Welcome to 620-381 Computational Mathematics

What is Computational Mathematics?

also called

- Numerical Methods

- Numerical Analysis (includes proof of error properties)

- Scientific Computing
There are three great branches of science: theory, experiment, and computation.

Trefethen’s Maxims

As technology advances, the ingenious ideas that make progress possible vanish into the inner workings of our machines, where only experts may be aware of their existence. Numerical algorithms, being exceptionally uninteresting and incomprehensible to the public, vanish exceptionally fast.
**Definition:** (Trefethen 1992)

Numerical analysis is the study of algorithms for the problems of *continuous mathematics*

⇒ not for the problems of *discrete mathematics* (graph theory, pattern matching, discrete optimization etc. → Computer Science)

*The big gulf in the mathematical sciences is between the continuous problems (and people) and the discrete ones. Most scientists and engineers are in the continuous group, and most computer scientists are in the discrete one.*

Trefethen’s Maxims
What is it for?

Problem from Eng/Sci

Maths

Numerical Analysis

Scientific Computing

Computer Science

Solution

Computer System

381
Many BIG problems:

- climate change modeling
- drug design
- salinity/pollution monitoring

require heavy ‘scientific computing’.

NOT (typically) bioinformatics — more discrete.
381 is a First Course

- root-finding \( f(x) = 0 \) find \( x \)
- linear systems \( Ax = b \) find \( x \)
- interpolation \( f(x_i) = y_i \) find \( f \)
- quadrature \( I = \int_a^b f(x) \, dx \) find \( I \)
- ode (IVP) \( \dot{x} = F(x, t); x(0) = x_0 \) find \( x(t) \)

Usually can't find an exact answer!

Not:

- nonlinear systems, eigenvalues, optimization, BVPs, PDEs etc.
Goal

Make you more informed users of the wide range of existing software

- Netlib, TOMS

- Matlab, Octave, Scilab, Mathematica etc.

- NAG, IMSL, GSL
Maths prerequisites

*Computational mathematics is mainly based on two ideas: Taylor series, and linear algebra.*

Trefethen’s Maxims

⇒ revise Taylor series, eigenvalues, vector spaces, matrices
Given a problem, we want an algorithm that

1. gives the answer (in principle)

2. is efficient \textit{how fast is it? how much memory?}

3. does not amplify numerical errors \textit{numerically stable}

In computer science, no need for 3!
This week: aim to cover

• ways to describe numerical errors

• the effect of numerical errors
Sources of error in scientific computing

<table>
<thead>
<tr>
<th>Error</th>
<th>Remedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Errors in the model</td>
<td>Experience in modelling</td>
</tr>
<tr>
<td>Errors in the data</td>
<td>Better equipment, more money</td>
</tr>
<tr>
<td>Truncation error</td>
<td>finer grid/keep more terms/ run longer</td>
</tr>
<tr>
<td>Roundoff errors</td>
<td>higher-precision arithmetic</td>
</tr>
<tr>
<td>Programming blunders</td>
<td>software engineering, testing</td>
</tr>
</tbody>
</table>

We focus on **Truncation error** and **Roundoff errors** which together are the **Computational error**. Usually one of these is dominant. Neither of these present in discrete problems.
**Truncation error** : arises because we need to approximate continuous objects (functions, integrals, DEs ..) by discrete ones (sums ...)

**Roundoff error** : arises because we need to approximate real numbers by numbers we can store in computer

A good algorithm minimizes numerical error and maximizes efficiency (poss. a tradeoff here)
End of Lecture 1
Measures of error

Let $\hat{x}$ be approximate value of $x$

- **absolute error**: $e = |\hat{x} - x|$

- **relative error**: $r = |\hat{x} - x| / |x| = e / |x|$

  not useful if $x = 0$

  a better measure if $x$ can be large
A number correct to \textbf{n decimal places} has \textbf{absolute error} < 1/2 in \textit{nth} decimal place \Rightarrow e < 0.5 \times 10^{-n}

\textbf{Example} \quad \frac{22}{7} \approx 3.14285714285 \text{ approximates } \pi \approx 3.14159265358 \text{ to 2 decimal places since } e \approx 0.00126448926735 < 0.5 \times 10^{-2}

A number correct to \textbf{n significant digits/figures} has error < 1/2 in \textit{nth} digit

related to relative error, since if we multiply \( \hat{x} \) and \( x \) by \( 10^k \) we don’t change number of sig. figs. or \( r \)
\( \hat{x} \) is correct to **n significant digits** \( \Rightarrow r < 5 \times 10^{-n} \)

Let \( x = f \times 10^k \) where \( 0.1 \leq f < 1 \). Then error < 1/2 in nth digit \( \Rightarrow e < 0.5 \times 10^{-n} \times 10^k \)

so \( r = e / |x| = \frac{e}{f \times 10^k} < \frac{0.5 \times 10^{-n} \times 10^k}{f \times 10^k} < 5 \times 10^{-n} \) since \( |f| \geq 0.1 \)

\(<\) Example \( 22/7 \approx 3.14285714285 \) approximates \( \pi \approx 3.14159265358 \) to 3 significant figures \( \Rightarrow r < 5 \times 10^{-3} \) (actually \( \approx 4.024994347 \times 10^{-4} \))

Conversely

\( r < 0.5 \times 10^{-n} \Rightarrow \hat{x} \) is correct to **n significant digits**
How many significant digits do we need?

No physical constants are known to more than around 11 digits, and no truly scientific problem requires computation with much more precision than this. (OK, throw in another 5 or 6 digits to counter the slow accumulation of rounding errors in very long calculations – using numerically stable algorithms, of course, without which you’re sunk in any precision.)

Trefethen’s Maxims

**Engineering accuracy** = 2–3 digits
Data error vs. computational error

Given a computation with input (data) \( x \) and output \( y \) e.g. \( y = f(x) \) we produce an approximation \( \hat{Y} \) for 2 reasons:

- data error: the true input \( x \) is replaced by an approximate value \( X \)

- computational error: we don’t compute exactly \( f \) but some approximation \( \hat{f} \)

\[
\hat{Y} - y = \hat{f}(X) - f(x) = [\hat{f}(X) - f(X)] + [f(X) - f(x)] = [\hat{Y} - Y] + [Y - y]
\]

Total error = computational error + propagated data error
Only the first part depends on our algorithm.

The second part depends only on \( f \) i.e. the problem, and the magnitude of the data error.

Given an idea of the data error, it makes no sense to make the computational error much less than the propagated data error.

\( \Rightarrow \) use the data error as a scale to assess size of computational error
Forward vs. backward error

Given a computation $y = f(x)$, in general we will produce an approximation $\hat{Y}$. The **forward error** is $\Delta Y = \hat{Y} - Y$.

We would like a small forward error for accurate answer BUT can be hard to estimate forward error.

Instead ask: **Is the approximation the exact answer to a nearby problem?**

Is $\hat{Y} = f(X + \Delta X)$ for some small **backward error** $\Delta X$?
Then total error \( \hat{Y} - y = [f(X + \Delta X) - f(X)] + [f(X) - f(x)] \)

If backward error \( \Delta X \) < data error \( = X - x \), we’ve done as well as can be expected! The errors we made during the computation are the same as if we’d solved the problem exactly but with different, equally plausible data.

Turns out to be easier often to bound the backward error.

If the backward error is not too big (compared to relevant data error) we say the algorithm is **backward stable** or just **stable**.
Examples of forward vs. backward error

How are forward and backward errors related?
Conditioning and sensitivity

Once we have introduced backward error, the error made by the algorithm can be treated as if it was error in the data. Then the forward error only depends on the problem \( f \) and the size of the input (data/backward) error.

A problem which is sensitive to small errors is called ill-conditioned or ill-posed \( \Rightarrow \) no numerical method will get a very accurate answer. Typically determine sensitivity using perturbation analysis (assume small errors).
We quantify sensitivity by defining a **condition number** to measure how sensitive the answer is to errors in the input.

Example  Sensitivity of evaluating a function

Note that condition number of inverse problem is reciprocal of that of original problem.
In previous example, we had an **absolute condition number**. More common to use **relative condition number** $\sim$ relative error of output/relative error of input i.e. the magnification factor of relative error

▲ Example
Rule of thumb:

\[ \text{Forward error} \sim \text{condition number} \times \text{backward error} \]

⇒ a stable method on a well-conditioned problem → accurate answer
YES!

⇒ stable method on an ill-conditioned problem → inaccurate answer

*If the answer is highly sensitive to perturbations, you have probably asked the wrong question.*

Trefethen’s Maxims

**re-examine the problem!**
⇒ unstable method on a well-conditioned problem → inaccurate answer

This is what we must avoid!

All that it is reasonable to ask for in a scientific calculation is stability, not accuracy.

Trefethen’s Maxims
Some examples of inaccurate calculations

1. Taylor series to compute \( \exp(-20) \)

2. recurrence relation

3. finite difference formula

What is going on in each case?
End of Lecture 2
This week: aim to cover

- floating point numbers
- roundoff error
- error propagation
How numbers are stored in a computer

To understand one ubiquitous source of error in scientific computing (roundoff error), have to understand how numbers are stored.

For details, see “Machine numbers — from the beginning”.

In scientific problems, numbers can vary greatly in magnitude ⇒ we try to store numbers with a fixed relative error (precision), not absolute error

Floating Point numbers invented to do this.

We describe current standard: IEEE 754 used by most computers (perhaps not completely) for double precision numbers
To represent numbers with greatly varying size, we use scientific notation

Example 6.023 \times 10^{23}, 1.055 \times 10^{-34}

In the computer, we do the same but store both the exponent and fraction in base 2 (binary representation).

1011_2 represents the integer 8 + 0 + 2 + 1 = 11_{10}

⇒ approximate real numbers by a nearby rational number with a finite binary expansion
\[ x = \pm (1.f_1 f_2 \cdots f_t) \times 2^e = (1 + f) \ 2^e \]

\( f \) is the \textit{fraction or mantissa}:\ 0 \leq f < 1, \text{ each } f_i = 0 \text{ or } 1

\( e \) is the \textit{exponent}, a binary integer

\langle \text{ Example } \ 125.75 = 64 + 32 + 16 + 8 + 4 + 1 + 0.5 + 0.25 = \ 1111101.11_2 = 1.1111101111_2 \times 2^6 = 1.1111101111_2 \times 2^{1102} \)

so \( 1 + f = 1.111110111_2, \ e = 1102 \)
$e$ determines the **range** of numbers that can be represented

$f$ determines the number of significant digits carried $\rightarrow$ the **precision** of numbers that can be represented

In double precision, use 64 bits (8 bytes) to store each number

- 1 for sign $\pm$

- 11 for exponent $\rightarrow 2^{11} = 2048$ possible values for $e$

- 52 for fraction $\Rightarrow t = 52$

$\Rightarrow$ to store $10^6$ double precision numbers takes 8 Mbytes.
To get exponent, subtract 1023 from the 11-bit integer $\in [1, 2046] \rightarrow e \in [-1022, 1023]$ (so we get equal numbers of ‘small’ and ‘large’ numbers)

(extreme values $e = -1023, 1024$ used for special cases/error flags)

$\Rightarrow$ largest number representable $\text{realmax} \approx 2^{1024} \approx 10^{308}$

Any number larger than this causes Overflow

stored as $\text{Inf}$, with $e = 1024, f = 0$
⇒ smallest (non-zero, normalized) number representable $\text{realmin} = 2^{-1022} \approx 10^{-308}$

If $e = -1023$, use **denormalized** numbers ($x = \pm f \, 2^{-1022}$)

⇒ smallest non-zero number representable $= 2^{-1074} \approx 5 \times 10^{-324}$

**Any number smaller than this gives Underflow $\rightarrow 0$**

Finally

**Results of illegal operations (divide by zero, $\log(0)$ etc. ) stored as NaN (Not a Number), with $e = 1024, f \neq 0$**
Only numbers $\in [\text{realmin, realmax}]$ with binary fraction expansion that terminates in less than 53 digits can be represented exactly as Floating Point number $\rightarrow \text{machine numbers}$, denoted by $\mathcal{F}$

$\triangle$ Example $125.75 = 1.11110111_2 \times 2^{1102}$ which needs only 9 binary digits in the fraction $\Rightarrow 125.75$ is a machine number

all the rest will be \textit{approximated} with an error $\rightarrow \text{Roundoff error}$

\textit{If rounding errors vanished, 95\% of numerical analysis would remain.}

Trefethen’s Maxims
The structure of the Floating Point number system

Since there are $2^{52}$ numbers in $[1,2)$, $2^{52}$ numbers in $[2,4)$, $2^{52}$ numbers in $[4,8)$ etc.

