ACM40290: Numerical Algorithms

Accuracy of Numerical Approximations

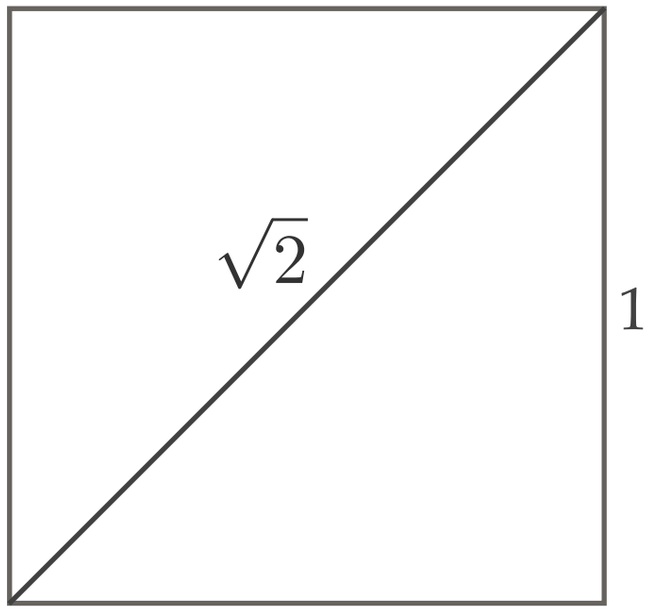
Dr Barry Wardell
School of Mathematics and Statistics
University College Dublin

Approximation

- Approximation pervades all parts of numerical computation
- Space is limited
- Time is finite

Finite Space

- Pythagoras' constant, $d = \sqrt{2}$, is the length of the diagonal of the unit square
- $\sqrt{2}$ cannot be expressed at the ratio of two integers and is therefore called *irrational*.



- First 100 digits:

  1.41421356237309504880168872420969807856967187537694807317667973799073247846210703885038753432764157 3…

- It is obvious that we cannot hope to represent $\sqrt{2}$ **exactly** in finite memory (space). This is one of the reasons why most numerical computations are **approximate**.

Finite Time

- The function e^x can be represented by the power series

$$e^x = 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^k}{k!} + \cdots$$

- which converges for all values of x. Letting $x=1$, we get

$$e = 1 + 1 + \frac{1}{2!} + \cdots + \frac{1}{k!} + \cdots$$

- All numbers in this formula are rationals and we use +, *, and / to calculate it. We can calculate this formula using the recurrence

$$S_k := S_{k-1} + \frac{1}{k!} \quad \text{for} \quad k = 2, 3, \ldots \quad \text{with} \quad S_1 = 2$$

Finite Time

The first 6 terms of this recurrence are

k	1	2	3	4	5	6
S_k	2	5/2	8/3	65/24	163/60	1957/720
fl(S_k)	2	2.5	2.66667	2.70833	2.71667	2.71806

Even if we assume that all results can be calculated and stored exactly we still have a problem: we must **truncate** the infinite series for some finite k, i.e., after a finite time. In fact this recurrence is calculating increasingly accurate rational approximations to the transcendental number e. The numbers fl(S_k) are floating point approximations to the rationals S_k .

Numerical vs Symbolic Calculations

Say we want to compute the roots of the following polynomial

$$x^3 + x^2 + x + 1 = 0$$

In MATLAB the function `roots` gives a **numerical** result

```
>> x = roots([1 1 1 1])
x =
   -1.0000 + 0.0000i
    0.0000 + 1.0000i
    0.0000 - 1.0000i
```

Maple or Mathematica will give an **exact** result

```
Solve[x^3 + x^2 + x + 1 == 0, x]
{{x -> -1}, {x -> -I}, {x -> I}}
```

Numerical vs Symbolic Calculations

Now let's look at a slightly different polynomial

$$x^3 + x^2 + x + 1 + 10^{-7} = 0$$

In MATLAB the function `roots` gives a **numerical** result

```
>> x = roots([1 1 1 1+10^-7])
   x =      -1.00000005 + 0i
2.499999848804e-08 + 1.000000025i
2.499999848804e-08 - 1.000000025i
```

Maple or Mathematica will give an **exact** result

```
Solve[x^3 + x^2 + x + 1 + 10^-7 == 0, x]
```

