Error Analysis

- Introduction
- Representation of Numbers in a Computer
- Conditioning of Numerical Operations
- Conditioning of Factorable Functions

Contents

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
- numerical errors and condition numbers
- factorable functions and propagation of errors
- numerical and algorithmic differentiation

Contents

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> $(1.1 + 0.1) == 1.2$	julia> $1.1 + 0.1$
`false`	1.2000000000000002

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

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

Floating Point Numbers

IEEE standard for double-precision floating point numbers:

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

Example:

```
julia>bits(3.0)
0100000000001000000000000000000000000000000000000000000000000000
```

The number 3.0 is represented as $+(1+1*2^{-1})*2^{1*2^{10}-1023}$.

Floating Point Numbers

- Numbers with magnitude less than 2^{-1022} are set to zero.
- Numbers greater than $2^{1023}(2 - 2^{-52})$ result in overflow (error).
- Many numbers cannot be represented exactly.
- If the magnitude is not larger than $2^{1023}(2 - 2^{-52})$, the closest representable number is stored (based on rounding).

A closer look at the example

```
julia>bits(1.1)
"0011111111110001100110011001100110011001100110011001100110011010"
julia>bits(0.1)
"0011111110111001100110011001100110011001100110011001100110011010"
julia>bits(1.2)
"0011111111110011001100110011001100110011001100110011001100110011"
julia>bits(1.1+0.1)
"0011111111110011001100110011001100110011001100110011001100110100"
```

Floating Point Numbers

- Numbers between 1 and $1 + 2^{-52}$ cannot be represented.
- The (relative) rounding $\mathrm{eps} = 2^{-52}$ is called *machine precision*.
- The absolute rounding error $\mathrm{eps} * 2^e$ depends on exponent e.

  (if we work with larger numbers, we get larger rounding errors)

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

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.

Contents

Condition number of a scalar operation

Consider a twice continuously differentiable map $\Phi : \mathbb{R} \to \mathbb{R}$.

- $x \in \mathbb{R}$ is the point at which we want to evaluate Φ.
- $\Delta x \in \mathbb{R}$ denotes a (small) numerical error.
- Taylor's theorem yields the first order approximation of the error

$$\Phi(x + \Delta x) - \Phi(x) \approx J \Delta x \quad \text{with} \quad J := \frac{\partial \Phi}{\partial x}(x)$$

Here, $c := |J|$ is can be interpreted as an error amplification factor; also called condition number. If $c \gg 1$, f is called ill-conditioned.