**machine numbers are nonuniformly distributed**

The granularity of the machine numbers is specified by the gap between 1 and the next machine number $1+2^{-t}$, called **machine epsilon** $\varepsilon_M$

$\Rightarrow$ IEEE double precision numbers have machine epsilon $2^{-52} \approx 2 \times 10^{-16}$, given by `eps` in Matlab
If you try to produce a non-machine number, must store it as nearest machine number by \textbf{rounding} — call this number $fl(x)$

default rounding mode: \textit{round to nearest, then even}

$\Rightarrow$ maximum error $= \varepsilon_M / 2 \times 2^e = u \times 2^e$ (\textbf{unit roundoff} $u$ is $\varepsilon_M / 2$)

$\Rightarrow$ max. relative error $= u \times 2^e / (1 + f) \times 2^e = u / (1 + f) \in (u/2, u] \leq u$

\textbf{unit roundoff determines the precision with which numbers are stored}
IEEE double precision numbers have precision $2^{-53} \approx 10^{-16}$

16 decimal digits of precision

also have IEEE single precision numbers with unit roundoff $2^{-24} \approx 10^{-7}$

7 decimal digits of precision e.g. ANSI C float, Matlab single

Floating point precision sets a lower bound to the level of (relative) data error — make this error just in storing the numbers in a computer.
Summary

1. Some numbers (machine numbers) can be represented exactly as a Floating Point number; most can’t

2. there’s a smallest and largest FP number

3. FP numbers have a precision \( u \) — the source of inevitable roundoff error
From these properties, we see the error on storing a number

\[ fl(x) = x(1 + \delta), \quad |\delta| \leq u \quad \forall x \in \mathcal{R} \]

⇒ nearest machine number to \( x \) is at most a factor \( 1 \pm u \) away

Since arithmetic operations on machine numbers produce results that must get rounded, we get a model for Floating point arithmetic (ignoring underflow/overflow)

\[ fl(x \text{ op } y) = (x \text{ op } y)(1 + \delta), \quad |\delta| \leq u \quad \forall x, y \in \mathcal{F}, \text{ op } = +, -, \times, \div \]
Error propagation

If error caused by $u$ stayed that size $\rightarrow$ no problem!

Does it?

Example Multiplication

$$(x \otimes y) \otimes z \equiv fl(fl(x \times y) \times z) = [xy(1 + \delta_1)] \times z(1 + \delta_2)$$

where $|\delta_i| < u$

$$(x \otimes y) \otimes z = xyz(1 + \delta_1)(1 + \delta_2)$$

Note: $(x \otimes y) \otimes z = x \times \hat{y} \times \hat{z}$ where $\hat{y} = y(1 + \delta_1)$ and $\hat{z} = z(1 + \delta_2)$

$\Rightarrow$ multiplication is backward stable
what about forward error?

\[(x \otimes y) \otimes z = xyz(1 + \theta_2) \quad |\theta_2| \leq 2u/(1 - 2u)\]

\[
\frac{(x \otimes y) \otimes z - xyz}{xyz} \leq 2u/(1 - 2u) \approx 2u
\]

**no problem with multiplication!** (must be well-conditioned)

using \(\prod^n (1 + \delta_i) \equiv 1 + \theta_n\) where \(|\theta_n| < \frac{nu}{1-nu}, \quad nu < 1\)
Example Addition?

\[ x \oplus y \equiv \text{fl}(x + y) = [x + y](1 + \delta_1) \]

so \( x \oplus y = \hat{x} + \hat{y} \) where \( \hat{x} = x(1 + \delta_1) \) and \( \hat{y} = y(1 + \delta_1) \)

\( \Rightarrow \) addition (subtraction) is backward stable

forward error?

\[ \frac{x \oplus y - (x + y)}{x + y} = \delta_1 \]

OK if \( x, y \) are machine numbers.

But what if they have errors e.g. roundoff?
\[ fl(x) \oplus fl(y) = [x(1 + \delta_1) + y(1 + \delta_2)](1 + \delta_3) \]
\[ = x(1 + \delta_1)(1 + \delta_3) + y(1 + \delta_2)(1 + \delta_3) = x(1 + \theta_2) + y(1 + \bar{\theta}_2) = \hat{x} + \hat{y} \]

still backward stable BUT

\[
\frac{|fl(x) \oplus fl(y) - (x + y)|}{x + y} = \frac{|x(1 + \theta_2) - y(1 + \bar{\theta}_2)|}{x + y} < \frac{|x| + |y|}{x + y} \frac{2u}{1 - 2u}
\]

but \[ \frac{|x| + |y|}{x + y} \] can be large if \( x, y \) are nearly equal and opposite!

⇒ subtraction of 2 nearly equal numbers is (relatively) ill-conditioned

If 2 nearly equal numbers (with error) are subtracted, relative error can be greatly magnified
End of Lecture 3
Severe cancellation/subtractive cancellation

Remedy? → change formula/algorith to avoid subtraction

Example quadratic formula, sample variance

⇒ try to avoid calculations that rely on cancellation ...
Now go back to our Taylor series for $e^{-20}$ ....

1. why does the loop terminate?

2. why is the answer wrong?
Stability of Algorithms

Since some error (roundoff) is usually present, must have algorithms that don’t grow error too fast

Some growth is inevitable

Example $S_n = \sum_{k=1}^{n} a_k$ typically has error $\sim n^{1/2}u$ and is guaranteed to have error $< (n - 1)u \sum_k |a_k| + O(u^2)$

but exponential growth (error $\sim K^n u$) is disastrous!
Example using a recurrence relation …

Remedy? run recurrence **backwards**!
Roundoff vs. truncation error

Sometimes there is a tradeoff between truncation error and roundoff error.

Example: forward difference approximation to the derivative

Approximate \( f'(x) \) by \( \frac{f(x+h) - f(x)}{h} \)

- Truncation error \( \sim h \)
- Roundoff error \( \sim \frac{u}{h} \)

\( \Rightarrow \) minimum total error at an optimal \( h \approx u^{1/2} \)

\( \Rightarrow \) no point using \( h \) smaller than this!
Summary: main effects of roundoff error after Stewart

1. can **accumulate** over long computations: inevitable
   ☐ Example sums

2. can **reveal** other errors by cancellation: try to do problem another way

3. can **grow** so fast it swamps the actual answer: try to do problem another way
Interval arithmetic

If we have control over the rounding mode (not in all compilers), can actually get rigorous bounds on the forward error due to roundoff, using interval arithmetic.

→ ‘verifiable scientific computing’

prob. will become more widespread
End of Lecture 4
Root-finding

Find $x$ such that $f(x) = 0$
'find zeroes of $f$', 'find roots of $f$', 'root-finding'

look for **iterative procedures**

guess $x_0 \rightarrow x_1 \rightarrow x_2 \cdots$

construct rule so the sequence of iterates **converges to the root** $x^*$

Since we have to **stop** the iteration $\rightarrow$ truncation error (dominates roundoff until very close to root)

*Most problems of continuous mathematics cannot be solved by finite algorithms.*

Trefethen’s Maxims
This week: aim to cover

- fixed point iteration
- bisection, Newton’s method
Fixed point iteration

Rearrange $f(x) = 0$ to $x = g(x)$ (not uniquely)

A point $x^*$ that satisfies $x^* = g(x^*)$ is a fixed point of the function $g$

then try the iteration

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

does it converge to the fixed point?

Let’s try it ...
what happens?

1. sometimes it blows up!

2. if it converges, (absolute) error behaves like

   \[ e_{n+1} \approx k e_n \]

   linear convergence

3. \( k \) is different for different \( g \)

4. the smaller \( k \) is, the faster the convergence
Explanation by Taylor series ...
Some Theorems!

**Definition:** If $g$ is defined on $[a,b]$ and $x^* = g(x^*)$ for some $x^* \subset [a,b]$, $g$ has a **fixed point** $x^* \subset [a,b]$

**Theorem:** Sufficient conditions for a unique fixed point

1. **Existence:** If $g \in C[a,b]$ and $g(x) \in [a,b]$ ($g$ maps $[a,b]$ onto $[a,b]$ or a subinterval) then $g$ has a fixed point in $[a,b]$

2. **Uniqueness:** If, also, if $g'(x)$ exists on $(a,b)$ and $|g'(x)| \leq k < 1$ on $(a,b)$ then $g$ has a unique fixed point in $[a,b]$
Proofs!

1. **Existence**: If \( g(a) = a \) or \( g(b) = b \) fixed point exists, so suppose not.

   Then \( g(a) > a, g(b) < b \). Define \( h(x) = g(x) - x \Rightarrow \exists h \in C[a,b] \) with \( h(a) > 0, h(b) < 0 \)

   \( \Rightarrow \), by IVT, \( \exists c \in [a,b] \) such that \( h(c) = 0 \)

   \( \Rightarrow g(c) = c \Rightarrow c \) is fixed point of \( g \)

2. **Uniqueness**: \( |g'(x)| \leq k < 1 \) and suppose \( p, q \) are fixed points of \( g \) with \( p \neq q \). We prove a contradiction \( \Rightarrow p = q \)
By MVT, \( \exists c \) such that

\[
g(p) - g(q) = g'(c)(p - q)
\]

\[\Rightarrow |p - q| = |g(p) - g(q)| \leq |g'(c)||p - q| \leq k |p - q| < |p - q|\]

which can’t be true

\[\Rightarrow p = q\]

\[\Rightarrow \text{fixed point is unique}\]

\[|g'(x)| \leq k < 1 \Rightarrow g \text{ is a contraction mapping}\]
**Theorem: Convergence of fixed point iteration**

Under conditions of previous theorem, for any \( x_0 \in [a, b] \), the sequence \( x_n = g(x_{n-1}) \) converges to the unique fixed point \( x^* \).

**Proof** By previous theorem, unique fixed point exists. Since \( g \) maps \([a,b]\) into \([a,b]\), the sequence of iterates \( \{x_n\} \) is defined.

\[
| x_n - x^* | = | g(x_{n-1}) - g(x^*) | = | g'(c) | | x_{n-1} - x^* |
\]
\[
\leq k | x_{n-1} - x^* | \leq k^2 | x_{n-2} - x^* | \cdots \leq k^n | x_0 - x^* |
\]

so \( \lim_{n \to \infty} | x_n - x^* | = | x_0 - x^* | \lim_{n \to \infty} k^n = 0 \) since \( k < 1 \)

so \( x_n \to x^* \)
so far so good

**BUT**

- not cheap to decide when conditions of theorem are met
- for a given \( g \), this method may not find all roots

but this method is used for some difficult problems via **Contraction Mapping Theorem**
Pseudocode

input x0
compute x1
while (cgce criterion not satisfied) and (no. of iterations < ITMAX)
    compute next iteration
end
output results

what convergence criterion to use?
Convergence criteria

Many possible:

1. tolerance on value of function \(| f(x) | < \delta \)

2. tolerance on absolute error \(| x_n - x_{n-1} | < \text{AbsTol} \)

3. tolerance on relative error \(| x_n - x_{n-1} | / | x_n | < \text{RelTol} \)

4. mixed \(| x_n - x_{n-1} | < \text{AbsTol} + \text{RelTol} \) | \( x_n \) |
1 is OK if $f$ is not shallow at the root

2 OK provided AbsTol not too small

usually 3 is safe if RelTol > a few $u$

4 switches between absolute tolerance if $x_n$ is small and relative tolerance for $|x_n| > 1$
End of Lecture 5
**Bisection**

Fixed point iteration is v. simple but not guaranteed to converge!

Simplest *globally convergent* method is *bisection aka interval halving*

Start with an interval \([a_1, b_1]\) s.t. \(f(a_1), f(b_1)\) of opposite sign

⇒ a root lies in \([a_1, b_1]\) by IVT
Find midpoint \( p_1 = (a_1 + b_1)/2 \)

Repeat until convergence the step:

\[
\begin{align*}
\text{if } f(p_1) &= 0 \\
\quad &\quad x^* = p_1 \\
\text{else (choose subinterval s.t. root lies there)} \\
\quad &\quad \text{if } f(p_1) \text{ and } f(a_1) \text{ have same sign} \\
\quad &\quad \quad x^* \text{ in } [p_1,b_1] \\
\quad &\quad \text{else} \\
\quad &\quad \quad x^* \text{ in } [a_1,p_1] \\
\end{align*}
\]

This is just a floating point version of **binary search**.
• Since we keep $x^*$ within the interval at each step, we are guaranteed convergence.