Numerical vs Symbolic Calculations

Maple or Mathematica will give an **exact** result

$$\frac{1}{300}\left(-100-20000\ 2^{2/3}\left(\frac{5}{3\sqrt{4800001200000081}-200000027}\right)^{1/3}\right.$$
$$\left.+\frac{\left(\frac{1}{5}\left(3\sqrt{4800001200000081}-200000027\right)\right)^{1/3}}{2^{2/3}}\right)$$

Numerical vs Symbolic Calculations

Maple or Mathematica will give an **exact** result

$$-\frac{1}{3}+\frac{100}{3}2^{2/3}\left(1+i\sqrt{3}\right)\left(\frac{5}{3\sqrt{4800001200000081}-200000027}\right)^{1/3}$$
$$-\frac{\left(1-i\sqrt{3}\right)\left(\frac{1}{5}\left(3\sqrt{4800001200000081}-200000027\right)\right)^{1/3}}{600\ 2^{2/3}}$$

Numerical vs Symbolic Calculations

Maple or Mathematica will give an **exact** result

$$-\frac{1}{3}+\frac{100}{3}2^{2/3}\left(1-i\sqrt{3}\right)\left(\frac{5}{3\sqrt{4800001200000081}-200000027}\right)^{1/3}$$
$$-\frac{\left(1+i\sqrt{3}\right)\left(\frac{1}{5}\left(3\sqrt{4800001200000081}-200000027\right)\right)^{1/3}}{600\ 2^{2/3}}$$

Sources of Error in Scientific Computing

- There are several sources of error in solving real-world Scientific Computing problems
- Some are beyond our control, e.g. uncertainty in modeling parameters or initial conditions
- Some are introduced by our numerical approximations:
  - **Rounding**: Computers work with finite precision arithmetic, which introduces rounding error
  - **Truncation/discretization**: We need to make approximations in order to compute (finite differences, truncate infinite series...)

Sources of Error in Scientific Computing

- It is crucial to understand and control the error introduced by numerical approximation, otherwise our results might be **garbage**
- This is a major part of Scientific Computing, called **error analysis**
- Error analysis became crucial with advent of modern computers: larger scale problems ⇒ more accumulation of numerical error
- Most people are more familiar with **rounding error**, but **discretisation error** is usually far more important in practice

Discretisation Error vs Rounding Error

Consider finite difference approximation to $f'(x)$:

$$f_{\text{diff}}(x;h) \equiv \frac{f(x+h)-f(x)}{h}$$

From Taylor series:

$$f(x+h) = f(x) + hf'(x) + f''(\theta)h^2/2, \text{ where } \theta \in [x, x+h]$$

we see that

$$f_{\text{diff}}(x;h) = \frac{f(x+h)-f(x)}{h} = f'(x) + f''(\theta)h/2$$

Suppose $|f''(\theta)| \leq M$, then bound on discretization error is

$$|f'(x) - f_{\text{diff}}(x;h)| \leq Mh/2$$

Discretisation Error vs Rounding Error

But we can't compute $f_{\text{diff}}(x; h)$ in exact arithmetic

Let $\tilde{f}_{\text{diff}}(x; h)$ denote finite precision approximation of $f_{\text{diff}}(x; h)$

Numerator of $\tilde{f}_{\text{diff}}$ introduces rounding error $\lesssim \epsilon f(x)$
(on modern computers $\epsilon \approx 10^{-16}$, will discuss this shortly)

Hence we have the rounding error

$$\begin{aligned} |f_{\text{diff}}(x; h) - \tilde{f}_{\text{diff}}(x; h)| &\lesssim \left| \frac{f(x+h) - f(x)}{h} - \frac{f(x+h) - f(x) + \epsilon f(x)}{h} \right| \\ &= \epsilon |f(x)|/h \end{aligned}$$

