus, we have

$$\begin{aligned} |y_n(t) - y_m(t)| &\leq \sum_{i=n}^{m-1} |y_{i+1}(t) - y_i(t)| \leq \sum_{i=n}^{m-1} \frac{(tL)^{i-1}}{(i-1)!} \Delta(t) \\ &\leq \frac{(tL)^{n-1}}{(n-1)!} e^{L|t|} \Delta(t) , \end{aligned}$$

i.e., (y_k) is a Cauchy sequence.

Some technical details

- Define $\Delta(t) = \max_{s \in [0, t]} |y_2(s) - y_1(s)|$.
- If $|y_{i+1}(t) - y_i(t)| \leq \frac{(tL)^{i-1}}{(i-1)!} \Delta(t)$, then

$$\begin{aligned} |y_{i+2}(t) - y_{i+1}(t)| &\leq L \left| \int_0^t [y_{i+1}(\tau) - y_i(\tau)] \, d\tau \right| \\ &\leq \int_0^t L \frac{(\tau L)^{i-1}}{(i-1)!} \Delta(t) \, d\tau = \frac{(tL)^i}{i!} \Delta(t) . \end{aligned}$$

- Thus, we have

$$\begin{aligned} |y_n(t) - y_m(t)| &\leq \sum_{i=n}^{m-1} |y_{i+1}(t) - y_i(t)| \leq \sum_{i=n}^{m-1} \frac{(tL)^{i-1}}{(i-1)!} \Delta(t) \\ &\leq \frac{(tL)^{n-1}}{(n-1)!} e^{L|t|} \Delta(t) , \end{aligned}$$

i.e., (y_k) is a Cauchy sequence.

Some technical details

- Define $\Delta(t) = \max_{s \in [0, t]} |y_2(s) - y_1(s)|$.
- If $|y_{i+1}(t) - y_i(t)| \leq \frac{(tL)^{i-1}}{(i-1)!} \Delta(t)$, then

$$\begin{aligned} |y_{i+2}(t) - y_{i+1}(t)| &\leq L \left| \int_0^t [y_{i+1}(\tau) - y_i(\tau)] \, d\tau \right| \\ &\leq \int_0^t L \frac{(\tau L)^{i-1}}{(i-1)!} \Delta(t) \, d\tau = \frac{(tL)^i}{i!} \Delta(t) . \end{aligned}$$

- Thus, we have

$$\begin{aligned} |y_n(t) - y_m(t)| &\leq \sum_{i=n}^{m-1} |y_{i+1}(t) - y_i(t)| \leq \sum_{i=n}^{m-1} \frac{(tL)^{i-1}}{(i-1)!} \Delta(t) \\ &\leq \frac{(tL)^{n-1}}{(n-1)!} e^{L|t|} \Delta(t) , \end{aligned}$$

i.e., (y_k) is a Cauchy sequence.

Some technical details

- Define $\Delta(t) = \max_{s \in [0, t]} |y_2(s) - y_1(s)|$.
- If $|y_{i+1}(t) - y_i(t)| \leq \frac{(tL)^{i-1}}{(i-1)!} \Delta(t)$, then

$$\begin{aligned} |y_{i+2}(t) - y_{i+1}(t)| &\leq L \left| \int_0^t [y_{i+1}(\tau) - y_i(\tau)] \, d\tau \right| \\ &\leq \int_0^t L \frac{(\tau L)^{i-1}}{(i-1)!} \Delta(t) \, d\tau = \frac{(tL)^i}{i!} \Delta(t) . \end{aligned}$$

- Thus, we have

$$\begin{aligned} |y_n(t) - y_m(t)| &\leq \sum_{i=n}^{m-1} |y_{i+1}(t) - y_i(t)| \leq \sum_{i=n}^{m-1} \frac{(tL)^{i-1}}{(i-1)!} \Delta(t) \\ &\leq \frac{(tL)^{n-1}}{(n-1)!} e^{L|t|} \Delta(t) , \end{aligned}$$

i.e., (y_k) is a Cauchy sequence.

Example: Linear ODEs

- Linear ODE: $\dot{x}(t) = Ax(t)$, $A \in \mathbb{R}^{n \times n}$, with $x(0) = x_0 \in \mathbb{R}^n$.
- Picard iteration:

$$y_1(t) = x_0$$

$$y_2(t) = x_0 + tAx_0$$

$$y_3(t) = x_0 + tAx_0 + \frac{t^2}{2}A^2x_0$$

$$\vdots$$

- Take the limit to get explicit solution

$$x(t) = e^{At}x_0 = \sum_{i=0}^{\infty} \frac{1}{i!} [tA]^i x_0 .$$

Example: Linear ODEs

- Linear ODE: $\dot{x}(t) = Ax(t)$, $A \in \mathbb{R}^{n \times n}$, with $x(0) = x_0 \in \mathbb{R}^n$.
- Picard iteration:

$$y_1(t) = x_0$$

$$y_2(t) = x_0 + tAx_0$$

$$y_3(t) = x_0 + tAx_0 + \frac{t^2}{2}A^2x_0$$

$$\vdots$$

- Take the limit to get explicit solution

$$x(t) = e^{At}x_0 = \sum_{i=0}^{\infty} \frac{1}{i!} [tA]^i x_0 .$$

Example: Linear ODEs

- Linear ODE: $\dot{x}(t) = Ax(t)$, $A \in \mathbb{R}^{n \times n}$, with $x(0) = x_0 \in \mathbb{R}^n$.
- Picard iteration:

$$y_1(t) = x_0$$

$$y_2(t) = x_0 + tAx_0$$

$$y_3(t) = x_0 + tAx_0 + \frac{t^2}{2}A^2x_0$$

$$\vdots$$

- Take the limit to get explicit solution

$$x(t) = e^{At}x_0 = \sum_{i=0}^{\infty} \frac{1}{i!} [tA]^i x_0 .$$

Examples for nonlinear ODEs

- The ODE $\dot{x}(t) = x(t)^2$, with $x(0) = 1$ has the explicit solution

$$x(t) = \frac{1}{1-t} \quad \text{for } t < 1$$

Why does the solution not exist for $t \geq 1$?

- The ODE $\dot{x}(t) = 2\sqrt{x}$, with $x(0) = 0$ has more than one solution,

for example $x(t) = 0$ and $x(t) = t^2$.

Why is there more than one solution?

Examples for nonlinear ODEs

- The ODE $\dot{x}(t) = x(t)^2$, with $x(0) = 1$ has the explicit solution

$$x(t) = \frac{1}{1-t} \quad \text{for } t < 1$$

Why does the solution not exist for $t \geq 1$?

- The ODE $\dot{x}(t) = 2\sqrt{x}$, with $x(0) = 0$ has more than one solution,

$$\text{for example } x(t) = 0 \quad \text{and} \quad x(t) = t^2.$$

Why is there more than one solution?

Examples for nonlinear ODEs

- The ODE $\dot{x}(t) = x(t)^2$, with $x(0) = 1$ has the explicit solution

$$x(t) = \frac{1}{1-t} \quad \text{for } t < 1$$

Why does the solution not exist for $t \geq 1$?

- The ODE $\dot{x}(t) = 2\sqrt{x}$, with $x(0) = 0$ has more than one solution,

$$\text{for example } x(t) = 0 \quad \text{and} \quad x(t) = t^2 .$$

Why is there more than one solution?

Gronwall's Lemma

Let f be globally Lipschitz continuous and

$$\dot{x}(t) = f(x(t)) \quad x(0) = x_0 \quad (1)$$

$$\dot{z}(t) = f(z(t)) \quad z(0) = z_0 . \quad (2)$$

Then we have

$$\|x(t) - z(t)\| \leq \|x_0 - z_0\| e^{Lt} .$$

Proof of Gronwall's Lemma

Main idea: start the Picard iteration

$$y_{i+1}(t) = z_0 + \int_0^t f(y_i(s)) \, ds \quad \text{at} \quad y_1(t) = x(t) .$$

The first iteration is given by

$$y_2(t) = z_0 + \int_0^t f(x(s)) \, ds = x(t) + [z_0 - x_0] = x(t) + e_1(t)$$

with

$$e_1(t) = y_2(t) - x(t) \quad \text{and} \quad \|e_1(t)\| \leq \|x_0 - z_0\|$$

Proof of Gronwall's Lemma

1. Use the first iteration as induction start (previous slide).
2. The induction assumption is that

$$y_{k+1}(t) = x(t) + e_k(t)$$

$$\text{and } \|e_k(t)\| \leq \|x_0 - z_0\| \sum_{i=0}^{k-1} \frac{L^i t^i}{i!}$$

3. Induction step: set $e_{k+1}(t) = y_{k+2}(t) - x(t)$ and work out

$$\begin{aligned} \|e_{k+1}(t)\| &= \left\| z_0 + \int_0^t f(x(s) + e_k(s)) \, ds - x(t) \right\| \\ &\leq \|x_0 - z_0\| + \left\| \int_0^t f(x(s) + e_k(s)) - f(x(s)) \, ds \right\| \\ &\leq \|x_0 - z_0\| \left[1 + \int_0^t \sum_{i=0}^{k-1} L \frac{L^i s^i}{i!} \, ds \right] = \|x_0 - z_0\| \sum_{i=0}^k \frac{L^i t^i}{i!}. \end{aligned}$$

Proof of Gronwall's Lemma

Summary:

- We know that the Picard iteration

$$y_{k+1}(t) = z_0 + \int_0^t f(y_k(s)) \, ds \quad \text{at} \quad y_1(t) = x(t) .$$

converges and the limit is given by

$$\lim_{k \rightarrow \infty} y_k(t) = z(t)$$

- The corresponding error is bounded by

$$\begin{aligned} \|z(t) - x(t)\| &= \lim_{k \rightarrow \infty} \|y_k(t) - x(t)\| = \lim_{k \rightarrow \infty} \|e_{k-1}(t)\| \\ &\leq \|x_0 - z_0\| \sum_{i=0}^{\infty} \frac{L^i t^i}{i!} = \|x_0 - z_0\| e^{Lt} . \end{aligned} \quad (3)$$

Conditioning of Differential Equations

- The factor e^{Lt} can be interpreted as a **global** upper bound on the condition number of a differential equation
- In general, for large t , predictions are impossible: “butterfly effect”.
- BUT: Gronwall's lemma has no information about the stability properties of the differential equation
- For some differential equations, a **local** analysis yields better bounds and potentially indicates *local stability*

First Order Variational Analysis

- Consider the differential equations

$$\dot{x}(t) = f(x(t)) \quad \text{with} \quad x(0) = x_0$$

$$\dot{z}(t) = f(z(t)) \quad \text{with} \quad z(0) = z_0$$

for a continuously differentiable right-hand f .