• We have a precise error bound, since

$$|p_N - x^*| \leq \max(p_N - a_N, b_N - p_N) \leq |b_N - a_N|$$

• Interval halves at every step, so we get an extra bit of accuracy every step.

  $\Rightarrow$ For a given tolerance $tol$, required number of steps $N$ is known in advance:

  $$2^{-N}(b - a) < tol \Rightarrow N > \log_2\left(\frac{b-a}{tol}\right)$$
error bound *halves* at every step (actual error jumps around a bit)
Newton’s method

\[ \text{slope} = f'(x_n) \]

based on **slope** of function
Derivation: Taylor series around $x_n$:

$$f(x_{n+1}) \approx f(x_n) + f'(x_n)(x_{n+1} - x_n) = 0$$

which gives

Newton-Raphson iteration:

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

Again, a first order recurrence relation.

$\Rightarrow$ problems if $f'(x_n) = 0$
**Definition:** $f$ has a **simple root** at $x^*$ if $f(x^*) = 0$, $f'(x^*) \neq 0$.

**Definition:** $f$ has a **double root** at $x^*$ if $f(x^*) = 0$, $f'(x^*) = 0$, $f''(x^*) \neq 0$.

$\Rightarrow$ Newton’s method has trouble at a double root

$\triangle$ Example
At a simple root:

\[ e_{n+1} \sim C e_n^2 \]

**quadratic convergence**

At a double root:

\[ e_{n+1} \sim C e_n \]

**linear convergence**

WHY?
Explanation by Taylor series ...
Quadratic convergence is a GOOD THING

Write $e_n = 10^{-b_n}$

$b_b = \text{number of correct decimal digits}$

**Linear Convergence** ⇒

$$e_{n+1} \approx Ce_n, \quad C < 1$$

$$\Rightarrow b_{n+1} \approx b_n - \log_{10} C$$

→ number of correct digits increases by a constant
Quadratic Convergence \( \Rightarrow \)

\[ e_{n+1} \approx C e_n^2 \]

\[ \Rightarrow b_{n+1} \approx 2 b_n - \log_{10} C \]

\( \Rightarrow \) number of correct digits doubles each iteration!!

\( \Rightarrow \) Quadratic Convergence very desirable
BUT

• $f'(x)$ may be hard to calculate

• it may not converge :-(

**Theorem:** If $f''(x)$ is $C^1$ then $\exists \delta > 0$ such that Newton’s method converges to the root $x^* \forall x_0 \in [x^* - \delta, x^* + \delta]$

$\Rightarrow$ converges provided we start close enough

BUT
• don’t know $x^*$

• don’t know $\delta$

$\Rightarrow$ get close with a globally convergent method, then switch to a fast method

$\rightarrow$ hybrid method

Note: for some functions, Newton’s method is guaranteed to converge

< Example
End of Lecture 6
This week: aim to cover

- secant method
- hybrid methods
- Gauss elimination
- Partial pivoting
- LU factorization
What if the derivative is really nasty? ⇒ try the secant method instead, which does not need the derivative function.

The secant method.

For this method, use two current “guesses” at a root of \( f(x) = 0 \). Want to replace the pair \( x_0, x_1 \) by the pair \( x_1, x_2 \) where \( x_2 \) is a new guess.

Use the straight line \( L \) from \( (x_0, f(x_0)) \) to \( (x_1, f(x_1)) \) (a secant of the curve \( y = f(x) \)) to give the slope.
slope = $\Delta y/\Delta x$

$y = f(x)$

$x_{n+1}$

$x_n$

$x_{n-1}$
Just replace $f'(x_n)$ from NR method by
\[
\frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}
\]
which gives

**Secant method:**
\[
x_{n+1} = x_n - f(x_n) \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}.
\]

Since each new guess depends on the previous two guesses $\Rightarrow$ a second order recurrence relation.
The secant method:

- quite efficient: converges quickly towards a root :-)

- must start with two guesses :-(

- don’t need derivative :-}
Using Taylor series, \( \rightarrow \) (at simple root)

\[
e_{n+1} \sim C e_n e_{n-1}
\]

\( \Rightarrow e_{n+1} \sim C e_n^p \) where \( p = \frac{1 + \sqrt{5}}{2} \approx 1.618 \)

\( \Rightarrow \) secant is **superlinearly convergent but not quadratically convergent**

(but only need 1 function evaluation/iteration; Newton needs 2)
Summary

fixed point  \( e_{n+1} \sim C e_n \), may diverge
bisection   \( e_{n+1} \sim \frac{1}{2} e_n \), globally convergent
Newton     \( e_{n+1} \sim C e_n^2 \), simple root
Newton     \( e_{n+1} \sim \frac{1}{2} e_n \), double root
secant     \( e_{n+1} \sim C e_n^{1.618} \), simple root
Hybrid methods

Attractive to combine global convergence of bisection with speed of another method e.g. Newton or secant

→ hybrid methods that switch between basic methods such that interval is always bracketing

Example Matlab’s fzero

switches between bisection, secant and IQI (faster than secant)

eeds no derivative
What about systems of nonlinear equations?

Try Newton’s Method for system of 2 equations in 2 variables.

\[ f(x, y) = 0 \quad g(x, y) = 0 \]

Expand in Taylor series about current iterate to get tangent plane approximation:

\[
\begin{align*}
    f(x_{n+1}, y_{n+1}) & \approx f(x_n, y_n) + \frac{\partial f}{\partial x} |n (x_{n+1} - x_n) + \frac{\partial f}{\partial y} |n (y_{n+1} - y_n) = 0 \\
    g(x_{n+1}, y_{n+1}) & \approx g(x_n, y_n) + \frac{\partial g}{\partial x} |n (x_{n+1} - x_n) + \frac{\partial g}{\partial y} |n (y_{n+1} - y_n) = 0
\end{align*}
\]
or in matrix notation

\[
\begin{bmatrix}
  f(x_n, y_n) \\
g(x_n, y_n)
\end{bmatrix} + \begin{bmatrix}
  \frac{\partial f}{\partial x} |_{n} & \frac{\partial f}{\partial y} |_{n} \\
  \frac{\partial g}{\partial x} |_{n} & \frac{\partial g}{\partial y} |_{n}
\end{bmatrix} \begin{bmatrix}
x_{n+1} - x_n \\
y_{n+1} - y_n
\end{bmatrix} = \begin{bmatrix}
0 \\
0
\end{bmatrix}
\]

where partial derivs are evaluated at \((x_n, y_n)\)

\[\Rightarrow\] at each iteration. must solve a linear system of the form

\[J |_{n} (x_{n+1} - x_n) = -f |_{n}\]

\[\Rightarrow\] must learn how to solve linear systems first!
Linear systems

In principle, the Taylor series of a function of \( n \) variables involves an \( n \)-vector, an \( n \times n \) matrix, an \( n \times n \times n \) tensor, and so on. Actual use of orders higher than two, however, is so rare that the manipulation of matrices is a hundred times better supported in our brains and in our software tools than that of tensors.

Trefethen’s Maxims
Given A: $n \times n$ square matrix

b: $n \times 1$ matrix (column vector)

find $x$ where

$$Ax = b$$

We assume A is nonsingular so a unique solution exists
A

- dense
  - direct methods
    - (only roundoff)

- sparse
  - (many entries = 0)
  - iterative methods
    - truncation error also
Gauss elimination

Recall from 1st year:

Use row operations to reduce augmented system to lower triangular form then solve by back-substitution

As an algorithm:

Denote original $A$ by $A^{(1)} = [a_{ij}^{(1)}]$

original $b$ by $b^{(1)} = [b_i^{(1)}]^T$
Step 1: Assume $a_{11}^{(1)} \neq 0$

Define multipliers $l_{i1} = \frac{a_{i1}^{(1)}}{a_{11}^{(1)}}, \; i = 2..n$

Put zeroes below $a_{11}$:

$$a_{ij}^{(2)} = a_{ij}^{(1)} - l_{i1}a_{1j}^{(1)}, \; i, j = 2..n$$

$$b_i^{(2)} = b_i^{(1)} - l_{i1}b_1^{(1)}, \; i, = 2..n$$
We now have

\[
\begin{pmatrix}
a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\
0 & a_{22}^{(2)} & \cdots & a_{2n}^{(2)} \\
\vdots \\
0 & a_{n2}^{(2)} & \cdots & a_{nn}^{(2)}
\end{pmatrix}
x = 
\begin{pmatrix}
b_1^{(1)} \\
b_2^{(2)} \\
\vdots \\
b_n^{(2)}
\end{pmatrix}
\]

Repeat this step.
After $k$ steps, $A^{(k)}x = b^{(k)}$

$$A^{(k)} = \begin{pmatrix}
    a^{(1)}_{11} & a^{(1)}_{12} & \cdots & \cdots & a^{(1)}_{1n} \\
    0 & a^{(2)}_{22} & \cdots & \cdots & a^{(2)}_{2n} \\
    \vdots & 0 & \cdots & \cdots & \vdots \\
    \vdots & \vdots & a^{(k)}_{kk} & \cdots & a^{(k)}_{kn} \\
    0 & 0 & a^{(k)}_{nk} & \cdots & a^{(k)}_{nn}
\end{pmatrix}$$

Assume $a^{(k)}_{kk} \neq 0$
Define multipliers $l_{ik} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}$, $i = k + 1..n$

Put zeroes below $a_{kk}$:

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik}a_{kj}^{(k)}, \quad i, j = k + 1..n$$

$$b_{i}^{(k+1)} = b_{i}^{(k)} - l_{ik}b_{k}^{(k)}, \quad i = k + 1..n$$
After $n$ steps we have

$$
\begin{pmatrix}
    a^{(1)}_{11} & \cdots & a^{(1)}_{1n} \\
    0 & \cdots & a^{(n)}_{nn}
\end{pmatrix}
\begin{pmatrix}
    x_1 \\
    \vdots \\
    x_n
\end{pmatrix}
= 
\begin{pmatrix}
    b^{(1)}_1 \\
    \vdots \\
    b^{(n)}_n
\end{pmatrix}
$$

where $A^{(n)}$ is upper triangular

$$
Ux = b^{(n)} = g
$$
Solve this by back-substitution

\[ x_n = \frac{g_n}{u_{nn}} \]

\[ x_k = \frac{1}{u_{kk}}[g_k - \sum_{j=k+1}^{n} u_{kj}x_j], \quad k = n-1 \ldots 1 \]

These formulae → algorithm for Gauss elimination (ready for coding)

BUT
1. what if $a^{(k)}_{kk} = 0$?

$a^{(k)}_{kk}$ is the **pivot**

Remedy: swap rows

2. in order to put zeroes under pivot, we subtract rows

$\Rightarrow$ we can amplify roundoff error if we subtract 2 large numbers
(most common if the multiplier is large)

**Gauss elimination is not backward stable!**

Remedy: choose pivot so that multipliers are never large

**pivoting strategy**
End of Lecture 7
Simplest pivoting strategy (and usually enough)

**Partial pivoting**

At step $k$,

- look at elements $a_{kk}^{(k)}$ and below
- choose the largest of these in magnitude to be the new pivot e.g. $a_{lk}^{(k)}$
- swap rows $l$ and $k$

$\Rightarrow$ multipliers formed from new pivot satisfy $|l_{ik}| < 1$
This also handles zero pivots

Note: don’t actually swap rows; swap pointers or row indices

Example

Other pivoting strategies exist but partial pivoting is regarded as usually sufficiently (backward) stable.

For more details, see later.
Operations Count for Gaussian Elimination

Measure by number of multiply/divides.

Assume no pivoting required.

