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

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

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:

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

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:

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

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

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 $+(1 + \mathbf{1} * 2^{-1}) * 2^{\mathbf{1} * 2^{10} - 1023}$.

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).

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”

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"

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”

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

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.

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

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.

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.

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 $\mathrm{eps} \approx 2 * 10^{-16}$.

Example

Let us evaluate the function

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

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

julia>$\sin(10\hat{}8\,\mathrm{pi})$
$-3.9082928156687315e-8$

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 $\mathrm{eps} \approx 2 * 10^{-16}$.

Example

Let us evaluate the function

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

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

$$\boxed{\begin{array}{l}\text{julia>}\sin(10\hat{\ }8\,\mathrm{pi}) \\ -3.9082928156687315e-8\end{array}}$$

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 $\mathrm{eps} \approx 2 * 10^{-16}$.

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.

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

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:

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

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:

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

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

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

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

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

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

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

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 $h \approx \sqrt[3]{\mathrm{eps}}$.

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 $h \approx \sqrt[3]{\mathrm{eps}}$.

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 $h \approx \sqrt[3]{\mathrm{eps}}$.

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

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

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

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.

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: $\text{eps} \approx 2 * 10^{-16}$.
- 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).