- The linear matrix differential equation

$$\dot{X}(t) = \frac{\partial f(x(t))}{\partial x} X(t) \quad \text{with} \quad X(0) = I$$

is called the first order variational differential equation.

- It yields the first order Taylor approximation

$$z(t) = x(t) + X(t)(z_0 - x_0) + \mathbf{o}(\|z_0 - x_0\|)$$

Unfortunately: in general only valid for finite $t \leq T < \infty$!

Details

Introduce the shorthands

$$e(t) = z(t) - x(t) - X(t)(z_0 - x_0) \quad \text{and} \quad A(t) = \frac{\partial f(x(t))}{\partial x}$$

We have

$$\begin{aligned}\dot{e}(t) &= f(z(t)) - f(x(t)) - \dot{X}(t)(z_0 - x_0) \\ &= A(t)e(t) + o(\|e(t)\|) + o(\|X(t)\|\|x_0 - z_0\|)\end{aligned}$$

$$\text{with } e(0) = 0.$$

We can use a Picard iteration (or variant of Gronwall's lemma) to show that $\|e(t)\| = o(\|x_0 - z_0\|)$ for all $t \leq T < \infty$. (Exercise!)

Steady-States

A point $x_0 \in \mathbb{R}^n$ is called a steady-state (or critical point) of f if

$$f(x_0) = 0 .$$

If f is continuously differentiable, the ODE

$$\dot{z}(t) = f(z(t)) \quad \text{with} \quad z(0) = z_0$$

can be analyzed in a local neighborhood of x_0 . We have

$$A = \frac{\partial f(x_0)}{\partial x_0} \quad \text{and} \quad X(t) = e^{At} = \sum_{i=0}^{\infty} \frac{1}{i!} (tA)^i .$$

Local Stability Analysis

- If the eigenvalues of A have all strictly negative real parts, then

$$\lim_{t \rightarrow \infty} e^{At} = 0 .$$

- In this case, we have

$$\forall t \in [0, \infty), \quad z(t) = e^{At} z_0 + o(\|z_0\|) \quad \text{and} \quad \lim_{t \rightarrow \infty} z(t) = 0$$

for sufficiently small $\|z_0\|$. This implies *local asymptotic stability*.

- A similar local stability analysis via first order variational analysis is possible in the neighborhood of periodic orbits (Floquet theory).

Contents

- Nonlinear Differential Equations
- Existence and uniqueness of solutions
- **Taylor-Model Based Integrators**
- Runge-Kutta Integrators
- Stability Analysis
- Multi-Step Methods

Taylor expansion of ODEs

A Taylor expansion of the solution $x(t)$ can be constructed recursively:

- $x(t_0) = x_0$
- $\dot{x}(t_0) = f(t_0, x_0)$
- $\ddot{x}(t_0) = \left. \frac{\partial}{\partial t} f(t, x(t)) \right|_{t=t_0} = f_t(t_0, x_0) + f_x(t_0, x_0)f(t_0, x_0)$
- and so on ...
- Finally, $x(t) =$
$$x_0 + f(t_0, x_0)(t - t_0) + \frac{(t-t_0)^2}{2} [f_t(t_0, x_0) + f_x(t_0, x_0)f(t_0, x_0)] + \dots$$

for small t .

Taylor expansion of ODEs

A Taylor expansion of the solution $x(t)$ can be constructed recursively:

- $x(t_0) = x_0$
- $\dot{x}(t_0) = f(t_0, x_0)$
- $\ddot{x}(t_0) = \left. \frac{\partial}{\partial t} f(t, x(t)) \right|_{t=t_0} = f_t(t_0, x_0) + f_x(t_0, x_0)f(t_0, x_0)$
- and so on ...
- Finally, $x(t) =$
$$x_0 + f(t_0, x_0)(t - t_0) + \frac{(t - t_0)^2}{2} [f_t(t_0, x_0) + f_x(t_0, x_0)f(t_0, x_0)] + \dots$$

for small t .

Taylor expansion of ODEs

A Taylor expansion of the solution $x(t)$ can be constructed recursively:

- $x(t_0) = x_0$
- $\dot{x}(t_0) = f(t_0, x_0)$
- $\ddot{x}(t_0) = \left. \frac{\partial}{\partial t} f(t, x(t)) \right|_{t=t_0} = f_t(t_0, x_0) + f_x(t_0, x_0)f(t_0, x_0)$
- and so on ...
- Finally, $x(t) =$
$$x_0 + f(t_0, x_0)(t - t_0) + \frac{(t-t_0)^2}{2} [f_t(t_0, x_0) + f_x(t_0, x_0)f(t_0, x_0)] + \dots$$

for small t .