1st stage: for each \(i = 2..n\)

\[
\begin{align*}
\text{form } l_{i1} &= a_{i1}^{(1)} / a_{11}^{(1)} \quad 1 \div \\
a_{ij}^{(2)} &= a_{ij}^{(1)} - l_{i1}a_{1j}^{(1)}, \quad i, j = 2..n \quad n - 1 \times \\
b_i^{(2)} &= b_i^{(1)} - l_{i1}b_1^{(1)} \quad 1 \times \\
&\quad n + 1 \times / \div \\
\rightarrow (n - 1)(n + 1) \times / \div \text{ for 1st stage (inc. } n - 1 \text{ for b)}
\end{align*}
\]
2nd stage: $n \mapsto n - 1$

$\rightarrow (n - 2)(n) \times / :$ for 2nd stage (inc. $n - 2$ for b)

...$

last stage: $n = 2$

$\rightarrow (1)(3) \times / :$ for last stage (inc. 1 for b)

Total so far: $\sum_{k=1}^{n} (k^2 - 1) = \frac{1}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n$

inc. $\sum_{k=1}^{n-1} k = \frac{1}{2}n^2 - \frac{1}{2}n$ for b
Now back-substitute

\[ x_n = \frac{g_n}{u_{nn}} \]
\[ x_{n-1} = \cdots \]
\[ \vdots \]
\[ x_1 = \cdots \]

→ total for back-substitution \( \sum_{k=1}^{n} k = \frac{1}{2}n^2 + \frac{1}{2}n \)
Total work:

\[
\frac{1}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n + \frac{1}{2}n^2 + \frac{1}{2}n
\]

\[= \frac{1}{3}n^3 + n^2 - \frac{1}{3}n \approx \frac{1}{3}n^3\]

of which \(\frac{1}{2}n^2 - \frac{1}{2}n + \frac{1}{2}n^2 + \frac{1}{2}n = n^2\) comes from b

**Important:** Work for Gauss elimination: \(\approx \frac{1}{3}n^3\) operations
So to solve

\[ Ax = (b_1, b_2, \ldots, b_k) \]

requires \( \approx \frac{1}{3}n^3 + kn^2 \) operations.

By comparison:
use inverse of matrix

\[ x = A^{-1}b \]

To find \( A^{-1} \), solve

\[ Ax = I = (e_1, e_2, \ldots, e_n) \]

takes \( \approx \frac{1}{3}n^3 + n \cdot n^3 \approx \frac{4}{3}n^3 \) operations

actually can be done in \( \approx n^3 \) operations

Moral: DON’T FORM MATRIX INVERSE — 3 times more work than solving the system!
Many software libraries contain no provision for Gauss Elimination — WHY NOT?

if you solve $Ax = b_1 \approx \frac{1}{3}n^3$ operations) and, some time later, solve $Ax = b_2 \approx \frac{1}{3}n^3$ operations) most of these operations have been done before! (all but $n^2$ operations)

⇒ try to store info (e.g. $l_{ij}$) so you don’t waste this effort.

In fact, store multipliers $l_{ij}$ in a lower triangular matrix
\[ L = \begin{pmatrix}
1 & 0 & 0 & 0 \\
\ell_{21} & 1 & 0 & 0 \\
\ell_{31} & \ell_{32} & 1 & 0 \\
\ell_{41} & \ell_{42} & \ell_{43} & 1
\end{pmatrix} \]

\[ L = \begin{cases}
\ell_{ij} & i > j \\
1 & i = j \\
0 & i < j
\end{cases} \]
if no pivoting required

\[ A = LU \]

**LU factorization of** \( A \)

where \( U \) is upper triangular matrix resulting from Gauss Elimination

**Proof:** straightforward from formulae for Gauss Elimination

**Note:** LU factorization is not unique, since

\[ LU = (LD)(D^{-1}U) = \bar{L}\bar{U} \]

where \( D \) is a nonsingular diagonal matrix.
Now our strategy becomes

to solve \( Ax = b_1 \)

factorize \( A = LU \) (\( \approx \frac{1}{3}n^3 \) operations)

to solve \( LUx = b_1 \); solve \( Ly = b_1 \) by forward substitution (\( \approx \frac{1}{2}n^2 \) operations) where \( y = Ux \)

solve \( Ux = y \) by back substitution (\( \approx \frac{1}{2}n^2 \) operations)

Total: \( \approx \frac{1}{3}n^3 + n^2 \) operations
then, to solve $Ax = b_2$

$L, U$ known

solve $Ly = b_2$ by forward substitution ($\approx \frac{1}{2}n^2$ operations)

solve $Ux = y$ by back substitution ($\approx \frac{1}{2}n^2$ operations)

$\Rightarrow$ next system with same $A$ takes $\approx n^2$ operations

$\rightarrow$ most libraries have an LU factorization procedure and an LU solve procedure instead of GE
End of Lecture 8
This week: aim to cover

- Special matrices
- Conditioning of linear systems
- Error analysis for solving linear systems
If row interchanges are required, then

\[
PA = LU
\]

**LU factorization of row-permuted A**

where P is a permutation matrix describing the row interchanges

→ PA is A but with rows interchanged so no pivoting required
then to solve $Ax = b$

$$PAx = Pb$$

$$LUx = Pb$$

solve $Ly = Pb$

solve $Ux = y$
Special Matrices

GE and LU apply to general dense matrices. If the matrix has any special structure, it’s often possible to exploit this.

1. **Diagonally dominant matrices**

   - easily recognizable

   - require no pivoting strategy
**Definition:** A is **diagonally dominant** (by rows) iff

\[ |a_{ii}| \geq \sum_{j=1, j \neq i}^{n} |a_{ij}| \]

**Theorem:** If A is **diagonally dominant**, then A is nonsingular and A can be factorized into LU with no pivoting required.

**Moral:** If you know A is diagonally dominant don’t bother pivoting
2. **Positive definite matrices**

**Definition:** A is **positive definite** iff

$$x^T Ax > 0 \ \forall x \neq 0$$

usually consider only symmetric positive definite matrices

- arise naturally in some applications
  - Example Least Squares fitting

- require no pivoting strategy
**Theorem:** A is **symmetric positive definite** is equivalent to

all the pivots encountered in GE are positive

⇒ no zero pivots

also LU factorization/solve is backward stable for positive definite matrices **without pivoting**
Since $\det A = \det L \times \det U = 1 \times \prod(pivots) > 0$

$\Rightarrow A$ is nonsingular.

Since $A = LU$ and $A = A^T$, $A = LU = A^T = (LU)^T = U^TL^T$

but the transpose of lower triang. is upper triang. and vice versa.

$\Rightarrow U = L^T, L = U^T$ so $A = U^TU \equiv R^TR$ in Matlab
Check: $x^T A x = x^T R^T R x = (Rx)^T R x = \|Rx\|^2 > 0 \; \forall x \neq 0$

$\Rightarrow$ a symmetric positive definite matrix has a factorization

$$A = R^T R$$

**Choleski factorization**

this takes $\approx \frac{1}{6} n^3 \times / \div + n \sqrt{}$ operations

**Moral:** If you know $A$ is symmetric positive definite, can save 50% of work by Choleski factorization.
3. **Banded matrices**

A matrix is banded if it has nonzeroes only near the main diagonal:

\[ a_{ij} = 0 \text{ for } |i - j| > k \]

**bandwidth** of matrix = \( 2k + 1 \)

i.e. a regular pattern of zeroes, a generalization of diagonal matrices.

Common cases: \( k = 1 \) **tridiagonal matrices**, \( k = 2 \) **pentadiagonal matrices**
If no pivoting required, the LU factors inherit this structure → can be factored very fast

Example factors of tridiagonal matrix are bidiagonal, can be factored in $\approx 3n \times \div$ operations VERY FAST!

For bandwidth $2k + 1$, takes $\approx nk^2 \times \div$ operations

Use any special structure of the matrix to save time/storage
Error analysis and conditioning

We have mentioned two kinds of error:

1. backward error: did we solve a problem close to the one we wanted to solve?

2. forward error: did we get a solution close to the actual one?
In linear systems, our inputs are a matrix $A$ and a vector $b$

and the output is a vector $x$.

So: how to measure the ‘size’ of errors in the inputs and outputs?

Need this to discuss sensitivity and (backward) stability of solving linear systems.

For details, see ‘MatrixNorms.pdf’.
Vector and Matrix norms

To be more precise, we have to introduce a way to measure the ‘size’ of vectors and matrices.

**Definition:** The **norm** of a vector \( \|x\| \) is a function \( \mathbb{R}^n \rightarrow \mathbb{R} \) such that

1. \( \|x\| \geq 0 \quad \forall x \in V \) where \( \|x\| = 0 \Leftrightarrow x = 0 \) (norms are positive for nonzero vectors)

2. \( \|\alpha x\| = |\alpha|\|x\| \) (scaling a vector scales norm by the same amount)

3. \( \|x + y\| \leq \|x\| + \|y\| \quad \forall x, y \in V \) (triangle inequality)
Example  The 3 most common vector norms are:

1. $\|x\|_1 = \sum |x_i|$ (the 1-norm)

2. $\|x\|_2 = (\sum x_i^2)^{1/2} = \sqrt{x^T x}$ (the 2-norm i.e. the usual Euclidean norm)

3. $\|x\|_\infty = \max_i |x_i|$ (the $\infty$-norm)

which are all special cases of the $p$-norm:

$$\|x\|_p = (\sum |x_i|^p)^{1/p}$$
In finite-dimensional spaces, it doesn’t matter much precisely which norm you use, since they can’t differ by more than a factor $n$ from each other.

\[
\|x\|_2 \leq \|x\|_1 \leq \sqrt{n}\|x\|_2 \\
\|x\|_\infty \leq \|x\|_2 \leq \sqrt{n}\|x\|_\infty \\
\|x\|_\infty \leq \|x\|_1 \leq n\|x\|_\infty
\]

So you may as well use whichever one is convenient.
A matrix norm is just a vector norm on the $m \cdot n$ dimensional vector space of $m \times n$ matrices.

**Definition:** The norm of a matrix $\|A\|$ is a function $\mathbb{M}_{m \times n} \mapsto \mathbb{R}$ such that

1. $\|A\| \geq 0 \quad \forall A \in \mathbb{M}_{m \times n}$ where $\|A\| = 0 \iff A = 0$ (norms are positive for nonzero matrices)

2. $\|\alpha A\| = |\alpha|\|A\|$ (scaling a matrix scales norm by the same amount)

3. $\|A + B\| \leq \|A\| + \|B\| \quad \forall A, B \in \mathbb{M}_{m \times n}$ (triangle inequality)
End of Lecture 9
Example The following are matrix norms:

1. $\|A\|_F = (\sum a_{i,j}^2)^{1/2}$ (the Frobenius norm)

2. $\|A\|_M = \max_{i,j} |a_{i,j}|$ (the max-norm)

Mostly we’ll use a common class of matrix norms, called subordinate matrix norms.
**Definition:** The **subordinate norm** of a (square) matrix $A$ is given by

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

for any vector norm. This is equivalent to:

$$\|A\| = \max_{\|x\|=1} \|Ax\|$$

In words, the (subordinate) norm of a matrix is the norm of the largest image under the map $A$ of a unit vector.
Example  The **subordinate p-norms** correspond to the vector norms listed above:

1. $\|A\|_1 = \max_j \sum_i |a_{ij}|$ (the maximum column sum)

2. $\|A\|_2$ (the 2-norm)

3. $\|A\|_\infty = \max_i \sum_j |a_{ij}|$ (the maximum row sum)

These are the only subordinate p-norms that are easy to compute.
The subordinate norms have some useful **submultiplicative** properties:

\[ \|Ax\|_p \leq \|A\|_p \|x\|_p \]

by definition; and

\[ \|AB\|_p \leq \|A\|_p \|B\|_p \]

Any norm with the latter property is called **consistent**.

Similarly, all matrix norms are within a factor of \(n\) of each other, so you may as well use whichever one is convenient.
Sensitivity of a linear system

Suppose \( A, b \) have errors — how big are errors in \( x \)?

For details, see ‘MatrixNorms.pdf’.

\[
\frac{\| \Delta x \|}{\| x \|} \leq \frac{\kappa(A)}{1 - \kappa(A) \frac{\| \Delta A \|}{\| A \|}} \left( \frac{\| \Delta A \|}{\| A \|} + \frac{\| \Delta b \|}{\| b \|} \right)
\]

provided \( \kappa(A) \frac{\| \Delta A \|}{\| A \|} < 1 \), which ensures that the matrix \( A + \Delta A \) is nonsingular.
For small enough $\|\Delta A\|$, we will have $\kappa(A)\frac{\|\Delta A\|}{\|A\|} \ll 1$ so that

$$\frac{\|\Delta x\|}{\|x\|} \lesssim \kappa(A)\left(\frac{\|\Delta A\|}{\|A\|} + \frac{\|\Delta b\|}{\|b\|}\right)$$

relative forward error $\lesssim$ (condition number) (relative error in $A, b$)

**Definition**: The (normwise) **condition number** of a square nonsingular matrix is

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

Note: it’s not cheap to compute $\kappa(A)$ since it takes $\approx n^3$ operations to find $A^{-1}$. Most codes instead try to **estimate** $\kappa(A)$. 
The precise value depends on what norm you’re using but they will all be quite similar in size (to within factors of $n$)

**Some properties of the condition number**

1. $\kappa(I) = \|I\|\|I^{-1}\| = \|I\|^2 = 1$ in any subordinate norm

2. $\kappa(A) = \|A\|\|A^{-1}\| \geq \|AA^{-1}\| = \|I\| = 1$ so $\kappa(A) \geq 1$

Matrices with $\kappa(A) \gg 1$ are **ill-conditioned** → the solution is very sensitive errors in $A$ or $b$
Heuristic

If $\kappa(A) \sim 10^k$ then in solving $Ax = b$ in t-digit arithmetic, you will lose $k$ decimal digits of precision $\Rightarrow x$ has $t - k$ digits of precision

$\Rightarrow$ if $\kappa(A) > 10^t$ it’s not worth solving the system since $x$ has no significant figures!