Example

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
```

Problem: The condition number is $c = 10^8 \cos(10^8 \pi) = 10^8$.

Recall: the error of storing π is in the order of $\text{eps} \approx 2 * 10^{-16}$.

Condition Number of Vector-Valued Functions

For a twice continuously differentiable function $\Phi : \mathbb{R}^n \to \mathbb{R}^m$ the first order expansion is

$$\Phi(x + \Delta x) - \Phi(x) \approx J * \Delta x \ , \quad \text{where } \ J := \frac{\partial \Phi}{\partial x}(x)$$

denotes the Jacobian matrix, $J \in \mathbb{R}^{m \times n}$.

If m and or n are large, "printing and inspecting" J may not be practical. In this case, it is helpful to define a condition number as

$$c \ := \ \|J\| \ = \ \max_{\Delta x} \frac{\|J * \Delta x\|}{\|\Delta x\|}$$

for a suitable vector norm $\| \cdot \|$. If $c \gg 1$, Φ is ill-conditioned.

Contents

Factorable Functions

Many (but not all) functions of our interest can be composed into a finite list of atom operations from a given library L,

e.g., $L = \{+, -, *, \sin, \cos, \log, \ldots\}$.

Example

- The function $f(x) = \sin(x_1 * x_2) + \cos(x_1)$ will (internally) be evaluated as

$$\begin{aligned} a_1 &= x_1 * x_2 \\ a_2 &= \sin(a_1) \\ a_3 &= \cos(x_1) \\ a_4 &= a_2 + a_3 \\ f(x) &= a_4 \ . \end{aligned}$$

  Here, the memory for $a_1, \ldots a_4$ is (usually) allocated temporarily.

Factorable Functions

In general, we may write the algorithm for evaluating a factorable function (in one variable $x \in \mathbb{R}$) in the form

$$a_0 = x \ , \ a_1 = \phi_1(a_0) \ , \ \ldots \ , \ f(x) = a_N = \phi_N(a_0, \ldots, a_{N-1}) \ .$$

In the worst case, the numerical errors associated with evaluating the atom operators $\phi_1, \ldots, \phi_N$ may add up and lead to a potentially large evaluation error Δa_N:

$$\begin{array}{rcl}
\Delta a_0 & \approx & \text{eps} \\
\Delta a_1 & \approx & \left| \frac{\partial \phi_1}{\partial a_0}(a_0) \right| * \Delta a_0 + \text{eps} \\
\Delta a_2 & \approx & \left| \frac{\partial \phi_2}{\partial a_0}(a_0, a_1) \right| * \Delta a_0 + \left| \frac{\partial \phi_2}{\partial a_1}(a_0, a_1) \right| * \Delta a_1 + \text{eps} \\
 & \vdots & \\
\Delta a_N & \approx & \sum_{i=0}^{N-1} \left| \frac{\partial \phi_N}{\partial a_i}(a_0, \ldots, a_{N-1}) \right| * \Delta a_i \ + \text{eps}.
\end{array}$$

Finite Differences

The derivative of a twice continuously differentiable function $f : \mathbb{R} \to \mathbb{R}$ 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 $h \to 0$.

- The numerical error is approximately

$$\frac{\max\{|\frac{\partial f}{\partial x}(x)|, 1\} * \mathrm{eps}}{h} = \mathbf{O}\left(\frac{\mathrm{eps}}{h}\right)$$

- If f is well conditioned, $\frac{1}{2} \left| \frac{\partial^2 f}{\partial x^2}(x) \right| \approx \left| \frac{\partial f}{\partial x}(x) \right| \approx 1$, we choose

$$h \approx \mathsf{argmin}_h \left(h + \frac{\mathrm{eps}}{h}\right) = \sqrt{\mathrm{eps}}\,.$$

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

$$\frac{\max\{|\frac{\partial f}{\partial x}(x)|, 1\} * \mathrm{eps}}{h} = \mathbf{O}\left(\frac{\mathrm{eps}}{h}\right)$$

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

Algorithmic Differentiation

In modern computer programs, algorithmic differentiation (AD) is used in order to avoid discretization errors. Let's try to understand the main idea of foward AD by looking at an example:

$$
\begin{array}{rcl|rcl}
a_0 & = & x & b_0 & = & 1 \\
a_1 & = & a_0 * a_0 & b_1 & = & a_0 * b_0 + b_0 * a_0 \\
a_2 & = & \sin(a_1) & b_2 & = & \cos(a_1) * b_1 \\
a_3 & = & a_1 + a_2 & b_3 & = & b_1 + b_2 \\
f(x) & = & a_3 \ . & f'(x) & = & b_3 \ .
\end{array}
$$

In practice, this is usually implemented by operator overloading.

Summary

- Programs often store numbers with finite precision only.
- IEEE double precision floating point numbers: $\mathrm{eps} \approx 2 * 10^{-16}$.
- The propagation of small numerical errors can be analyzed approximately using first order Taylor approximations.
- For factorable function the errors of each iteration may add up.
- Numerical differentiation is in general less accurate than algorithmic differentiation (AD).

Polynomial Interpolation

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

Contents

Polynomial Interpolation

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