Taylor expansion of ODEs

A Taylor expansion of the solution $x(t)$ can be constructed recursively:

- $x(t_0) = x_0$
- $\dot{x}(t_0) = f(t_0, x_0)$
- $\ddot{x}(t_0) = \left. \frac{\partial}{\partial t} f(t, x(t)) \right|_{t=t_0} = f_t(t_0, x_0) + f_x(t_0, x_0)f(t_0, x_0)$
- and so on ...
- Finally, $x(t) =$
$$x_0 + f(t_0, x_0)(t - t_0) + \frac{(t-t_0)^2}{2} [f_t(t_0, x_0) + f_x(t_0, x_0)f(t_0, x_0)] + \dots$$

for small t .

Taylor expansion of ODEs

A Taylor expansion of the solution $x(t)$ can be constructed recursively:

- $x(t_0) = x_0$
- $\dot{x}(t_0) = f(t_0, x_0)$
- $\ddot{x}(t_0) = \left. \frac{\partial}{\partial t} f(t, x(t)) \right|_{t=t_0} = f_t(t_0, x_0) + f_x(t_0, x_0)f(t_0, x_0)$
- and so on ...
- Finally, $x(t) =$
$$x_0 + f(t_0, x_0)(t - t_0) + \frac{(t-t_0)^2}{2} [f_t(t_0, x_0) + f_x(t_0, x_0)f(t_0, x_0)] + \dots$$

for small t .

Taylor expansion of ODEs

A Taylor expansion of the solution $x(t)$ can be constructed recursively:

- $x(t_0) = x_0$
- $\dot{x}(t_0) = f(t_0, x_0)$
- $\ddot{x}(t_0) = \left. \frac{\partial}{\partial t} f(t, x(t)) \right|_{t=t_0} = f_t(t_0, x_0) + f_x(t_0, x_0)f(t_0, x_0)$
- and so on ...
- Finally, $x(t) =$
$$x_0 + f(t_0, x_0)(t - t_0) + \frac{(t-t_0)^2}{2} [f_t(t_0, x_0) + f_x(t_0, x_0)f(t_0, x_0)] + \dots$$

for small t .

Taylor expansion of ODEs

A general Taylor expansion can be computed by consecutive differentiation:

1. Set $\phi_0(t, x) = x$.

2. For $r = 0 : s - 1$

$$\text{set } \phi_{r+1}(t, x) = \left(\frac{\partial}{\partial t} \phi_r(t, x) \right) + \left(\frac{\partial}{\partial x} \phi_r(t, x) \right) f(t, x).$$

3. Return the Taylor expansion

$$x(t) = \sum_{i=0}^s \frac{1}{i!} \phi_i(t_0, x_0) (t - t_0)^i + \mathbf{O}((t - t_0)^{s+1}).$$

Taylor expansion of ODEs

A general Taylor expansion can be computed by consecutive differentiation:

1. Set $\phi_0(t, x) = x$.

2. For $r = 0 : s - 1$

$$\text{set } \phi_{r+1}(t, x) = \left(\frac{\partial}{\partial t} \phi_r(t, x) \right) + \left(\frac{\partial}{\partial x} \phi_r(t, x) \right) f(t, x).$$

3. Return the Taylor expansion

$$x(t) = \sum_{i=0}^s \frac{1}{i!} \phi_i(t_0, x_0) (t - t_0)^i + \mathbf{O}((t - t_0)^{s+1}).$$

Taylor expansion of ODEs

A general Taylor expansion can be computed by consecutive differentiation:

1. Set $\phi_0(t, x) = x$.

2. For $r = 0 : s - 1$

$$\text{set } \phi_{r+1}(t, x) = \left(\frac{\partial}{\partial t} \phi_r(t, x) \right) + \left(\frac{\partial}{\partial x} \phi_r(t, x) \right) f(t, x).$$

3. Return the Taylor expansion

$$x(t) = \sum_{i=0}^s \frac{1}{i!} \phi_i(t_0, x_0) (t - t_0)^i + \mathbf{O}((t - t_0)^{s+1}).$$

Integration Algorithm (Constant Step-Size)

Input:

- The right-hand side function f and an initial value x_0 .
- Order s and constant step-size $h = T/N$; set $i = 0$ and $y_0 = x_0$.

Repeat: (until $i = N$)

- Compute $y_{i+1} = \sum_{k=0}^s \frac{1}{k!} \phi_k(t_i, y_i) h^k$
- Compute $t_{i+1} = t_i + h$ and set $i \leftarrow i + 1$.

Theorem:

- If f is globally Lipschitz continuous and smooth, then

$$\forall i \in \{0, \dots, N\}, \quad y_i = x(t_i) + \mathbf{O}(h^s) .$$

Integration Algorithm (Constant Step-Size)

Input:

- The right-hand side function f and an initial value x_0 .
- Order s and constant step-size $h = T/N$; set $i = 0$ and $y_0 = x_0$.

Repeat: (until $i = N$)

- Compute $y_{i+1} = \sum_{k=0}^s \frac{1}{k!} \phi_k(t_i, y_i) h^k$
- Compute $t_{i+1} = t_i + h$ and set $i \leftarrow i + 1$.

Theorem:

- If f is globally Lipschitz continuous and smooth, then

$$\forall i \in \{0, \dots, N\}, \quad y_i = x(t_i) + \mathbf{O}(h^s) .$$

Integration Algorithm (Constant Step-Size)

Input:

- The right-hand side function f and an initial value x_0 .
- Order s and constant step-size $h = T/N$; set $i = 0$ and $y_0 = x_0$.