Example
A is ill-conditioned ⇔ A is ‘nearly singular’

\[
\frac{1}{\kappa_p(A)} = \min\left[\frac{\|\Delta A\|_p}{\|A\|_p}\right]
\]

where \(A + \Delta A\) is singular

i.e. if \(\kappa(A)\) is large, \(A\) is close to a singular matrix,

Note: det \(A\) is NOT a good measure of how close to singular a matrix is.

Example
Geometrically, ill-conditioned matrix ⇒ a system of hyperplanes that are **almost parallel**

⇒ wouldn’t have to tilt them much to make them parallel and hence have no or infinitely many solutions.

\( \kappa(A) \) is computed in Matlab with the `cond` command.

Not cheap to compute \( \kappa(A) \)

instead usually estimate its value. In Matlab one can use either `condest` to estimate \( \kappa_1(A) \) or `rcond` which estimates its reciprocal.
Backwards error

Suppose all backward error in A not b. Then

\[(A + \Delta A)\hat{x} = b \Rightarrow \]

\[\Delta A\hat{x} = -r\]

where \(r = A\hat{x} - b\) is the residual. Then

\[\|r\| \leq \|\Delta A\| \|\hat{x}\| \Rightarrow \|\Delta A\| \geq \frac{\|r\|}{\|\hat{x}\|}\]

so the relative backward error

\[\frac{\|\Delta A\|}{\|A\|} \geq \frac{\|r\|}{\|A\| \|\hat{x}\|} \equiv \eta\]

is measured by the relative residual \(\eta\).
large relative residual $\rightarrow$ large (normwise) relative backward error.

Bound can be attained $\Rightarrow$ small relative residual $\rightarrow$ small backward error.

**A stable method produces small relative residuals.**

BUT small residuals does NOT $\Rightarrow$ small errors in $x$

(depends on condition number of $A$)
Error analysis

Under standard model for Floating point arithmetic:

**Theorem**: If the $n \times n$ triangular system

$$Tx = b$$

is solved by substitution, the computed solution $\hat{x}$ satisfies

$$(T + \Delta T)\hat{x} = b$$

where

$$|\Delta T| \leq \gamma_n |T| \approx nu |T|$$

since $\gamma_n \equiv \frac{nu}{1-nu} \approx nu$

i.e. it solves a system with a nearby matrix (and the correct $b$!).
Solving triangular systems by forward/backsubstitution is componentwise backward stable.

In general, componentwise stability is a more stringent requirement than normwise stability, since for any monotone norm

$$| \Delta T | \leq \gamma_n | T | \Rightarrow \| \Delta T \| \leq \gamma_n \| T \|$$

Solving triangular systems by forward/backsubstitution is norm-wise backward stable which is the usual usage of the term backward stable.

$\Rightarrow$ final stages of solving a linear system are no problem; any problems must arise in the factorization stage.
**Theorem**: If the $n \times n$ system

$$Ax = b$$

is solved by Gauss elimination, the computed solution $\hat{x}$ satisfies

$$(A + \Delta A)\hat{x} = b$$

where

$$|\Delta A| \leq 2\gamma_n |\hat{L}||\hat{U}|$$

which is **not what we’re after**. Want a bound for $\Delta A$ involving $|A|$.

**The problem with Gauss elimination is that** $|\hat{L}||\hat{U}|$ **can be much bigger than** $|A|$. 
Gauss elimination with partial pivoting (GEPP)

Partial pivoting ensures that

\[ |\hat{L}| \leq 1 \text{ componentwise, but what can we say about } |\hat{U}|? \]

To progress, go to a normwise analysis:

\[ \|\Delta A\| \leq 2\gamma_n \|\hat{L}\| \|\hat{U}\| \]

for a monotone norm.

**Definition:** The **growth factor** (using the max-norm):

\[ \rho = \|\hat{L}\|_M \|\hat{U}\|_M / \|A\|_M \]
⇒ for the 1 or ∞ norms

\[ \| \Delta A \| \leq 2n^2 \gamma_n \rho \| A \| \]

→ normwise backward error is determined by the growth factor.

using

\[ \| A \|_M \leq \| A \|_1 \leq n \| A \|_M \]

\[ \| A \|_M \leq \| A \|_{\infty} \leq n \| A \|_M \]
For Gauss elimination without pivoting, there is no bound for the growth factor.

Example The matrix \[
\begin{bmatrix}
\delta & 1 \\
1 & 1
\end{bmatrix}
\] for \( \delta \ll 1 \) has \( \rho \approx \delta^{-2} \) which can be arbitrarily large.

For Gauss elimination with partial pivoting:
• There are matrices that have $\rho = 2^{n-1}$ — for them GEPP is not backward stable.

• These matrices appear to be very rare in practise. For most matrices, $\rho$ is very modest in size e.g. $< 10$

• For random matrices e.g. matrices with components drawn from a standard normal distribution, $\rho \sim n^{1/2}$

Hence experts say that, for practical purposes, GEPP is (norm-wise) backward stable — the default direct method for solving general dense linear systems.
End of Lecture 10
This week: aim to cover

- Polynomial interpolation
- Piecewise polynomial interpolation
- Cubic splines
SUMMARY:

1. Gauss elimination without pivoting is not backward stable.

2. GEPP is, for practical purposes, (normwise) backward stable.

3. A backward stable method produces small (relative) residuals.

4. This does not imply small (forward) errors if \( \kappa(A) \gg 1 \) i.e. your problem is ill-conditioned.
What if $A$ is ill-conditioned?

Remedy:

1. do you really need $x$ or just $f(x)$ or $r$?

2. do the problem another way i.e. change $A$

3. live with the reduced accuracy
What does Matlab’s \ do?

To solve the system $Ax = b$, in Matlab just type

$x = A\backslash b$; remember $x = A^{-1}b$

This **backslash** command:

- solves by substitution if $A$ is triangular
- attempts Cholesky factorization if $A$ is symmetric
- does GEPP by LU factorization if $A$ is general dense matrix
⇒ easiest way to solve with LU if you want to keep the LU factors

\[[L,U,P]=lu(A); \ x=U\backslash(L\backslash P*b)\]

similarly for Cholesky:

\[R=chol(A); \ x=R\backslash(R'\backslash b)\]
Interpolation

The problem:

you are given a set of data points \( \{x_i, y_i\}, i = 0..n \) which represent values of some function.

How to approximate \( f(x), x \in [x_0, x_n] \neq x_i \)?

- used where \( f \) is expensive to compute or obtain

- census data

came originally from working with tables of functions
Mathematically: Given \( \{x_i, y_i\}, i = 0..n \), find \( f(x) \) such that

\[
f(x_i) = y_i, i = 0..n
\]

\( f \) passes through the data points → **interpolation**
If we know more can get variations:

Example Let $s_i = f'(x_i)$. Then given data $\{x_i, y_i, s_i\}, i = 0..n$, find $f(x)$ such that

$$f(x_i) = y_i, \quad f'(x_i) = s_i, \quad i = 0..n$$

$f$ agrees with data in values and slope $\rightarrow$ Hermite interpolation
• No unique answer to this problem — you have to say what kind of function you are looking for

• interpolation only makes sense if there is little error in \(\{x_i, y_i\}\) — else no point to force \(f\) through the data points. If \(\{x_i, y_i\}\) have error, instead fit a function to the data \(\rightarrow\) data fitting (later)

• as stated, \(\{x_i\}\) are given; you can’t choose to evaluate \(f\) where you want. If you can choose, have a different problem \(\rightarrow\) function approximation
Types of interpolation

1. Linear interpolation: choose class of functions where parameters appear linearly
   - Example: polynomial interpolation, piecewise polynomial interpolation, trigonometric interpolation

2. Nonlinear interpolation: parameters appear nonlinearly (harder)
   - Example: rational interpolation

We do only polynomial/piecewise polynomial interpolation.
Linear interpolation

$f$ is linear in the parameters $\alpha_j$

$$f = \sum_j \alpha_j b_j(x)$$

$b_j(x)$ are the basis functions

Interpolation $\rightarrow$

$$f(x_i) = \sum_j \alpha_j b_j(x_i) = y_i$$

or $B\alpha = y$ where $B_{ij} = b_j(x_i)$
Choose basis functions so that

1. easy to construct $B_{ij}$

2. B is well-conditioned

3. the interpolant approximates $f$ ‘well’
Why interpolate?

- determine \( \{ \alpha_j \} \) and then process \( f(x) \) further
  - Example differentiate/integrate
  - \( \rightarrow \) ill-conditioning an issue

- find \( f(x), x \neq x_i \)
  - Example for plotting, tables
  - \( \rightarrow \) may only need small residual, not accurate \( \{ \alpha_j \} \)
Polynomial interpolation

Why a polynomial?

- easy to differentiate/integrate

- **Weierstrass’ Approximation Theorem:**

  If $f(x) \in C[a, b], \forall \epsilon > 0, \exists$ a polynomial $P_n(x)$ of degree $n$ such that $|f(x) - P_n(x)| < \epsilon \ \forall x \in [a, b]$

$\Rightarrow$ a polynomial exists that approximates $f$
BUT theorem doesn’t say how big \( n \) is, or how good it is for a given \( \{x_i, y_i\} \)

How to find the **interpolating polynomial**?
$n = 1 \Rightarrow 2$ points $\rightarrow$ linear interpolant

$p_1(x) = y_0 + \frac{(y_1-y_0)}{(x_1-x_0)}(x - x_0)$

$= y_0 \frac{(x-x_1)}{(x_0-x_1)} + y_1 \frac{(x-x_0)}{(x_1-x_0)}$

$= [y_0 - x_0 \frac{(y_1-y_0)}{(x_1-x_0)}] + x \frac{(y_1-y_0)}{(x_1-x_0)}$

3 forms for the same interpolant
Similarly for 3 points:

\[ p_2(x) = y_0 \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} + y_1 \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} + y_2 \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} \]

guarantees \( p_2(x_i) = y_i \)

In general, define the **Lagrange interpolating polynomial**

\[ p_n(x) = \sum y_j l_j(x) \]
where

\[ l_j(x) = \frac{(x - x_0)(x - x_1) \cdots (x - x_{j-1})(x - x_{j+1}) \cdots (x - x_n)}{(x_j - x_0)(x_j - x_1) \cdots (x_j - x_{j-1})(x_j - x_{j+1}) \cdots (x_j - x_n)} \]

\[ = \prod_{k=0, k \neq j}^{n} \left( \frac{x - x_k}{x_j - x_k} \right) \]

so that

\[ l_j(x_i) = \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \]
\[ p_n(x_i) = \sum y_j l_j(x_i) = \sum y_j \delta_{ij} = y_i \]

There is only one such polynomial

**Proof**: (by contradiction)

Suppose there are 2 such (different) interpolating polynomials \( p_n(x), q_n(x) \) then \( p_n(x) - q_n(x) \) is of degree at most \( n \) but \( p_n(x_i) - q_n(x_i) = y_i - y_i = 0, \ i = 0..n \)

so \( p_n(x) - q_n(x) \) has \( n + 1 \) zeroes

but a polynomial of degree \( n \) can have at most \( n \) zeroes \( \Rightarrow q_n(x) \) can't exist \( \Rightarrow p_n(x) \) is unique
This is the **Lagrange form** of the interpolating polynomial.