$$p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$$

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

Application Examples:

- We have a (smooth) function $f : \mathbb{R} \to \mathbb{R}$. 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 $[0°, 10°]$. 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 $x_0, \ldots, x_n$ are equal, there exists a unique sequence of coefficients $a_0, \ldots, a_n$ such that $p(x_i) = y_i$ for all $i \in \{0, \ldots, n\}$, where

$$p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n \ .$$

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 $i \in \{0, \ldots, n\}$.

Important Property:

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

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

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 $p(x_1) = y_1$.

Existence and Uniqueness of Solutions

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

$$p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n \ .$$

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 requirments; 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 $x_0, \ldots, x_n$ are equal, there exists a unique sequence of coefficients $a_0, \ldots, a_n$ such that $p(x_i) = y_i$ for all $i \in \{0, \ldots, n\}$, where

$$p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n \ .$$

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 $i \in \{0, \ldots, n\}$. The function $r(x) = p(x) - q(x)$ satisfies

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

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), ..., (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

$$\begin{aligned} y_0 = p(x_0) &= b_0 \\ y_1 = p(x_1) &= b_0 + b_1(x_1 - x_0) \\ &\vdots \\ y_n = p(x_n) &= b_0 + b_1(x_n - x_0) + \ldots + b_n(x_n - x_0)\ldots(x_n - x_{n-1}) \end{aligned}$$

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

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 $(x_0, y_0), \ldots, (x_n, y_n)$ are given data points and f and g functions that satisfy

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

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 $i \in \{0, \ldots, n\}$.

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

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

- The constant functions $p_{0,0}(x) = y_0$, $p_{11}(x) = y_1$, and $p_{22}(x) = y_2$ interpolate the first, second, and third data point, respectively.
- The polynomial $p_{0,1}(x) = p_{0,0}(x) + \frac{p_{11}(x) - p_{00}(x)}{x_1 - x_0}(x - x_0)$ satisfies $p_{0,1}(x_i)) = y_i$ for $i \in \{0, 1\}$.
- The polynomial $p_{1,2}(x) = p_{1,1}(x) + \frac{p_{22}(x) - p_{11}(x)}{x_2 - x_1}(x - x_1)$ satisfies $p_{0,1}(x_i)) = y_i$ for $i \in \{1, 2\}$.
- The polynomial $p(x) = p_{0,1} + \frac{p_{12}(x) - p_{01}(x)}{x_2 - x_0}(x - x_0)$ satisfies $p(x_i) = y_i$ for $i \in \{0, 1, 2\}$.

Divided Differences

In general, Neville's recursion is initialized with

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

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 $k \in \{1, \ldots, n-i\}$ 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

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

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

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

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

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

Proof: We have

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

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

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:

$$\begin{array}{c|c|cc} x_0 & y_0 & d_{01} & d_{02} \\ x_1 & y_1 & d_{12} & \\ x_2 & y_2 & & \end{array}$$

... $d_{02} = \frac{d_{12}-d_{01}}{x_2-x_0}$.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:

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

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

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:

$$\begin{aligned} b_n &= d_{0n} \\ b_k &= d_{0k} + (x - x_k) b_{k+1} \quad k = n-1, \ldots, 0 \\ p(x) &= b_0. \end{aligned}$$

Contents

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

$$y_i = f(x_i) \;, \quad i \in \{1, \ldots, n\} \;,$$

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

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