Repeat: (until $i = N$)

- Compute $y_{i+1} = \sum_{k=0}^s \frac{1}{k!} \phi_k(t_i, y_i) h^k$
- Compute $t_{i+1} = t_i + h$ and set $i \leftarrow i + 1$.

Theorem:

- If f is globally Lipschitz continuous and smooth, then

$$\forall i \in \{0, \dots, N\}, \quad y_i = x(t_i) + \mathbf{O}(h^s) .$$

Proof (main idea)

1. Since f is globally Lipschitz, the solution x of the ODE exists.
2. Since f is smooth, the functions $\phi_0, \phi_1, \dots, \phi_s$ are smooth, too.
3. We already know that $x(t) = \sum_{k=0}^s \frac{1}{k!} \phi_k(x_0) h^k + \mathbf{O}(h^{s+1})$.
4. Show by induction (use Gronwall's lemma for a clean proof) that

$$y_i = x(ih) + i \cdot \mathbf{O}(h^{s+1}) = x(ih) + \frac{T}{h} \cdot \mathbf{O}(h^{s+1}) = x(ih) + \mathbf{O}(h^s).$$

The integer s is called the convergence order of the integrator.

Proof (main idea)

1. Since f is globally Lipschitz, the solution x of the ODE exists.
2. Since f is smooth, the functions $\phi_0, \phi_1, \dots, \phi_s$ are smooth, too.
3. We already know that $x(t) = \sum_{k=0}^s \frac{1}{k!} \phi_k(x_0) h^k + \mathbf{O}(h^{s+1})$.
4. Show by induction (use Gronwall's lemma for a clean proof) that

$$y_i = x(ih) + i \cdot \mathbf{O}(h^{s+1}) = x(ih) + \frac{T}{h} \cdot \mathbf{O}(h^{s+1}) = x(ih) + \mathbf{O}(h^s).$$

The integer s is called the convergence order of the integrator.

Proof (main idea)

1. Since f is globally Lipschitz, the solution x of the ODE exists.
2. Since f is smooth, the functions $\phi_0, \phi_1, \dots, \phi_s$ are smooth, too.
3. We already know that $x(t) = \sum_{k=0}^s \frac{1}{k!} \phi_k(x_0) h^k + \mathbf{O}(h^{s+1})$.
4. Show by induction (use Gronwall's lemma for a clean proof) that

$$y_i = x(ih) + i \cdot \mathbf{O}(h^{s+1}) = x(ih) + \frac{T}{h} \cdot \mathbf{O}(h^{s+1}) = x(ih) + \mathbf{O}(h^s).$$

The integer s is called the convergence order of the integrator.

Proof (main idea)

1. Since f is globally Lipschitz, the solution x of the ODE exists.
2. Since f is smooth, the functions $\phi_0, \phi_1, \dots, \phi_s$ are smooth, too.
3. We already know that $x(t) = \sum_{k=0}^s \frac{1}{k!} \phi_k(x_0) h^k + \mathbf{O}(h^{s+1})$.
4. Show by induction (use Gronwall's lemma for a clean proof) that

$$y_i = x(ih) + i \cdot \mathbf{O}(h^{s+1}) = x(ih) + \frac{T}{h} \cdot \mathbf{O}(h^{s+1}) = x(ih) + \mathbf{O}(h^s).$$

The integer s is called the convergence order of the integrator.

Proof (main idea)

1. Since f is globally Lipschitz, the solution x of the ODE exists.
2. Since f is smooth, the functions $\phi_0, \phi_1, \dots, \phi_s$ are smooth, too.
3. We already know that $x(t) = \sum_{k=0}^s \frac{1}{k!} \phi_k(x_0) h^k + \mathbf{O}(h^{s+1})$.
4. Show by induction (use Gronwall's lemma for a clean proof) that

$$y_i = x(ih) + i \cdot \mathbf{O}(h^{s+1}) = x(ih) + \frac{T}{h} \cdot \mathbf{O}(h^{s+1}) = x(ih) + \mathbf{O}(h^s).$$

The integer s is called the convergence order of the integrator.

Contents

- Nonlinear Differential Equations
- Existence and uniqueness of solutions
- Taylor-Model Based Integrators
- **Runge-Kutta Integrators**
- Stability Analysis
- Multi-Step Methods

Limitations of Taylor model based integrators

1. Taylor model based integration is easy to implement, but
 - we need to evaluate derivatives of f
 - it is not the most efficient scheme for obtaining convergence order s .
2. Runge-Kutta integrators compute an approximation $y \approx x(h)$ by evaluating f at more than one point, but don't evaluate derivatives.

Limitations of Taylor model based integrators

1. Taylor model based integration is easy to implement, but
 - we need to evaluate derivatives of f
 - it is not the most efficient scheme for obtaining convergence order s .
2. Runge-Kutta integrators compute an approximation $y \approx x(h)$ by evaluating f at more than one point, but don't evaluate derivatives.

Explicit Runge Kutta method (constant step-size)

Initialization:

- Set $h = T/N$, $t_0 = 0$, $i = 0$, and $y_0 = x_0$.