Another form: the **monomial or power form**

\[ p_n(x) = a_0 + a_1x + \cdots + a_nx^n \rightarrow p_n(x_i) = \sum_j a_jx_i^j = y_i \]

\[
\begin{pmatrix}
1 & x_0 & x_0^2 & \cdots & x_0^n \\
1 & x_1 & x_1^2 & \cdots & x_1^n \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & x_n & x_n^2 & \cdots & x_n^n
\end{pmatrix}
\begin{pmatrix}
a_0 \\
a_1 \\
\vdots \\
a_n
\end{pmatrix}
= y
\]

\[ V a = y \]

\( V \) is a **Vandermonde matrix**
Since $p_n(x)$ exists $\Rightarrow V$ is nonsingular so solve $Va = y$ for $a \rightarrow p_n(x)$

BUT as $n$ increases, $V$ typically becomes ill-conditioned (depends on $\{x_i\}$)

so OK for low-order polynomials , no good for large $n$

Here it’s not the problem that’s ill-conditioned but our formulation
Recall we can have any basis for $\mathbb{P}_n$

\textbf{Example} An alternative representation: \textbf{Newton form}

Write $p_n(x)$ using the basis functions

$$b_j(x) = 1, x - x_0, (x - x_0)(x - x_1), \cdots \prod_{k=0}^{n} (x - x_k)$$

i.e. $p_n(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + \cdots + c_n \prod_{k=0}^{n} (x - x_k)$

which gives a matrix

$$B_{ij} = b_j(x_i) = 0, j > i$$

triangular system $\Rightarrow$ stable and fast: $O(n^2)$
Example: choose Lagrange polynomial form

\[ p_n(x_i) = \sum y_j l_j(x_i) = \sum y_j \delta_{ij} = y_i \]

or

\[ I\alpha = y \]

which is perfectly-conditioned and diagonal.

Moral: Choose the basis to improve numerical properties.
Error in polynomial interpolation

**Theorem:** If \( x_0, \cdots x_n \) are distinct points in \([a, b]\) and \( f \in C^{m+1}[a, b] \)

\[
f(x) = p_n(x) + \frac{f^{(n+1)}(\xi)}{(n+1)!} (x - x_0)(x - x_1) \cdots (x - x_n)
\]

for some \( \xi \in [a, b] \)

This is useful if you can bound the derivative

\(\triangledown\) Example
End of Lecture 11
so for functions with bounded high derivatives, expect

\[ e_n(x) \approx h^{n+1} \]

a truncation error, since not affected by roundoff.

Replacing a continuous object \( f(x) \) with one determined by a finite number of coefficients \( p_n(x) \) → \text{discretization error}
Convergence of polynomial interpolant

For functions with bounded derivatives

\[ |e_n(x)| \leq \frac{M}{4(n + 1)} \left(\frac{b - a}{n}\right)^{n+1} \to 0 \text{ as } n \to \infty \]

so \( p_n(x) \to f(x) \) as \( n \to \infty \)

Unfortunately, for many functions, \( f^{(n+1)} \) grows rapidly with \( n \) so \( p_n(x) \) does not \( \to f(x) \) as \( n \to \infty \)
Famous example: (Runge, 1901)

\[ f(x) = \frac{1}{1 + 25x^2} \text{ on } [-1, 1] \]

*Moral: High order polynomial interpolation is a bad idea!*

If you can choose \( \{x_i\} \), can do better.

\[ \triangle \quad \text{Example using Chebyshev points} \]
**Piecewise polynomial interpolation**

The problem with high order polynomial interpolation is trying to use 1 function to describe a lot of data.

Instead, interpolate with low order polynomials locally and then stitch pieces together

→ piecewise polynomial interpolation

As \( n \) increases, the number of pieces increases, not the order of each piece.
**Piecewise cubic interpolation**

Piecewise linear interpolation gives no info on slopes etc. so → piecewise cubic interpolation

In each piece we have a cubic

\[ q_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^2(x - x_{i+1}) \text{ in } [x_i, x_{i+1}] \]

For \( x_0 \cdots x_n \) we have \( n \) cubic pieces → \( 4n \) coefficients

How to determine them?

We’ll start by assuming we **know** the slopes (Hermite interpolant)
Since we want to interpolate

- \( q_i(x_i) = a_i = y_i \)  
  \( n \) equations
  goes through LH knot

- \( q_i(x_{i+1}) = a_i + b_i h_i + c_i h_i^2 = y_{i+1} \)  
  where \( h_i = x_{i+1} - x_i \)  
  \( n \) equations
  goes through RH knot

→ continuous interpolant
we also know the slopes: do the same to $q'(x)$:

- $q'_i(x_i) = b_i = s_i \quad n \text{ equations}$

- $q'_i(x_{i+1}) = b_i + 2c_i h_i + d_i h_i^2 = s_{i+1} \quad n \text{ equations}$

$\rightarrow 4n$ equations for $4n$ coefficients
so we solve for \( c_i \):

\[
c_i = \frac{y_{i+1} - y_i}{h_i} - s_i \equiv \frac{y_i' - s_i}{h_i}
\]

and \( d_i \):

\[
d_i = \frac{s_i + s_{i+1} - 2y_i'}{h_i^2}
\]

so we’re done!

Example: Matlab’s \texttt{pchip} chooses the slopes to make the interpolant “shape-preserving”
But often we **don’t know** the slopes!

Any choice of \( \{s_i\} \) in the above produces a \( C^1 \) interpolant.

Can we choose the slopes to make a smoother interpolant e.g. \( C^2 \)? We try by demanding \( q''(x) \) be continuous at **internal knots** \( i = 1..n - 1 \)

\[
q''_i(x_{i+1}) = q''_{i+1}(x_{i+1}) \quad i = 0..n - 2
\]

\( n - 1 \) equations for \( n + 1 \) unknowns (the slopes)
⇒ we need 2 extra conditions (the **end conditions**)

→ **cubic splines**

most common form of interpolation now

can be written in terms of ‘nice’ basis functions but we won’t cover them
How to find coefficients?

Express everything in terms of the slope at each knot \( s_i, i = 0 \ldots n \)

\[
q''_i(x_{i+1}) = 2c_i + 2d_i h_i = \frac{2}{h_i} (2s_{i+1} + s_i - 3y'_i)
\]

\[
q''_{i+1}(x_{i+1}) = -d_{i+1} h_{i+1} = \frac{2}{h_{i+1}} (-2s_{i+1} - s_{i+2} + 3y'_{i+1})
\]

so we make these equal:

\[
\frac{2}{h_i} (2s_{i+1} + s_i - 3y'_i) = \frac{2}{h_{i+1}} (-2s_{i+1} - s_{i+2} + 3y'_{i+1})
\]
\( \rightarrow \) equation for \( s_i \)

\[
h_{i+1}s_i + (2h_{i+1} + 2h_i)s_{i+1} + h_is_{i+2} = 3(h_i y'_{i+1} + h_{i+1}y_i') \quad i = 0..n-2
\]

\( n - 1 \) linear equations for \( n + 1 \) \( s_i \)

\( \Rightarrow \) we use 2 extra conditions to close the system

Let's move \( s_0, s_n \) (the end slopes) to RHS:
\[(2h_0 + 2h_1)s_1 + h_0s_2 = 3(h_0y'_1 + h_1y'_0) - h_1s_0\]
\[h_2s_1 + (2h_1 + 2h_2)s_2 + h_1s_3 = 3(h_1y'_2 + h_2y'_1)\]
\[\vdots \quad \vdots\]
\[h_{n-1}s_{n-2} + (2h_{n-2} + 2h_{n-1})s_{n-1} = 3(h_{n-2}y'_{n-1} + h_{n-1}y'_{n-2}) - h_{n-2}s_n\]

or in matrix form

\[As = b\]

where
\[
A = \begin{bmatrix}
2(h_0 + h_1) & h_0 & 0 & 0 & 0 \\
h_2 & 2(h_1 + h_2) & h_1 & 0 & 0 \\
0 & h_3 & 2(h_2 + h_3) & h_2 & 0 \\
0 & 0 & 0 & \cdots & \vdots \\
0 & 0 & 0 & 0 & 2(h_{n-2} + h_{n-1})
\end{bmatrix}
\]

**diagonally dominant, symmetric, tridiagonal**

\[\Rightarrow\text{ no need to pivot}\]

\[\rightarrow\text{ can be solved quickly!}\]
Common choices:

• use known values of slopes $s_0, s_n$ at end $\rightarrow$ **clamped or complete cubic splines**

  then RHS is known $\Rightarrow$ solve for $s_1 \cdots s_{n-1}$

• $q''(x_0) = q''_{n-1}(x_n) = 0$ **natural cubic splines**

  $\Rightarrow f$ is linear outside $[x_0, x_n]$

• $q'''_0 = q'''_1, q'''_{n-2} = q'''_{n-1}$ **not-a-knot cubic splines** Matlab’s default

  1st and 2nd cubics are identical; last and 2nd last cubics are identical
Now use end conditions to express $s_0, s_n$ in terms of the rest:

◆ Example for natural splines

$$s_0 = \frac{1}{2}(3y'_0 - s_1)$$

$$s_n = \frac{1}{2}(3y'_n - s_{n-1})$$

substitute into RHS and rearrange; changes 1st equation to

$$(2h_0 + \frac{3}{2}h_1)s_1 + h_0s_2 = 3h_0y'_1 + \frac{3}{2}h_1y'_0$$

and last equation to

$$h_{n-1}s_{n-2} + (\frac{3}{2}h_{n-2} + 2h_{n-1})s_{n-1} = \frac{3}{2}h_{n-2}y'_{n-1} + 3h_{n-1}y'_{n-2}$$
Once we know $s_i$, can evaluate spline $S_n(x)$ anywhere

**Error in spline interpolation**

Let $h = \max_i h_i$

error in each piece $\leq f^{(4)}(\xi)h^4$

so if $f \in C^4[a,b]$ and $f^{(4)}$ is bounded

$$ |f(x) - S_n(x)| \leq cf^{(4)}(\xi)h^4 $$

an **error result** for fixed $\{x_i\}$
Also a convergence result

as $n$ increases, $h \to 0 \Rightarrow$

$$S_n(x) \to f(x)$$

since only $f^{(4)}$ enters, not $f^{(n+1)}$

Example Runge example by splines

Even better

$$|f^{(k)}(x) - S_n^{(k)}(x)| \leq Ch^{4-k}, \quad k = 0, 1, 2, 3$$

derivatives are also approximated (but not as well as $f$)
To evaluate a polynomial

from power form: use **nested multiplication**

\[ p_n(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + a_n x)) \cdots) \]

optimal \((n \times)\)

Matlab's polyval
To evaluate a spline

1. find the piece of the spline where evaluation point $x$ is
   e.g. binary search

2. evaluate the cubic piece
   by nested multiplication

Matlab’s `ppval`, `spline`
End of Lecture 12
This week: aim to cover

- Linear least squares
- Normal equations
- QR factorization
Data fitting: least squares

A very common task:

given a set of data \( \{x_i, y_i\}, i = 1 \cdots m \) with observational error

find a line \( y = a + bx \) that ‘fits’ the data

\[ \Rightarrow \text{we want } y_i = a + bx_i + \epsilon_i \]

If \( m = 2 \), can interpolate; what if \( m > 2 \)?
2 approaches: 1 from linear algebra, 1 from calculus (optimization)

Write $y_i = f(x_i)$

\[
\begin{align*}
    a + bx_1 &= y_1 \\
    a + bx_2 &= y_2 \\
    \vdots &= \vdots \\
    a + bx_m &= y_m
\end{align*}
\]
\[
\begin{bmatrix}
1 & x_1 \\
1 & x_2 \\
1 & x_3 \\
\vdots & \vdots \\
1 & x_m \\
\end{bmatrix}
\begin{bmatrix}
a \\
b \\
\end{bmatrix}
= 
\begin{bmatrix}
y_1 \\
y_2 \\
y_3 \\
\vdots \\
y_m \\
\end{bmatrix}
\]

a linear system $Ax = b$ with $A$ an $m \times 2$ rectangular matrix.