for a $\xi_x \in [\min_i x_i, \max_i x_i]$.

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

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

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

which converges to 0 for $n \to \infty$ 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 (see Homework 2 for details).

Contents

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 $f'(x)$:

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

For $h \to 0$ 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

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

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

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

Hermite Interpolation

Hermite's interpolation problem is to find a polynomial of degree $\sum_{i=0}^{n} m_i$ satisfying the condition

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

for all $i \in \{0, \ldots, n\}$ 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

$$\begin{array}{c|c|c|c|c} a & f(a) & f'(a) & \frac{f(b)-f(a)-f'(a)(b-a)}{(b-a)^2} & \frac{2(f(b)-f(a))+(f'(a)+f'(b))(b-a)}{(b-a)^3} \\ a & f(a) & \frac{f(b)-f(a)}{b-a} & \frac{f(a)-f(b)+f'(b)(b-a)}{(b-a)^2} & \\ b & f(b) & f'(b) & & \\ b & f(b) & & & \end{array}$$

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

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

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 $(x_0, y_0), \ldots, (x_n, y_n)$, 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 $n \to \infty$.
- Hermite interpolation additionally interpolates derivatives.

Applications of Polynomial Interpolation

- Extrapolation

- Splines

Contents

- Extrapolation
- Splines

Problem Formulation

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

$$\lim_{h \to 0} f(h) \, .$$

Example: We want to evaluate the expression

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

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

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

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

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

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

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

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

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

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

$$(h_1, f(h_1)), \quad (h_2, f(h_2)), \quad (h_3, f(h_3)), \quad \ldots$$

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

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

$$(h_1, f(h)),\ (h_2, f(h_2)),\ \ldots, (h_n, f(h_n))$$

for small $h \geq h_1 > h_2 > \ldots > h_n > 0$, 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 $n \to \infty$.

Example: Interpolation of

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

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.

Cubic Splines using Hermite Interpolation

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

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

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

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

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

for all $i \in \{0, \ldots, n\}$. 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 $p(x_i) = f(x_i)$, $i \in \{0, \ldots, n\}$, ($2n$ 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 ($i \in \{1, \ldots, n\}$):

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

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

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

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

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

4. Boundaries: $c_n = 0 \qquad c_1 - 3d_1 h = 0.$

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 $c_0, \ldots, c_n$.

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:

$$\begin{aligned}
&\tfrac{f(x_i)-f(x_{i-1})}{h} + \tfrac{2c_i+c_{i-1}}{3}h \\
&= \tfrac{f(x_{i+1})-f(x_i)}{h} + \tfrac{2c_{i+1}+c_i}{3}h - 2c_{i+1}h + (c_{i+1} - c_i)h \\
&\Longrightarrow h(c_{i-1} + 4c_i + c_{i+1}) = r_i
\end{aligned}$$

for $i = 1, \ldots, n-1$ and $c_0 = c_n = 0$ as well as the short-hand

$$r_i = \frac{3}{h}\left(f(x_{i+1}) - 2f(x_i) + f(x_{i-1})\right) \ .$$

Natural Cubic Splines

Thus, the equation system for $c_1, \ldots, c_{n-1}$ can be written in the form