Repeat: (until $i = N$)

- Compute $t_{i+1} = t_i + h$.
- Compute $k_r = f(t_i + h\gamma_r, y_i + \sum_{j=1}^{r-1} h\alpha_{r,j}k_j)$ for $r = 1, \dots, s$.
- Set $y_{i+1} = y_i + h \sum_{r=1}^s \beta_r k_r$ and then $i \leftarrow i + 1$.

Output:

- Time grid $[t_1, t_2, \dots, t_N]$ and state trajectory $y_0, y_1, y_2, \dots, y_N$.

Consistency conditions

Main idea:

- Choose the coefficients $\alpha_{r,j}$, β_r , and γ_r such that

$$\forall r \in \{1, \dots, q\}, \quad \left. \frac{\partial^r y_{i+1}}{\partial h^r} \right|_{h=0} = \Phi_r(y_i) .$$

Example 1: Euler's method

- For $s = 1$, the Runge-Kutta method takes the form

$$\begin{aligned}k_1 &= f(t_i, y_i) \\ y_{i+1} &= y_i + h\beta_1 k_1 = y_i + h\beta_1 f(t_i, y_i)\end{aligned}\tag{4}$$

- We have

$$\left. \frac{\partial y_{i+1}}{\partial h} \right|_{h=0} = \left. \frac{\partial}{\partial h} (y_i + h\beta_1 f(t_i, y_i)) \right|_{h=0} = \beta_1 f(t_i, y_i)\tag{5}$$

and

$$\phi_1(t, x) = f(t, x)\tag{6}$$

Example 1: Euler's method

- For $s = 1$, the Runge-Kutta method takes the form

$$\begin{aligned}k_1 &= f(t_i, y_i) \\ y_{i+1} &= y_i + h\beta_1 k_1 = y_i + h\beta_1 f(t_i, y_i)\end{aligned}\tag{4}$$

- We have

$$\left. \frac{\partial y_{i+1}}{\partial h} \right|_{h=0} = \left. \frac{\partial}{\partial h} (y_i + h\beta_1 f(t_i, y_i)) \right|_{h=0} = \beta_1 f(t_i, y_i)\tag{5}$$

and

$$\phi_1(t, x) = f(t, x)\tag{6}$$

Example 1: Euler's method

- The equation

$$\left. \frac{\partial y_{i+1}}{\partial h} \right|_{h=0} = \phi_1(t_i, y_i) \quad \stackrel{(5),(6)}{\Longleftrightarrow} \quad \beta_1 f(t_i, y_i) = f(t_i, y_i)$$

is satisfied for $\beta_1 = 1$.

- Result: Euler's method

$$y_{i+1} = y_i + hf(t_i, y_i) .$$

Example 1: Euler's method

- The equation

$$\left. \frac{\partial y_{i+1}}{\partial h} \right|_{h=0} = \phi_1(t_i, y_i) \quad \stackrel{(5),(6)}{\Longleftrightarrow} \quad \beta_1 f(t_i, y_i) = f(t_i, y_i)$$

is satisfied for $\beta_1 = 1$.

- Result: Euler's method

$$y_{i+1} = y_i + hf(t_i, y_i) .$$

Example 2: Heun's method

- Heun's method is given by the coefficient scheme

$$\begin{array}{c|cc} \gamma_1 & 0 & \\ \gamma_2 & \alpha_{2,1} & 0 \\ \hline & \beta_1 & \beta_2 \end{array} = \begin{array}{c|cc} 0 & 0 & \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

- The corresponding method can be written as

$$\begin{aligned} k_1 &= f(t_i, y_i) \\ k_2 &= f(t_i + h, y_i + hk_1) \\ y_{i+1} &= y_i + h \left(\frac{1}{2}k_1 + \frac{1}{2}k_2 \right). \end{aligned}$$

Example 2: Heun's method

- Heun's method is given by the coefficient scheme

$$\begin{array}{c|cc}
 \gamma_1 & 0 & \\
 \gamma_2 & \alpha_{2,1} & 0 \\
 \hline
 & \beta_1 & \beta_2
 \end{array}
 =
 \begin{array}{c|cc}
 0 & 0 & \\
 1 & 1 & 0 \\
 \hline
 & \frac{1}{2} & \frac{1}{2}
 \end{array}$$

- The corresponding method can be written as

$$\begin{aligned}
 k_1 &= f(t_i, y_i) \\
 k_2 &= f(t_i + h, y_i + hk_1) \\
 y_{i+1} &= y_i + h \left(\frac{1}{2}k_1 + \frac{1}{2}k_2 \right) .
 \end{aligned}$$

Example 3: RK 4

- A very elegant method of order 4 is given by the scheme

$$\begin{aligned}k_1 &= f(t_i, y_i) \\k_2 &= f\left(t_i + \frac{h}{2}, y_i + \frac{h}{2}k_1\right) \\k_3 &= f\left(t_i + \frac{h}{2}, y_i + \frac{h}{2}k_2\right) \\k_4 &= f(t_i + h, y_i + hk_3) \\y_{i+1} &= y_i + h\left(\frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4\right) .\end{aligned}$$

This method is called the classical Runge Kutta method.