Since more rows than columns $\rightarrow$ **overdetermined system** $-$ unless $b$ is exceptional, there is no solution to such a system
there is no solution to such a system → what is the best we can do?

exact solution \( r = b - Ax = 0 \)

so let’s find \( x \) that minimizes residual \( r \) — but in what norm?

can do more statistics (inference) if we choose 2-norm

→ minimize \( \| r \|_2 \)
**Definition:** the least squares solution to $Ax = b$ where $m > n = \text{rank}(A)$ is the solution $x$ that minimizes $\| r \|_2$

In our case, $r_i = y_i - a - bx_i, \quad i = 1 \cdots m$

so minimize $\| r \|_2 \Rightarrow$ minimize $\| r \|_2^2 = r^T r = \sum_i (y_i - a - bx_i)^2 = S$

Necessary conditions:

$$\frac{\partial S}{\partial a} = \frac{\partial S}{\partial b} = 0$$
\[-2 \sum_i (y_i - a - bx_i) = 0 \Rightarrow \sum y_i = ma + b \sum x_i\]

\[-2 \sum_i x_i(y_i - a - bx_i) = 0 \Rightarrow \sum x_iy_i = a \sum x_i + b \sum x_iy_i\]

\[\rightarrow \text{a } 2 \times 2 \text{ matrix equation for } a, b\]

\[
\begin{bmatrix}
m & \sum x_i \\
\sum x_i & \sum x_i^2
\end{bmatrix}
\begin{bmatrix}
a \\
b
\end{bmatrix} =
\begin{bmatrix}
\sum y_i \\
\sum x_iy_i
\end{bmatrix}
\]
In terms of the overdetermined system, $Ax = b$, this square system is the **normal equations**

$$A^T Ax = A^T b$$

→ the simplest way to solve any overdetermined system is to solve the normal equations
Another derivation

In $Ax = b$, $b$ is a vector in $\mathbb{R}^m$; the columns of $A = [a_1 \ a_2]$ are each vectors in $\mathbb{R}^m$.

Any linear combination of those columns can be written $x_1a_1 + x_2a_2 = Ax$: these form a 2D subspace of $\mathbb{R}^m$.

We want to find a vector in this subspace (i.e. all vectors of the form $Ax$) as close to $b$ as possible. How?

To minimize $\|r\|_2$, we make $r \perp$ the subspace i.e. $r \perp$ the columns of $A$. 
\[ a_1^T r = 0; a_2^T r = 0 \text{ or } A^T r = 0 \]

\[ A^T r = A^T (b - Ax) = A^T b - A^T A x = 0 \]

→ the normal equations

⇒ Ax is the projection of b onto the subspace formed by columns of A

If A has rank 2, then \( A^T A \) is square and nonsingular

**Proof:**
So if $A$ has rank 2, we can write

$$x = (A^TA)^{-1}A^Tb$$

which is also written

$$x = A^\dagger b$$

where $A^\dagger = (A^TA)^{-1}A^T$ is the pseudoinverse of $A$

Of course, don’t compute inverse; just solve the normal equations

\[ Example \]
This can all be generalized to any linear model

i.e. fit to \( y = x_1\phi_1(t) + x_2\phi_2(t) + \cdots + x_n\phi_n(t) \) from data points \( \{X_i, Y_i\}, i = 1 \cdots m, m > n \)

- form the overdetermined system \( Ax = b \)

\[
A = \begin{bmatrix}
\phi_1(X_1) & \phi_2(X_1) & \cdots & \phi_n(X_1) \\
\phi_1(X_2) & \phi_2(X_2) & \cdots & \vdots \\
\vdots & \vdots & \ddots & \vdots \\
\phi_1(X_m) & \phi_2(X_m) & \cdots & \phi_n(X_m)
\end{bmatrix}
\quad b = \begin{bmatrix}
Y_1 \\
Y_2 \\
\vdots \\
Y_m
\end{bmatrix}
\]

- form the normal equations \( A^T Ax = A^T b \) as before
Since $A_{ij} = \phi_j(X_i)$, $(A^T A)_{ij} = \sum_k \phi_i(X_k) \phi_j(X_k)$

then, if $\text{rank}(A) = \min(m, n) = n$ (A is **full rank**), normal equations have a unique solution

\( \therefore \) Example polynomial curve fitting
But $A^TA$ is symmetric and positive definite

**Proof:**

⇒ solve using Cholesky factorization → takes $\approx n^3/6$ ops.

also must form $A^TA$, an $n \times n$ (symmetric) matrix with $n^2/2$ different entries, each one $\sum_k \phi_i(X_k)\phi_j(X_k)$ i.e. $m$ multiplies

→ $\frac{1}{2}n^2(m + \frac{1}{3}n)$ operations
End of Lecture 13
BUT solving the normal equations by Cholesky is NOT the recommended way to find the least squares solution - WHY NOT?

1. if A is ‘close to singular’, can get $A^T A$ singular

2. forming normal equations CAN worsen conditioning (sensitivity) of least squares problem

3. if A is rank-deficient then $A^T A$ is singular ⇒ can’t solve normal equations (Cholesky factors are singular)

Example
The 2-norm is the natural norm for LSQ problems (minimizing \( \| r \|_2 \))
\[ \Rightarrow \text{can no longer avoid matrix 2-norm} \]

for a square matrix \( A \) (see ‘MatrixNorms’ for proof)
\[
\| A \|_2 = \sqrt{\lambda_{\text{max}}(A^T A)}
\]

\( \lambda_{\text{max}}(A^T A) \) is the largest eigenvalue of \( A^T A \) (all eigenvalues are positive since \( A^T A \) is positive definite).

then
\[
\| A^{-1} \|_2 = \sqrt{1/\lambda_{\text{min}}(A^T A)}
\]
Proof:

\[ \kappa_2(A) = \sqrt{\frac{\lambda_{\max}(A^T A)}{\lambda_{\min}(A^T A)}} \]

This holds (no proof) even if \( A \) is \( m \times n \), as long as it's full rank.
In solving the normal equations, we solve a square system with matrix \( A^T A \) so condition is determined by \( \kappa_2(A^T A) \)

But

\[
\kappa_2(A^T A) = \kappa_2(A)^2
\]

Proof:
It turns out: condition number of LSQ problem is

- \( \approx \kappa_2(A) \) if the fit to the data is good (not much scatter)

- \( \approx \kappa_2(A)^2 \) if the fit to the data is poor (a lot of scatter)

⇒ using normal equations worsens conditioning of problem (if fit is good)
To solve problems 1, 2 use another matrix factorization — the **QR factorization**

To solve problem 3 use **QR factorization with column pivoting** (Matlab) or **singular value decomposition** — not in 381.
The idea of (reduced) QR factorization is:

- form a factorization

\[ A = QR \]

where Q is orthogonal \( m \times n \) matrix, R is upper triangular \( n \times n \) matrix i.e. \( Q^T Q = I_n \)

- to solve \( Ax = b \),

\[ QRx = b \Rightarrow Q^T QRx = Q^T b \Rightarrow Rx = Q^T b \]

so solve a triangular system for \( x \) in \( O(n^2) \) ops!
You’ve seen a QR factorization before (in disguise) in **Gram-Schmidt orthogonalization**:

given a set of linearly independent vectors $\{a_1, a_2 \cdots a_n\}$ forming an n-D subspace of $\mathbb{R}^m$, Gram-Schmidt orthogonalization produces a set of orthonormal vectors $\{q_1, q_2 \cdots q_n\}$, an orthonormal basis of the same subspace.

$$ A = QR $$

i.e. use triangular transformations to produce an orthogonal matrix. We don’t do it this way because it’s numerically unstable; instead we use orthogonal transformations to turn $A$ into $R$. 

212
Orthogonal transformations are good because:

- they involve perfectly-conditioned matrices
- they don’t change the conditioning of the problem
- they don’t change the solution of the LSQ problem
The QR factorization takes $n^2(m - n/3)$ ops

i.e. for $m \gg n, \approx$ twice as expensive as Cholesky factorization of normal equations

but allows us to handle a larger class of matrices.

Example

For square systems, can use QR (normwise backward stable) → takes $2n^3/3$ ops

twice as expensive as GEPP but no issues re growth factor etc.
QR in Matlab

I have described what Matlab calls ‘economy-size QR’ factorization.

\[ A = QR \]

where Q is orthogonal \( m \times n \) matrix, R is upper triangular \( n \times n \) matrix. This is all we need for the LSQ problem.

Matlab by default produces the ‘full QR’ factorization

\[ A = \bar{Q}\bar{R} \]

where \( \bar{Q} \) is orthogonal \( m \times m \) matrix, \( \bar{R} \) is upper triangular \( m \times n \) matrix.
\[ \bar{Q} = [Q \mid \text{extra orthog. cols}] \]

so that \( \bar{Q}^T \bar{Q} = I_m \). The extra columns are never used in the LSQ problem.

\[ \bar{R} = \begin{bmatrix} R \\ 0 \end{bmatrix} \]

Since \( A^T A = R^T Q^T QR = R^T R \), \( R \) is the Cholesky factor of \( A^T A \).

Hence to solve LSQ problem, we could:
1. \([q, r] = \text{qr}(A); x = r \backslash (q' \ast b)\);
   easiest to understand

2. \(r = \text{triu}(\text{qr}(A)); x = r \backslash (r' \backslash (A' \ast b))\);
   better since never need to form Q

3. \(x = A \backslash b\);
   \(\backslash\) acting on overdetermined system does the same as 2 (unless
   \(A^T A\) is rank-deficient)
End of Lecture 14
This week: aim to cover

- Numerical differentiation
- Newton-Cotes methods for quadrature
- Composite Newton-Cotes
Numerical differentiation

Sometimes we need to evaluate derivatives of functions e.g. Newton’s Method, solving DEs

If function is a known function → symbolic differentiation

use chain rule etc. or a Computer Algebra System

Example Mathematica, Maple

can get messy
If function is a computer program → use **automatic differentiation** → a program for the derivatives(s)!

If function given by tabulated data → use **numerical differentiation**

Generally, not a great thing to do, because **numerical differentiation is ill-conditioned**

Example: Change $f$ by $\delta f = \epsilon \sin \omega x$

⇒ (absolute) condition number $\kappa_\infty = \omega$ i.e. unbounded

using $\infty$-norm for functions $\|f\|_\infty = \sup_{a \leq x \leq b} |f(x)|$
Approximate derivatives by **Finite Difference Formulae**

Can derive by Taylor series or by **differentiating interpolants**

Example

recall that Finite Difference Formulae are susceptible to roundoff ⇒
don't let $h$ get too small

Example Lec2Deriv
Often the indices of \( \{x_i\} \) indicate a ‘direction’ e.g. of time

If derivative is expressed in terms of points ‘ahead’ of where you are → **Forward Difference** formulae

\[ f'(x_0) \approx p'(x_0) = \frac{f(x_1) - f(x_0)}{h} \]

Interpolation error → **truncation error** for FD formula

\[ f'(x_0) = \frac{f(x_1) - f(x_0)}{h} - \frac{h}{2} f''(\xi) \]
If derivative is expressed in terms of points on both sides of where you are → **Central Difference** formulae

\[ f'(x_1) \approx p'_2(x_1) = \frac{f(x_2) - f(x_0)}{2h} \text{ with truncation error } O(h^2) \]

If derivative is expressed in terms of points ‘behind’ where you are → **Backward Difference Formulae (BDF)**

\[ f'(x_2) \approx p'_2(x_2) = \frac{3f(x_2) - 4f(x_1) + f(x_0)}{2h} \]

with truncation error \( O(h^2) \)
Can do the same for higher derivatives but more susceptible to round-off $\Rightarrow$ can’t get same accuracy

\[ f''(x_1) \approx p''_2(x_1) = \frac{f(x_0) - 2f(x_1) + f(x_2)}{h^2} \]

with truncation error $O(h^2)$

or higher order formulae (smaller truncation error) from higher order $p_n(x)$

**but don’t go too high**

Or can differentiate spline.
Numerical integration = ‘Quadrature’

Aim: to find $I = \int_{a}^{b} f(x) \, dx$ ($a, b$ could be $\pm\infty$) where you can’t find the indefinite integral of $f$.

By contrast, finding integrals is difficult symbolically, even with a CAS, but numerical integration is well-conditioned.

Change $f$ by $\delta f$, changes $I$ by $\delta I = \int_{a}^{b} \delta f(x) \, dx$

$$\left| \delta I \right| \leq \int_{a}^{b} |\delta f(x)| \, dx = \|\delta f\|_1$$

(absolute) condition number $\kappa_1 = 1$ i.e. well-conditioned

using 1-norm for functions $\|f\|_1 = \int_{a}^{b} |f(x)| \, dx$
Sometimes you may only know $f$ at a set of points $\{x_i\}$; sometimes you can evaluate $f$ wherever you want.