$$\begin{pmatrix} 4h & h & & \ldots & 0 \\ h & 4h & h & & \\ & \ddots & \ddots & \ddots & \\ \vdots & & h & 4h & h \\ 0 & & & h & 4h \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n-2} \\ c_{n-1} \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_{n-2} \\ r_{n-1} \end{pmatrix}$$

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

$$\int_{x_{\min}}^{x_{\max}} |p''(x)|^2 \, \mathrm{d}x \leq \int_{x_{\min}}^{x_{\max}} |f''(x)|^2 \, \mathrm{d}x \; .$$

"The natural cubic spline p never oscillates more than the function f."

Proof: Any twice continuously differentiable function w with $w(x_{\min}) = w(x_{\max}) = 0$ satisfies

$$\begin{aligned} \int_{x_{\min}}^{x_{\max}} p''(x) w''(x) \mathrm{d}x &= \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} p''(x) w''(x) \mathrm{d}x \\ &= \sum_{i=0}^{n-1} \left\{ p'' w' |_{x_i}^{x_{i+1}} - p''' w |_{x_i}^{x_{i+1}} \right\} = 0 \; . \end{aligned}$$

Here, we have used that $p''(x_{\min}) = p''(x_{\max})$ as well as $p'''' \equiv 0$, as p is piecewise cubic. For $w = f - p$ we find

$$\int_{x_{\min}}^{x_{\max}} |f''(x)|^2 \, \mathrm{d}x = \int_{x_{\min}}^{x_{\max}} |p''(x) + w''(x)|^2 \, \mathrm{d}x \geq \int_{x_{\min}}^{x_{\max}} |p''(x)|^2 \, \mathrm{d}x \; .$$

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.

Trigonometric Interpolation

- Introduction to Trigonometric Interpolation
- Background on Complex Analysis
- Discrete Fourier Analysis
- Fast Fourier Transform

Contents

Problem Formulation

Periodic functions occur naturally in many applications, e.g.,

- Power generating devices, e.g., windmills, wave-power devices, ...
- Thermodynamic converters, e.g., periodic Carnot processes ...
- Robotics/human motions: walking, swimming, ...

A function $f : \mathbb{R} \to \mathbb{R}$ is called periodic with period T if

$$f(x + T) = f(x) \quad \text{for all} \quad x \in \mathbb{R} \, .$$

Problem Formulation

Assuming a function/a given sequence of data points is periodic with known period length T, we are interested in interpolating it with trigonometric sums of the form

$$p(x) = \frac{1}{2}a_0 + \sum_{k=1}^{m} \left(a_k \sin\left(\frac{2\pi kx}{T}\right) + b_k \cos\left(\frac{2\pi kx}{T}\right)\right)$$

at the equidistant points

$$x_k = k\frac{T}{n+1}\ , \quad k \in \{0, \ldots, n\} \quad \text{with} \quad n = 2m\ .$$

Contents

Basic Complex Analysis

Recall from complex analysis that we have

$$e^{ix} = \cos(x) + i\sin(x) \quad \text{with} \quad i = \sqrt{-1}\,.$$

In particular, $e^{\pi i} = -1$ and $e^{2\pi i} = 1$. The other way around, we can also express the functions $\sin$ and $\cos$ as

$$\sin(x) = \frac{e^{ix} - e^{-ix}}{2i} \quad \text{and} \quad \cos(x) = \frac{e^{ix} + e^{-ix}}{2}\,.$$

Unit Roots

The $(n+1)$-th unit root is defined as

$$\omega = e^{\frac{2\pi i}{n+1}}$$

It satisfies

$$1 = \omega^{n+1} \quad \Leftrightarrow \quad (1-\omega)\left(1+\omega+\ldots+\omega^{n}\right) = 0\,.$$

For $n > 0$ we have $\omega \neq 1$ and consequently

$$1+\omega+\ldots+\omega^{n} = 0\,.$$

A simple generalization of this equation is that we also have

$$1+\omega^{k}+\omega^{2k}+\ldots+\omega^{kn} = 0 \quad \text{for all } \; k \not\equiv 0 \bmod (n+1)\,.$$

Sylvester's unitary matrix

We can use $\omega = e^{\frac{2\pi i}{n+1}}$ to construct a matrix M with components

$$M_{j,k} = \omega^{j \cdot k} \ , \quad j,k \in \{0, \ldots, n\}$$

The matrix M satisfies $M^T M^* = (n+1) \cdot I$, since

$$\sum_{l=0}^{n} M_{l,j} M^*_{l,k} \ = \ 1 + \omega^{j-k} + \ldots + \omega^{(j-k)n} \ = \ (n+1)\delta_{k,j} \ .$$

Notation: M^* = complex conjugate of M, $\delta_{k,j}$ = Kronecker's symbol, and I = unit matrix.

We may call the matrix $U = \frac{1}{\sqrt{n+1}} M^T$ Sylvester's unitary matrix.

Contents

Trigonometric Interpolation

Before we come back to the real-valued trigonometric interpolation problem, we first try to construct a complex valued function

$$p(x) = \sum_{k=0}^{n} c_k e^{ikx} \ ,$$

which interpolates the points $(x_0, y_0), \ldots, (x_n, y_n)$ with $y_0, \ldots, y_n \in \mathbb{C}$ and

$$x_k = \frac{2\pi k}{n+1} \ , \quad k \in \{0, \ldots, n\} \ .$$

(We simply assume $T = 2\pi$, as we may re-scale the x-axis otherwise)

Trigonometric Interpolation

The interpolation conditions can be written in the form