Discretisation Error vs Rounding Error

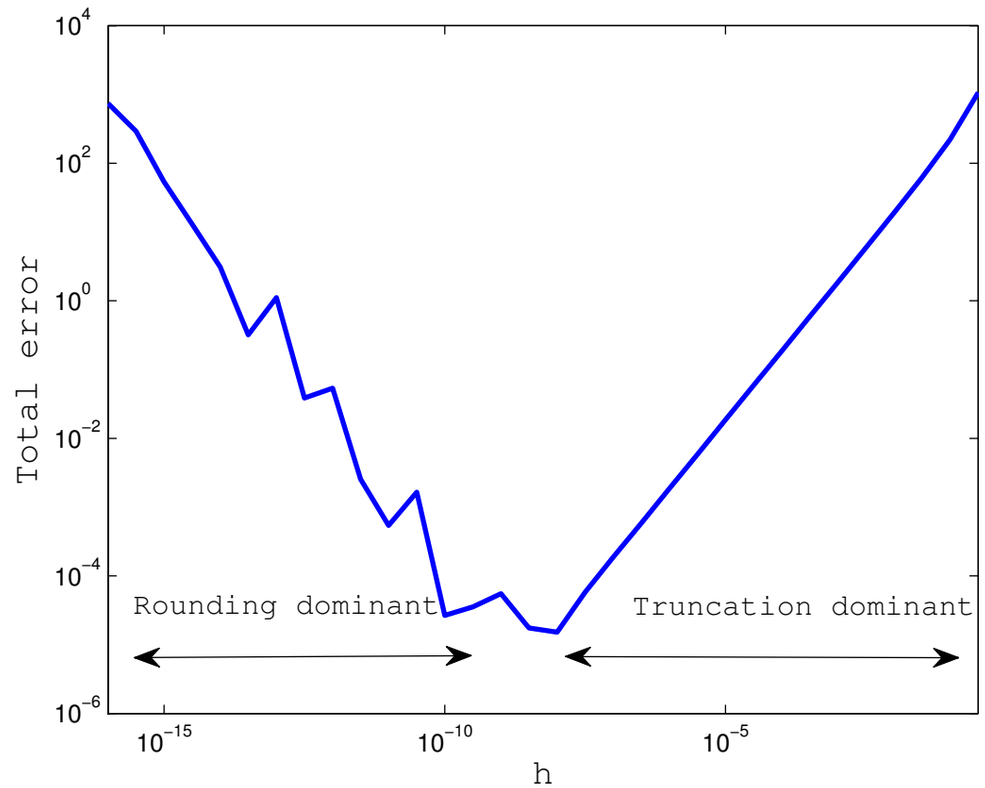
We can then use the triangle inequality ($|a+b| \leq |a| + |b|$) to bound the total error (discretization and rounding)

$$\begin{aligned}
|f'(x) - \tilde{f}_{\text{diff}}(x;h)| &= |f'(x) - f_{\text{diff}}(x;h) + f_{\text{diff}}(x;h) - \tilde{f}_{\text{diff}}(x;h)| \\
&\leq |f'(x) - f_{\text{diff}}(x;h)| + |f_{\text{diff}}(x;h) - \tilde{f}_{\text{diff}}(x;h)| \\
&\leq Mh/2 + \epsilon|f(x)|/h
\end{aligned}$$

Since ϵ is so small, we expect discretization error to dominate until h gets sufficiently small

Discretisation Error vs Rounding Error

For example, consider $f(x) = \exp(5x)$, f.d. error at $x = 1$ as function of h:



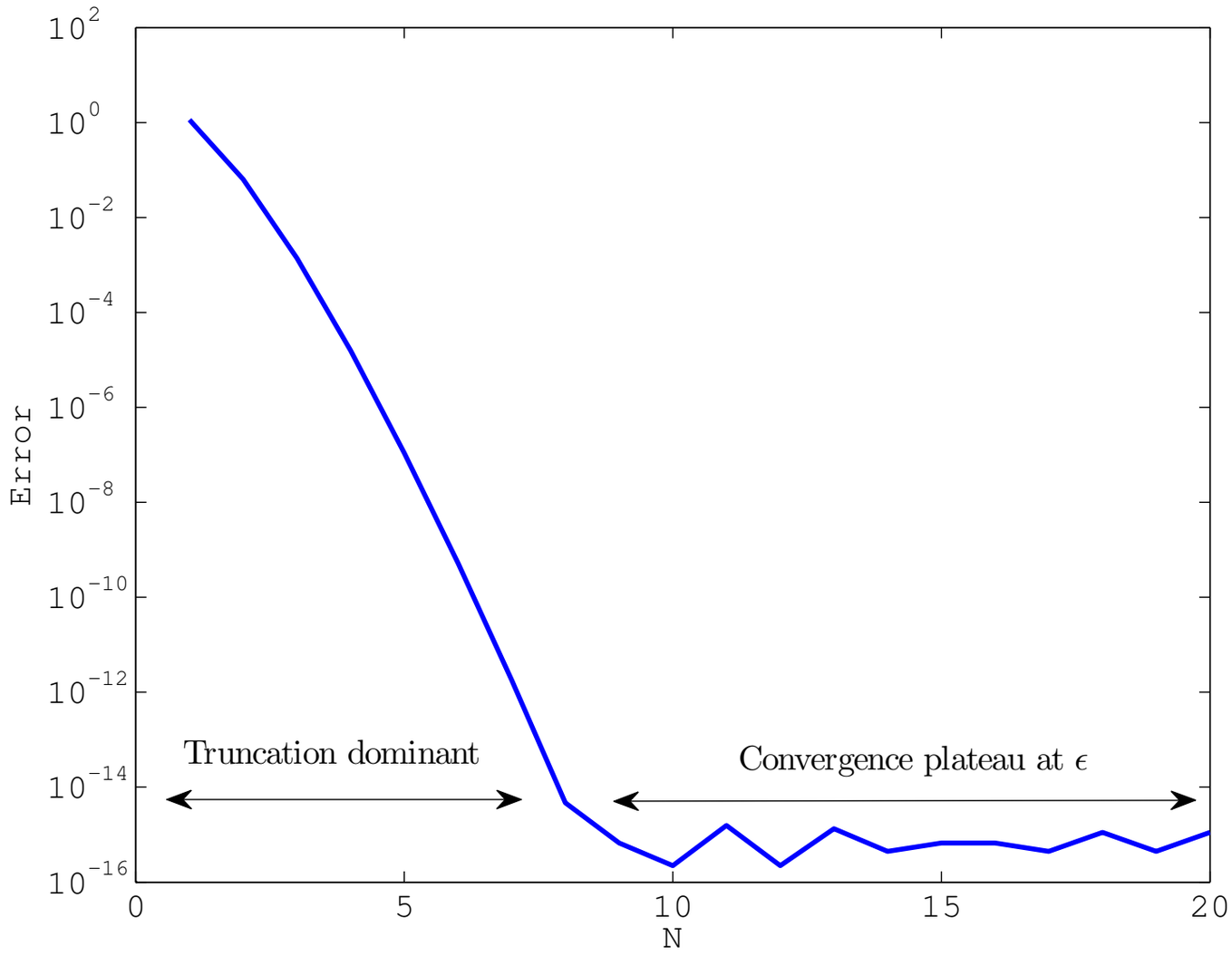
Exercise: Use calculus to find local minimum of error bound as a function of h to see why minimum occurs at $h \approx 10^{-8}$

Discretisation Error vs Rounding Error

- Note that in this finite difference example, we observe error growth due to rounding as $h \to 0$.
- This is a particularly nasty situation, due to factor of h in the denominator in the error bound.

Discretisation Error vs Rounding Error

Error plateau



More common situation is an error plateau due to rounding

Absolute vs Relative Error

Recall our bound $|f'(x) - \tilde{f}_{\text{diff}}(x; h)| \leq Mh/2 + \epsilon|f(x)|/h$

This is a bound on Absolute Error:

Absolute Error $\equiv$ true value - approximate value

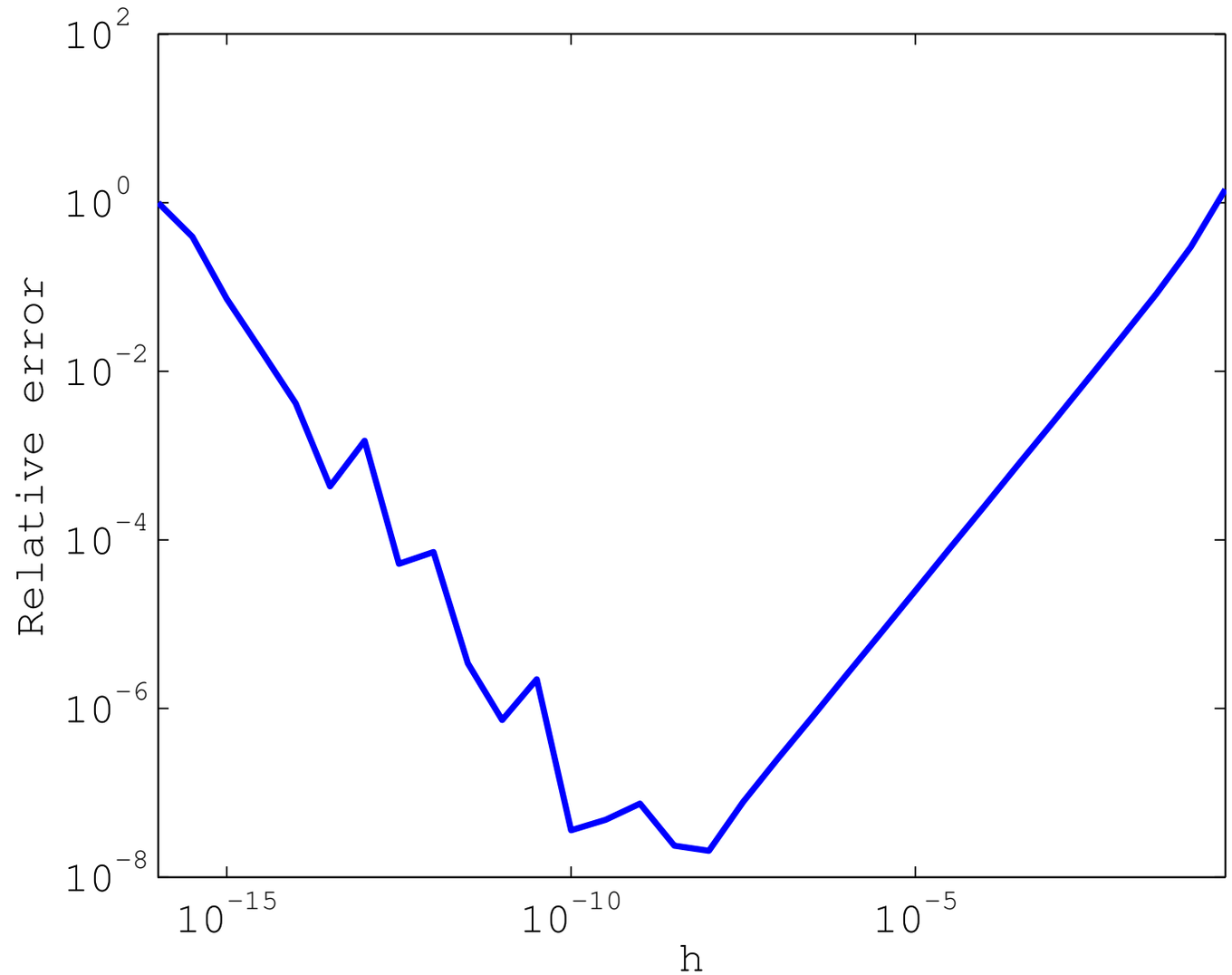
Generally more interesting to consider Relative Error:

Relative Error $\equiv$ Absolute Error / true value

Relative error takes the scaling of the problem into account

Absolute vs Relative Error

For our finite difference example, plotting relative error just rescales the error values



Sidenote: Convergence plots

We have shown several plots of error as a function of a discretization parameter

These types of plots are very important, since they allow us to demonstrate that a numerical method is behaving as expected

To display convergence data in a clear way, it is important to use appropriate axes for our plots

Sidenote: Convergence plots

Most often we will encounter algebraic convergence, where error decreases as αh^β for $h \to 0$, for some $\alpha, \beta \in \mathbb{R}$

Algebraic convergence: If $y = \alpha h^\beta$, then

$\log(y) = \log(\alpha) + \beta \log(h)$

Plotting algebraic convergence on log-log axes (`loglog` in Matlab) asymptotically yields a straight line with gradient β

Hence a good way to deduce algebraic convergence rate is by comparing error to αh^β on log-log axes

Sidenote: Convergence plots

Sometimes we will encounter exponential convergence, where error decays as $\alpha e^{-\beta N}$, for $N \to \infty$

If $y = \alpha e^{-\beta N}$, then $\log(y) = \log(\alpha) - \beta N$

Hence for exponential convergence, better to use "semilog-y" axes (`semilogy` in Matlab), e.g. see previous "error plateau" plot

Numerical Sensitivity

In practical problems we will always have input perturbations (modeling uncertainty, rounding error)

Let $y = f(x)$, and denote perturbed input $\hat{x} = x + \Delta x$

Also, denote perturbed output by $\hat{y} = f(\hat{x})$, and $\hat{y} = y + \Delta y$

The function f is sensitive to input perturbations if $\Delta y \gg \Delta x$

This sensitivity is a property of f (i.e. not related to a numerical approximation of f)

Sensitivity and Conditioning

Hence for a sensitive problem: small input perturbation $\Longrightarrow$ large output perturbation

Can be made quantitive with concept of condition number[1]

$$\text{Condition number} \equiv \frac{|\Delta y/y|}{|\Delta x/x|}$$

$$\text{Condition number} \gg 1 \iff \text{small perturbations are amplified}$$
$$\iff \text{ill-conditioned problem}$$

[1] Here we introduce the relative condition number, generally more informative than the absolute condition number

Sensitivity and Conditioning

Condition number can be analyzed for different types of problem (independent of algorithm used to solve the problem), e.g.