Step-size control

Main idea

- User input: local error tolerance TOL and absolute tolerance ATOL.
- Compute two approximations

$$y_{n+1} = x(y_n, h) + \mathbf{O}(h^{r+1}) \quad \text{and} \quad z_{n+1} = x(y_n, h) + \mathbf{O}(h^{r+2})$$

with different local convergence orders, where $x(\cdot, h)$ denotes the exact solution of

$$\forall t \in [0, h], \quad \dot{x}(x_0, t) = f(x(x_0, t)) \quad \text{with} \quad x(x_0, 0) = x_0$$

in dependence on x_0 .

Step-size control

Main idea

- User input: local error tolerance TOL and absolute tolerance ATOL.
- Compute two approximations

$$y_{n+1} = x(y_n, h) + \mathbf{O}(h^{r+1}) \quad \text{and} \quad z_{n+1} = x(y_n, h) + \mathbf{O}(h^{r+2})$$

with different local convergence orders, where $x(\cdot, h)$ denotes the exact solution of

$$\forall t \in [0, h], \quad \dot{x}(x_0, t) = f(x(x_0, t)) \quad \text{with} \quad x(x_0, 0) = x_0$$

in dependence on x_0 .

Step-size control

Main idea (continued)

- Compute componentwise error estimates

$$e_j = \frac{\|y_{n+1,j} - z_{n+1,j}\|}{\text{TOL}|y_{n,j}| + \text{ATOL}}$$

- Determine a new step-size $h \rightarrow \rho \frac{h}{\sqrt[r+1]{\|e\|}}$, e.g. with $\rho = 0.9$.
- Accept the step if $\|e\| \leq 1$ (usually ∞ -norm); reject otherwise.

Step-size control

Main idea (continued)

- Compute componentwise error estimates

$$e_j = \frac{\|y_{n+1,j} - z_{n+1,j}\|}{\text{TOL}|y_{n,j}| + \text{ATOL}}$$

- Determine a new step-size $h \rightarrow \rho \frac{h}{\sqrt[r+1]{\|e\|}}$, e.g. with $\rho = 0.9$.
- Accept the step if $\|e\| \leq 1$ (usually ∞ -norm); reject otherwise.

Step-size control

Main idea (continued)

- Compute componentwise error estimates

$$e_j = \frac{\|y_{n+1,j} - z_{n+1,j}\|}{\text{TOL}|y_{n,j}| + \text{ATOL}}$$

- Determine a new step-size $h \rightarrow \rho \frac{h}{\sqrt[r+1]{\|e\|}}$, e.g. with $\rho = 0.9$.
- Accept the step if $\|e\| \leq 1$ (usually ∞ -norm); reject otherwise.

Example: Fehlberg's Method

The Runge-Kutta-Fehlberg method is given by the tableau

0			
$\frac{1}{2}$	$\frac{1}{2}$		
1	$\frac{1}{256}$	$\frac{255}{256}$	
	$\frac{1}{512}$	$\frac{255}{256}$	$\frac{1}{512}$
	$\frac{1}{256}$	$\frac{255}{256}$	

The two steps have consistency orders 1 and 2, respectively.

Contents

- Nonlinear Differential Equations
- Existence and uniqueness of solutions
- Taylor-Model Based Integrators
- Runge-Kutta Integrators
- **Stability Analysis**
- Multi-Step Methods

Linear Test Problem

An important test problem for numerical integration schemes is given by the linear ODE

$$\dot{x}(t) = f(x(t)) \quad \text{with} \quad f(x) = \lambda x$$

for a parameter $\lambda \in \mathbb{C}$, $\operatorname{Re}(\lambda) < 0$.

- Write the ODE integrator step in the form $y_{k+1} = I(f, h, y_k)$
- If the integrator satisfies

$$\forall h > 0, \quad \|I(f, h, y)\|_2 \leq \|y\|_2$$

for all $\lambda \in \mathbb{C}$ with $\operatorname{Re}(\lambda) < 0$, then the integrator is called A-stable.

Stability of Runge-Kutta Integrators

The integrator step of a Runge-Kutta scheme is given by

$$\begin{aligned}k &= \lambda(y \cdot \mathbf{1} + hAk) \\ I(f, h, y) &= y + hb^T k\end{aligned}$$

It can be written in the form

$$\begin{aligned}I(f, h, y) &= R(h\lambda)y \\ \text{with } R(\alpha) &= 1 + \alpha b^T (I - \alpha A)^{-1} \mathbf{1}\end{aligned}$$

The method is A-stable if $|R(\alpha)| \leq 1$ for all $\alpha \in \mathbb{C}$ with $\operatorname{Re}(\alpha) \leq 0$.

Notice: there is no A-stable explicit Runge-Kutta integrator!

Examples

- The explicit Euler method, $y_{k+1} = y_k + hf(y_k)$, satisfies

$$R(\alpha) = 1 + \alpha .$$

This method is clearly not A-stable.

- The implicit Euler method, $y_{k+1} = y_k + hf(y_{k+1})$, satisfies

$$R(\alpha) = 1 + \frac{\alpha}{1 - \alpha} = \frac{1}{1 - \alpha} .$$

This is an example for an A-stable method.