$$y_j = p(x_j) = \sum_{k=0}^{n} c_k e^{\frac{2\pi i}{n+1} kj} = \sum_{k=0}^{n} \omega^{j \cdot k} c_k = \sum_{k=0}^{n} M_{k,j} c_k \ .$$

Using the vector notation $y = [y_0, \ldots, y_n]^T$ and $c = [c_0, \ldots, c_n]^T$ this relation can be summarized as

$$y = M^T c \ .$$

As we have $M^T M^* = (n+1)I$, this equation has the unique solution

$$c = \frac{1}{n+1} M^* y \quad \Leftrightarrow \quad c_j = \frac{1}{n+1} \sum_{k=0}^{n} y_k e^{-ikx_j} \ .$$

Discrete Fourier Transform

Theorem

Let $x_j = \frac{2\pi j}{n+1}$ be an equidistant mesh of the interval $[0, 2\pi]$. There exists exactly one function of the form

$$p(x) = \sum_{k=0}^{n} c_k e^{ikx} \ ,$$

which satisfies the interpolation conditions $p(x_j) = y_j$ for all $j \in \{1, \ldots, n\}$. Its coefficients are given by

$$c_j = \frac{1}{n+1} \sum_{k=0}^{n} y_k e^{-ikx_j} \quad \text{for all } j \in \{1, \ldots, n\} \ .$$

Discrete Fourier Transform

For $n = 2m$ we can exploit the periodicity relation $c_{n+1-k} = c_{-k}$ to write the interpolating function at all points x_j in the form

$$\begin{aligned} p(x_j) &= \sum_{k=0}^{n} c_k e^{ikx_j} \\ &= c_0 + \sum_{k=1}^{m} c_k e^{ikx_j} + \sum_{k=1}^{m} c_{-k} e^{-ikx_j} \\ &= c_0 + \sum_{k=1}^{m} (c_k + c_{-k}) \cos(kx_j) + i(c_k - c_{-k}) \sin(kx_j) \ , \end{aligned}$$

which yields the coefficients

$$\begin{aligned} a_k &= c_k + c_{-k} &&= \tfrac{2}{n+1} \sum_{j=0}^{n} y_j \cos(jx_k) \\ b_k &= i(c_k - c_{-k}) &&= \tfrac{2}{n+1} \sum_{j=0}^{n} y_j \sin(jx_k) \end{aligned}$$

of the interpolating trigonometric function (only needed if we want to avoid complex numbers).

Discrete Fourier Transform

Theorem

Let $x_j = \frac{2\pi j}{n+1}$ be an equidistant mesh of the interval $[0, 2\pi]$ and $n = 2m$.

There exists exactly one function of the form

$$p(x) = \frac{1}{2} a_0 + \sum_{k=1}^{m} \left(a_k \sin(kx) + b_k \cos(kx) \right) \ ,$$

which satisfies the interpolation conditions $p(x_j) = y_j$ for all

$j \in \{1, \ldots, n\}$. Its coefficients are given by

$$a_k = \frac{2}{n+1} \sum_{j=0}^{n} y_j \cos(jx_k) \quad \text{and} \quad b_k = \frac{2}{n+1} \sum_{j=0}^{n} y_j \sin(jx_k) \ .$$

Contents

Complexity of Computing a Discrete Fourier Transform

If we evaluate all coefficients of a discrete Fourier transform directly from the formula

$$c_j = \frac{1}{n+1} \sum_{k=0}^{n} y_k \, e^{-ikx_j}$$

we will need $\mathbf{O}(n^2)$ operations.

Fortunately, this computation can be speed up significantly achieving a computational complexity of $\mathbf{O}(n \log(n))$. Algorithms which can compute the coefficients with this reduced complexity are called "Fast Fourier Transform (FFT)" algorithms.

Main Idea of FFT

The main idea of fast fourier transform is to recursively break up the whole sum into sums with even and odd indices (assume $n+1 = 2^p$):

$$c_j = \sum_{k=0}^{2^p-1} y_k \omega^{kj} = \underbrace{\sum_{k=0}^{2^{p-1}-1} y_{2k} (\omega^2)^{kj}}_{E_j} + \omega^j \underbrace{\sum_{k=0}^{2^{p-1}-1} y_{2k+1} (\omega^2)^{kj}}_{O_j}$$

Due to periodicity we have $E_j = E_{2^{p-1}+j}$ and $O_j = O_{2^{p-1}+j}$. Now, we can write the coefficients in the form

$$\begin{aligned} c_j &= E_j + \omega^j O_j \\ c_{2^{p-1}+j} &= E_j - \omega^j O_j \end{aligned} \qquad \text{f.a.} \quad j \in \{0, \ldots, 2^{p-1}-1\} \, .$$

This procedure can be repeated for E_j and O_j.

Summary

- Trigonometric Interpolation on equidistant grid

  = Discrete Fourier Transform (DFT).

- The coefficients in the complex Fourier series $\sum_{k=0}^{n} c_k e^{ikx}$ can be computed as

$$c_j = \frac{1}{n+1} \sum_{k=0}^{n} y_k e^{-ikx_j} \ .$$

- Fast Fourier Transform (FFT) can reduce the complexity of computing all coefficients to $\mathbf{O}(n \log(n))$.

Gauss Approximation

- Problem Formulation
- Gram-Schmidt Algorithm
- Orthogonal Polynomials
- Solution of Gauss' Approximation Problem

Contents

Problem Formulation

Gauss' approximation problem is to construct a polynomial p of degree $\leq n$ which solves

$$\min_{p \in P_n} \|f - p\| \quad \text{with} \quad \|g\| = \sqrt{\int_a^b g(x)^2 \, \mathrm{d}x} \, .$$

denoting the L_2-norm. Here P_n denotes the set of polynomials $p : \mathbb{R} \to \mathbb{R}$ with degree $\leq n$ and $f : \mathbb{R} \to \mathbb{R}$ is a given function.

L_2-Scalar Products

Recall that the L_2-scalar product of two functions $f, g : [a, b] \to \mathbb{R}$ on an interval $[a, b]$ is given by