- Function evaluation, $y = f(x)$
- Matrix multiplication, $Ax = b$ (solve for b given x)
- Matrix equation, $Ax = b$ (solve for x given b)

Condition of various problems

Ex: Function Evaluation

Want to evaluate $f(x)$ for any value of x

How does solution $g = f(x)$ change with x

$$\text{cond}(f(x)) = \frac{|f(x+\delta x) - f(x)| / |f(x)|}{|\delta x| / |x|}$$

$$= \frac{|x|}{|f(x)|} \frac{|f(x+\delta x) - f(x)|}{|\delta x|} \approx \frac{|x|\,|f'(x)|}{f(x)}$$

Example: $f(x) = x^{1/2}$

$$\text{cond}(f(x)) = \frac{|x|\,\left|\frac{1}{2} x^{-1/2}\right|}{|x|^{1/2}} = \frac{1}{2}$$

Calculating square roots is a well-conditioned problem.

Ex: $f(x) = \frac{10}{1-x^2}$, $\text{cond}(f) = \frac{2x^2}{|1-x|}$

ill conditioned for values of x near 1

Example: Subtractions (Trefethen)

View subtraction as a mapping

$$f: \mathbb{R}^2 \to \mathbb{R}, \quad f(x_1, x_2) = x_1 - x_2$$

$$\text{cond}(f(x)) = \frac{\|x\| \, \|J\|}{\|f\|}, \quad \text{where} \quad J = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{pmatrix} = \nabla f = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

Use $\| x_1, x_2, \ldots, x_n \|_\infty = \max \{ |x_1|, |x_2| \ldots |x_n| \}$

$$\frac{\|x\| \, \|J\|}{\|f\|} = \frac{\left\| \begin{matrix} 1 \\ -1 \end{matrix} \right\|_\infty \left\| \begin{matrix} x_1 \\ x_2 \end{matrix} \right\|_\infty}{|x_2 - x_1|} = \frac{\max \{ |x_1|, |x_2| \}}{|x_2 - x_1|}$$

subtraction is ill conditioned

when $x_1 \approx x_2$.

Example : Zero finding.

Zeros of $f(x) = (x-1)^4$ are $x_i = 1$, $i = 1, 2, 3, 4$

Perturbed problem: $(x-1)^4 - 10^{-8} = 0$

$$\Rightarrow \quad x = 1 \pm \sqrt{\pm 10^{-4}}$$

$x_1 = 1.01$, $x_2 = 0.99$, $x_3 = 1 + 10^{-2} i$, $x_4 = 1 - 10^{-2} i$

Change in 10^{-8} of problem has caused a change in 10^{-2} in soln.

$$\text{Cond}(P) = \frac{10^{-2}}{10^{-8}} = 10^6.$$

Ill Conditioned.

Stability of an Algorithm

In practice, we solve problems by applying a numerical method to a mathematical problem, e.g. apply Gaussian elimination to $Ax = b$

To obtain an accurate answer, we need to apply a stable numerical method to a well-conditioned mathematical problem

Question: What do we mean by a stable numerical method?

Answer: Roughly speaking, the numerical method doesn't accumulate error (e.g. rounding error) and produce "garbage"

Numerical Stability of an Algorithm

For an algorithm to be useful, it must be stable in the sense that rounding errors do not accumulate and result in "garbage" output

More precisely, numerical analysts aim to prove backward stability: The method gives the exact answer to a slightly perturbed problem

For example, a numerical method for solving $Ax = b$ should give the exact answer for $Ax = (b + \Delta b)$ for small Δb

Numerical Stability of an Algorithm

Hence the importance of conditioning is clear: Backward stability doesn't help us if the mathematical problem is ill-conditioned!

For example, if A is ill-conditioned then a backward stable algorithm for solving $Ax = b$ can still give large error for x

Backward stability analysis is a deep subject

We will, however, compare algorithms with different stability properties and observe the importance of stability in practice