Examples

- The trapezoidal scheme (order 2) is given by

$$\begin{aligned}k_1 &= f(y_k) \\k_2 &= f\left(y_k + \frac{1}{2}hk_1 + \frac{1}{2}hk_2\right) \\y_{k+1} &= y_k + \frac{h}{2}k_1 + \frac{h}{2}k_2\end{aligned}$$

- The method is A-stable, because we have

$$\begin{aligned}R(\alpha) &= 1 + \begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -\frac{\alpha}{2} & 1 - \frac{\alpha}{2} \end{pmatrix}^{-1} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\&= \frac{2 + \alpha}{2 - \alpha}\end{aligned}$$

L-Stability

- Notice that A -stable methods do not always perform well for very “stiff” test differential equations; that is, for $\operatorname{Re}(\lambda) \ll -1$.
- A method is called L-stable if
 1. it is A -stable and
 2. we additionally have $\lim_{\alpha \rightarrow -\infty} R(\alpha) = 0$.

Examples

- The implicit Euler method, $R(\alpha) = \frac{1}{1-\alpha}$, is L-stable.
- The trapezoidal rule, $R(\alpha) = \frac{2+\alpha}{2-\alpha}$, is A -stable but not L-stable.

Contents

- Nonlinear Differential Equations
- Existence and uniqueness of solutions
- Taylor-Model Based Integrators
- Runge-Kutta Integrators
- Stability Analysis
- **Multi-Step Methods**

Multi-Step Methods

- Taylor model based integration and Runge-Kutta methods construct y_k in dependence on y_{k-1} .
- This is in contrast to multi-step methods, where y_k may depend on $y_{k-1}, y_{k-2}, \dots, y_{k-m}$.
- Of special interest are the so-called linear multi-step methods, which have the form

$$\sum_{i=0}^m \alpha_{m-i} y_{k-i} = h \sum_{i=0}^m \beta_{m-i} f(y_{k-i}) \quad \text{with} \quad a_R = 1$$

- The method is explicit if $\beta_m = 0$. Otherwise, it is implicit.

Multi-Step Methods

- Linear Multi-Step Methods (LMMs) can be constructed by starting with a numerical integration formula,

$$y_{k-s} + \int_{t-sh}^t f(x(\tau)) \, d\tau = y_{k-s} + h \sum_{i=0}^m \beta_{m-i} f(x(t - ih)) + O(h^{q+1}).$$

Thus, if $y_{k-i} \approx x(t - ih)$, we have

$$y_k = y_{k-s} + h \sum_{i=0}^m \beta_{m-i} f(y_{k-i}) \quad \implies \quad y_k \approx x(t) .$$

The consistency order q of this approximation depends merely on the accuracy of the numerical integration formula.

Examples

- Adams-Bashforth formulas (explicit)

$$s = 1, m = 1 : \quad y_k = y_{k-1} + hf(y_{k-1})$$

$$s = 1, m = 2 : \quad y_k = y_{k-1} + h \left(\frac{3}{2}f(y_{k-1}) - \frac{1}{2}f(y_{k-2}) \right)$$

$$\vdots$$

- Adams-Moulton formulas (implicit)

$$s = 1, m = 0 : \quad y_k = y_{k-1} + hf(y_k)$$

$$s = 1, m = 1 : \quad y_k = y_{k-1} + h \left(\frac{1}{2}f(y_k) + \frac{1}{2}f(y_{k-1}) \right)$$

$$\vdots$$

More Examples

- Nyström's method ($s = 2$)

$$y_k = y_{k-2} + 2hf(y_{k-1})$$

- Milne-Simpson method ($s = 2$)

$$y_k = y_{k-2} + h \left(\frac{1}{3}f(y_k) + \frac{4}{3}f(y_{k-1}) + \frac{1}{3}f(y_{k-2}) \right)$$

- It's easy to come up with more methods: any numerical integration formula yields yet another LMM.

Backward Differencing Formulas (BDF)

- Instead of numerical integration, we can also use numerical differentiation to derive LMMs. This yields BDF methods.
- Examples:

$$\begin{aligned}y_k - y_{k-1} &= hf(y_k) \\y_k - \frac{4}{3}y_{k-1} + \frac{1}{3}y_{k-2} &= \frac{2}{3}hf(y_k) \\y_k - \frac{18}{11}y_{k-1} + \frac{9}{11}y_{k-2} - \frac{2}{11}y_{k-3} &= \frac{6}{11}hf(y_k) \\&\vdots\end{aligned}$$

- Exercise: combine numerical differentiation and integration to derive even more LMMs...

Dahlquist's Theorem

Theorem. Let f be globally Lipschitz. The iterates of the LMM

$$\sum_{i=0}^m \alpha_{m-i} y_{k-m} = h \sum_{i=0}^m \beta_{m-i} f(y_{k-i}) \quad \text{with} \quad a_R = 1$$

converge to the solution of the ODE, $\dot{x}(t) = f(x(t))$ for $h \rightarrow 0$, if

1. the initial iterates $y_0, y_1, \dots, y_{m-1}$ converge,
2. the method is consistent (= has at least consistency order 1),

$$\sum_{i=0}^m \alpha_i = 0 \quad \text{and} \quad \sum_{i=0}^m i \alpha_i + \sum_{i=0}^m \beta_i = 0, \quad \text{and}$$

3. the roots $\lambda_j \in \mathbb{C}$ of the first characteristic polynomial

$$\rho(\lambda) = \sum_{i=0}^m \alpha_i \lambda^i$$

satisfy $|\lambda_j| \leq 1$ and all multiple roots satisfy $|\lambda_j| < 1$.