$$\langle f, g \rangle = \int_a^b f(x) g(x) \, \mathrm{d}x \; .$$

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:

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

$$0 = \frac{\partial}{\partial t} \|f - p - tq\|^2 \Big|_{t=0} = \langle f - p, q \rangle \ .$$

Optimality Conditions

Theorem The polynomial p is a solution of the minimization problem

$$\min_{p \in P_n} \|f - p\| \ .$$

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

$$\|f - p\|^2 \ = \ \langle f - p, f - q \rangle + \langle f - p, q - p \rangle \leq \|f - p\| \, \|f - q\|$$

and thus $\|f - p\| \leq \min_{q \in P_n} \|f - q\|$, i.e., p is a minimizer. $\square$

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

$$\langle f - p_1, q \rangle = \langle f - p_2, q \rangle = 0 \quad \text{for all} \quad q \in P_n \ ,$$

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 $a_1, \ldots, a_k \in \mathbb{R}^n$. Gram-Schmidt's algorithm can be used to check for linear independence:

Gram-Schmidt Algorithm:

For $i = 1, \ldots, k$:

- Orthogonalization. $\overline{q}_i = a_i - \langle q_1, a_i \rangle q_1 - \ldots - \langle q_{i-1}, a_i \rangle q_{i-1}$.
- Test for dependence. If $\overline{q}_i = 0$, quit.
- Normalization. $q_i = \frac{\overline{q}_i}{\|\overline{q}_i\|}$.

If the algorithm does not quit, the vectors a_i are linearly independent.

Gram-Schmidt Algorithm

The Gram-Schmidt Algorithm computes the vectors $q_1, \ldots, q_k$. These vectors are orthonormal. This can be proven by induction:

- The vector $q_1 = \frac{a_1}{\|a_1\|}$ is normalized.
- Assume the vectors $q_1, \ldots, q_{i-1}$ are already orthonormal. Then, the vector $\overline{q}_i$ satisfies

$$\begin{aligned} \langle \overline{q}_i, q_j \rangle &= \langle a_i, q_j \rangle - \sum_{k=1}^{i-1} \langle q_k, a_i \rangle \langle q_k, q_j \rangle \\ &= \langle a_i, q_j \rangle - \sum_{k=1}^{i-1} \langle q_k, a_i \rangle \delta_{k,j} = 0 \end{aligned}$$

  for all $j \in \{1, \ldots, i-1\}$, i.e., the vectors $q_1, \ldots, q_i$ are orthonormal.

Contents

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 $a_0(x) = 1$, $a_1(x) = x$, $a_2(x) = x^2$, ..., $a_n = x^n$:

- $q_0(x) = \sqrt{\frac{1}{2}}$.
- $q_1(x) = \sqrt{\frac{3}{2}}x$.
- $q_2(x) = \sqrt{\frac{5}{8}}(3x^2 - 1)$.
- ...
- $q_n(x) = \sqrt{\frac{2n+1}{2}} \frac{1}{2^n n!} \frac{\partial^n}{\partial x^n} \left(x^2 - 1\right)^n$. (Exercise)

Legendre Polynomials

The orthogonal polynomials

$$L_n(x) = \frac{1}{2^n n!} \frac{\partial^n}{\partial x^n} \left(x^2 - 1\right)^n .$$

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 $q_0, \ldots, q_n$,

$$p(x) = \sum_{i=0}^{n} c_i \, q_i(x) \; .$$

The coefficients $c_0, \ldots, c_n$ can be found by substituting the orthogonal polynomials in the optimality condition

$$\forall q \in P_n, \qquad \langle f - p, q \rangle = 0 \; .$$

This yields

$$c_i = \langle p, q_i \rangle = \langle f, q_i \rangle$$

for all $i \in \{1, \ldots, n\}$.

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 $q_0, \ldots, q_n \in P_n$, 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 $c_0, \ldots, c_n$ are given by

$$\forall i \in \{0, \ldots, n\}, \qquad c_i = \langle p, q_i \rangle = \langle f, q_i \rangle \ .$$

Numerical Integration

- Problem Formulation
- Lagrange Quadrature
- Gauss Quadrature

Contents

Problem Formulation

We are interested in computing integrals of the form

$$\int_a^b f(x)\,\mathrm{d}x$$

for given functions $f : \mathbb{R} \to \mathbb{R}$.

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

$$\int_a^b f(x)\,\mathrm{d}x = \int_a^b \sum_{i=0}^{n} c_i x^i\,\mathrm{d}x$$

can be computed explicitly. We find

$$\int_a^b f(x)\,\mathrm{d}x = \int_a^b \sum_{i=0}^{n} c_i x^i\,\mathrm{d}x = \sum_{i=0}^{n} \frac{c_i}{i+1}\left(b^{i+1} - a^{i+1}\right)\ ,$$

which can be evaluated with Horner's algorithm.

Contents

Integration using Lagrange Interpolation

Since we know how to integrate polynomials, one strategy to approximate the general integral

$$\int_a^b f(x)\,\mathrm{d}x$$

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 $x_0, \ldots, x_n \in [a, b]$ 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:

$$x_i = a + i * H, \quad i = 0, \ldots, n, \quad \text{with} \quad H = \frac{b-a}{n} \ .$$

- Open Newton-Cotes methods choose:

$$x_i = a + (i+1) * H, \quad i = 0, \ldots, n, \quad \text{with} \quad H = \frac{b-a}{n+2} \ .$$

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

$$\alpha_i = \int_0^n \prod_{j=0, j\neq i}^{n} \frac{t - j}{i - j} \, \mathrm{d}t$$

can be computed "once and forever".The integral approximation is then given by

$$\int_a^b f(x)\mathrm{d}x \approx \sum_{i=0}^{n} H\alpha_i f(x_i) \; .$$

Example: Simpson's rule

For $n = 2$ the coefficients of the closed Newton-Cotes methods are

- $\alpha_0 = \int_0^2 \frac{t-1}{0-1}\frac{t-2}{0-2}\,\mathrm{d}t = \frac{1}{3}$.
- $\alpha_1 = \int_0^2 \frac{t-0}{1-0}\frac{t-2}{1-2}\,\mathrm{d}t = \frac{4}{3}$.
- $\alpha_2 = \int_0^2 \frac{t-0}{2-0}\frac{t-1}{2-1}\,\mathrm{d}t = \frac{1}{3}$.
- The corresponding approximation formula

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{H}{3}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)$$

with $H = \frac{b-a}{2}$ is called Simpson's rule.

Integration Error

Since we approximate the function f with a polynomial first, we will obtain an integration error, given by

$$\left| \int_a^b (f(x) - p(x)) \mathrm{d}x \right| \leq \frac{1}{(n+1)!} \int_a^b \frac{\partial^{n+1} f(\xi(x))}{\partial x^{n+1}} \prod_{j=0}^{n} (x - x_j) \, \mathrm{d}x \; .$$

For Simpson's formula this error estimate can be worked out further finding

$$\left| \int_a^b (f(x) - p(x)) \mathrm{d}x \right| \leq \frac{(b-a)^5}{2880} \max_{\xi \in [a,b]} f^{(4)}(\xi) \; .$$

Contents

- Problem Formulation
- Lagrange Quadrature
- Gauss Quadrature

Maximum Order of Integration Formulas

What is the maximum order of integration formulas of the form

$$I(f) = \int_a^b f(x)\,\mathrm{d}x \approx I_n(f) = \sum_{i=0}^{n} \alpha_i f(x_i) \ ?$$

Can we choose the points $x_0, \ldots, x_n \in [a, b]$ 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

$$p(x) = \prod_{i=0}^{n} (x - x_i)^2 \ .$$

If we had a interpolation formula with order larger than $2n + 2$ it would be exact for $p(x)$, i.e.,

$$0 = I_n(p) = I(p) = \int_a^b p(x) \, \mathrm{d}x > 0 \ .$$

This is a contradiction! Thus, $m = 2n + 2$ is an upper bound on the order.

Gauss Quadrature Rule: Main Idea

How can we construct an interpolation formula with order $2n+2$?

In the following, we use the devided difference notation

$$f[x_0, \ldots, x_n] = \sum_{i=0}^{n} f(x_i) \prod_{j=0, j\neq i}^{n} \frac{1}{x_i - x_j} \, .$$

The interpolation based integration formula for $2n+2$ points,

$x_0, \ldots, x_n, x_{n+1}, \ldots, x_{2n+1}$ can now be written as

$$\begin{aligned} I_{2n+1}(f) &= \sum_{i=0}^{2n+1} f[x_0, \ldots, x_i] \int_a^b \prod_{j=0}^{i-1} (x - x_j) \, \mathrm{d}x \\ &= I_n(f) + \sum_{i=n+1}^{2n+1} f[x_0, \ldots, x_i] \int_a^b \prod_{j=0}^{i-1} (x - x_j) \, \mathrm{d}x \end{aligned}$$

Gauss Quadrature Rule: Main Idea

The integral term in the equation

$$I_{2n+1}(f) \quad = \quad I_n(f) + \sum_{i=n+1}^{2n+1} f[x_0, \ldots, x_i] \int_a^b \prod_{j=0}^{i-1} (x - x_j)\, \mathrm{d}x$$

can be written in the form

$$\int_a^b \prod_{j=0}^{i-1} (x - x_j)\, \mathrm{d}x = \int_a^b \underbrace{\prod_{j=0}^{n} (x - x_j)}_{\in P_{n+1}} \underbrace{\prod_{j=n+1}^{i-1} (x - x_j)}_{\in P_n}\, \mathrm{d}x \; .$$

Thus, if we succeed in choosing $x_0, \ldots, x_n$ such that

$$\int_a^b \prod_{j=0}^{n} (x - x_j)\, q(x)\, \mathrm{d}x = 0 \quad \text{for all} \;\; q \in P_n \; ,$$

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 $x_0, x_1, \ldots, x_n$ such that we have

$$L_{n+1}(x_i) = 0$$

with L_{n+1} being the $(n+1)$-th Legendre polynomial we must have

$$\int_{-1}^{1} \underbrace{\prod_{j=0}^{n}(x - x_j)}_{\sim L_{n+1}(x)} q(x)\,\mathrm{d}x = 0 \quad \text{for all} \quad q \in P_n \ ,$$

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 $q \perp L_{n+1}$.

Gauss Quadrature

If $x_0, \ldots, x_n$ 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 $\alpha_i = \int_a^b \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} \, \mathrm{d}t$ has order $2n+2$, i.e.,

$$|I(f) - I_n(f)| \leq \mathbf{O}\left((a-b)^{2n+2}\right) \ .$$

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

$$\int_{-1}^{1} f(x)\,\mathrm{d}x \;\approx\; f\left(-\sqrt{\frac{1}{3}}\right) + f\left(-\sqrt{\frac{1}{3}}\right)\;,$$

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 $\alpha_i = \int_0^n \prod_{j=0, j\neq i}^{n} \frac{t-j}{i-j} \, \mathrm{d}t$ 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.