We typically evaluate integrals by quadrature formulae of the form

$$I \approx \sum_i w_i f(x_i)$$

This has the form of a scalar product $\Rightarrow$ backward stable wrt rounding errors (see MachineNumbers.pdf)

$\Rightarrow$ not susceptible to rounding error if terms are all same sign (no cancellation)

The 1st basic idea is: **interpolate $f$ then integrate the interpolant**
Newton-Cotes methods

The simplest rules use polynomial interpolants which interpolate the endpoints $a, b$ and equispaced internal points → closed Newton-Cotes formulae

Simplest case: for $I = \int_{a}^{b} f(x) \, dx$, approximate $f$ by $p_1(x)$, the linear interpolant though the endpoints. Then

$$I = \int_{a}^{b} f(x) \, dx \approx \int_{a}^{b} p_1(x) \, dx = \sum_{0}^{1} f(x_i) \int_{a}^{b} l_i(x) \, dx$$

$$= f(a) \int_{a}^{b} l_0(x) + f(b) \int_{a}^{b} l_1(x) = \frac{b - a}{2} [f(a) + f(b)]$$

→ trapezoid rule
The **error in the trapezoid rule** can be estimated from the error of polynomial interpolation \( f(x) = p_1(x) + \frac{f''(\xi_x)}{2!}(x - a)(x - b) \)

\[
I = \int_a^b f(x) \, dx = \frac{b - a}{2} [f(a) + f(b)] + \frac{1}{2} \int_a^b f''(\xi_x)(x - a)(x - b) \, dx
\]

so error is (using generalized Mean Value Theorem)

\[
\frac{1}{2} \int_a^b f''(\xi_x)(x - a)(x - b) \, dx = \frac{1}{2} f''(\xi) \int_a^b (x - a)(x - b) \, dx = -\frac{1}{12} f''(\xi) (b - a)^3
\]

or, writing \( a = x_0, b = x_1, x_1 = x_0 + h \)

\[
\int_{x_0}^{x_1} f(x) \, dx = \frac{h}{2} [f(x_0) + f(x_1)] - \frac{1}{12} f''(\xi) h^3
\]

This error formula OK if \( f \in C^2[a, b] \). Note error \( \sim h^3 \). This is a **truncation error** (replacing an integral by sum).
Why stop at the linear interpolant?

Set $x_0 = a, x_1 = x_0 + h, x_2 = x_0 + 2h = b, h = (b - a)/2$

$$\int_a^b f(x) \, dx \approx \int_a^b p_2(x) \, dx = \sum_0^2 f(x_i) \int_a^b l_i(x) \, dx = \frac{h}{3}[f(x_0) + 4f(x_1) + f(x_2)]$$

→ **Simpson’s rule**

with error (more complicated derivation) $-\frac{1}{90}f^{(4)}(\xi)h^5, \xi \in [a, b]$.

This error formula OK if $f \in C^4[a, b]$ i.e. need a smoother $f$.

Also Simpson’s rule exact if $f$ is cubic even though we only used $p_2!!$
End of Lecture 15
We can extend this procedure:

interpolate on equispaced points between $a, b$ including endpoints; integrate interpolant $\rightarrow$ \textbf{quadrature rule}

\[ I \approx \sum_{i} w_i f(x_i) \]

with \textbf{quadrature weights} $w_i$.

\begin{itemize}
  \item Example using $p_3 \rightarrow$ \textbf{Simpson’s 3/8 rule} with error $\sim h^5$ and exact for cubics (like normal Simpson’s rule)
  \item In practise, don’t go beyond $n = 8$ since then weights start to change sign $\rightarrow$ poss. increased error due to subtractive cancellation
\end{itemize}
Note: methods using even degree $p_n$ have extra accuracy

i.e.  $p_2 \rightarrow$ error $\sim h^5$;  $p_3 \rightarrow$ error $\sim h^5$;  $p_4 \rightarrow$ error $\sim h^7$ etc.

Can also devise rules which don’t use endpoints, only interpolate at interior points $\rightarrow$ open Newton-Cotes formulae

\begin{align*}
\text{Example use } p_0 \text{ at } x = \frac{a+b}{2} \rightarrow \\
\int_a^b f(x) \, dx \approx \int_a^b p_0(x) \, dx = f\left(\frac{a+b}{2}\right) \int_a^b dx = hf\left(\frac{a+b}{2}\right) \\
\rightarrow \text{midpoint rule}
\end{align*}

with error $\frac{1}{24} f^{(2)}(\xi) h^3$, $\xi \in [a, b]$. 
Composite Newton-Cotes

Instead of using a global interpolant, can use piecewise interpolant.

Actually more natural for definite integrals, since can partition an integral

\[ I = \int_a^b f(x) \, dx = \sum_i \int_{x_i}^{x_i+1} f(x) \, dx \]

in any way, with no error.

Most common: use equispaced points in \([a,b]\), use Newton-Cotes formulae for each piece and add results together →

**composite Newton-Cotes formulae**
Simplest case:

approximate $f$ by piecewise linear interpolant, use NC with $p_1 = \text{trapezoid rule on each piece}$ →

$$I = \frac{h}{2}[f(x_0) + f(x_1)] + \frac{h}{2}[f(x_1) + f(x_2)] + \cdots + \frac{h}{2}[f(x_{n-1}) + f(x_n)]$$

$$I = \frac{h}{2}[f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(x_n)]$$

**composite trapezoid rule**

to get error: add the errors from each piece
error = \sum_{j=1}^{n} \frac{-h^3}{12} f''(\xi_j) = \frac{-h^3}{12} \left[ \frac{1}{n} \sum_{j=1}^{n} f''(\xi_j) \right] = \frac{-(b-a)h^2}{12} \left[ \frac{1}{n} \sum_{j=1}^{n} f''(\xi_j) \right]

Now, if \( f \in C^2[a, b] \)

\[
\min_{x \in [a, b]} f'' \leq \frac{1}{n} \sum_{j=1}^{n} f''(\xi_j) \leq \max_{x \in [a, b]} f''
\]

so by IVT

\[
\frac{1}{n} \sum_{j=1}^{n} f''(\xi_j) = f''(\xi)
\]

for some \( \xi \in [a, b] \).

\( \Rightarrow \) error in composite trapezoid rule

\[
E_T = \frac{-(b - a)h^2}{12} f''(\xi)
\]
Note: overall error $\sim h^2$ — the **global truncation error**

comes from adding up $n$ **local truncation errors** $\sim h^3$ each

BUT to use this form, need to know e.g. bound for $f'''$.

For another form for error, look at limit $h \to 0 \equiv n \to \infty$
\[
\lim_{n \to \infty} \frac{E_T}{h^2} = -\frac{1}{12} \lim_{n \to \infty} \sum h f''(\xi_j)
\]

a Riemann sum $\to$ integral

\[
= -\frac{1}{12} \int_a^b f''(x) \, dx = -\frac{1}{12} [f'(b) - f'(a)]
\]

\[
E_T = -\frac{h^2}{12} [f'(b) - f'(a)] + o(h^2)
\]
as $h \to 0$

**asymptotic error formula**

Note: if $f$ is periodic with period $b - a$, $f'(b) = f'(a)$

$\Rightarrow$ trapezoid rule unusually accurate if $f$ is periodic
Next simplest case:

approximate $f$ by piecewise quadratic interpolant, use NC with $p_2 = \text{Simpson's rule on each pair of pieces}$ →

$$I = \frac{h}{3} [f(x_0) + 4f(x_1) + f(x_2)] + \frac{h}{3} [f(x_2) + 4f(x_3) + f(x_4)] + \cdots + \frac{h}{3} [f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)]$$

composite Simpson's rule
By same argument, error comes from adding error in each piece

\[ E_S = \sum_{j=1}^{n/2} \frac{-h^5}{90} f^{(4)}(\xi_j) = \frac{-h^5 n}{180} \left[ 2 \sum_{j=1}^{n/2} f^{(4)}(\xi_j) \right] = \frac{-(b - a) h^4}{180} f^{(4)}(\xi) \]

As before, this is valid if \( f \in C^4[a, b] \) and the global error is \( O(h^4) \) cf. local error \( O(h^5) \)

As before, can get an asymptotic error formula

\[ E_S = -\frac{(b - a) h^4}{180} [f'''(b) - f'''(a)] + o(h^5) \]

as \( h \to 0 \)
Example

We could go on, but instead, how about integrating piecewise cubic interpolants e.g. spline?

It turns out you only do about as well as Simpson’s rule!

Moral: don’t get much benefit from using smoother interpolant when integrating

because integration smooths out roughness in the integrand
Extrapolation Methods

The other method of improving on trapezoid/Simpson’s rule takes advantage of the asymptotic error structure.

Recall NC rules with \( n \) panels:

- trapezoid rule \( I \approx I_n + C_1 h^2 \) if \( f \in C^2 \)
- Simpson’s rule \( I \approx I_n + C_2 h^4 \) if \( f \in C^4 \)

what if \( f \) aren’t so smooth?

⇒ we’ll assume quadrature rules have asymptotic error behaviour

\[
I \approx I_n + Ch^p = I_n + Cn^{-p}
\]
**Aitken extrapolation**

If we don’t know value of $p$ e.g. $f$ not smooth enough, can proceed as follows.

Assume asymptotic behaviour is exact $\Rightarrow$ solve for $p, I \rightarrow$ improved estimate

Assume $I = I_n + C n^{-p}$, generate a sequence of estimates $I_n, I_{2n}, I_{4n} \cdots$

\[
\begin{align*}
I &= I_n + C n^{-p} \\
I &= I_{2n} + C (2n)^{-p} \\
I &= I_{4n} + C (4n)^{-p}
\end{align*}
\]

3 equations for $I, p, C$
Solving for $p$:

\[ 2^p = \frac{I_{2n} - I_n}{I_{4n} - I_{2n}} \]

Solving for $I \rightarrow$ new estimate $\hat{I}_{4n}$:

\[ \hat{I}_{4n} = I_{4n} - \frac{(I_{4n} - I_{2n})^2}{(I_{4n} - 2I_{2n} + I_n)} \]

This new estimate is not exact, but is usually so much better that we estimate error in $I_{4n}$ by

\[ I - I_{4n} \approx \hat{I}_{4n} - I_{4n} \]
This method works even if \( f \) is not smooth enough that we know value of \( p \)

\( \triangledown \) Example

but if we do know \( p \), can do it more simply ...
Richardson extrapolation

\[ I = I_n + Cn^{-p} \]

\[ I = I_{2n} + C(2n)^{-p} \]

2 equations for \( I, C \) (\( p \) assumed known)

Solving for \( I \rightarrow \) new estimate \( \hat{I}_{2n} \):

\[ \hat{I}_{2n} = I_{2n} + \frac{1}{2^p - 1}(I_{2n} - I_n) \]

This new estimate is not exact, but is usually so much better that we estimate error in \( I_{2n} \) by

\[ I - I_{2n} \approx \hat{I}_{2n} - I_{2n} \]
Example trapezoid rule $p = 2$

Example Simpson’s rule $p = 4$
End of Lecture 16
This week: aim to cover

- Adaptive quadrature
- Gaussian quadrature
- Improper integrals
In fact this is a quick way to do Simpson’s rule since

\textbf{trapezoid rule} + \textit{Richardson extrapolation} \rightarrow \textbf{Simpson’s rule}

so why not keep going? Is it true that

\text{Simpson’s rule} + \text{Richardson extrapolation} \rightarrow O(h^6) \text{ method?}

If $f$ has many cts derivatives, the answer is YES, due to a deep result about the asymptotic error behaviour ...
Euler-Maclaurin summation formula

\[ I = I_n^T + C_1 h^2 + C_2 h^4 + C_3 h^6 + \cdots \]

i.e. error in trapezoid rule has asymptotic expansion in **even powers** of \( h \) where

\[ C_k \propto f^{(2k-1)}(a) - f^{(2k-1)}(b) \]

⇒ if \( f \) is periodic with period \( b - a \) and \( C^\infty \), error decays faster than any power of \( h \)

actually error \( \sim \exp(-\alpha/h) \)

**Trapezoid rule is optimal for this special class of functions!**
Romberg quadrature

Hence, if integrand is smooth enough, can continue extrapolating with $p = 2, 4, \cdots$ until accuracy tolerance is satisfied.

A systematic way to do this is called Romberg integration but we won’t say more, since it only works for very smooth integrands.
Adaptive Quadrature

But using a NC rule with fixed \( h \) is like root-finding with fixed number of iterations!

⇒ better to find an algorithm so we can specify a tolerance and let the algorithm figure out what \( h \) to use

In particular, no need to use same \( h \) everywhere — let the algorithm decide where to evaluate \( f \) to greatest effect

→ Adaptive Quadrature — must be able to evaluate \( f \) anywhere!

simplest kind uses Simpson’s rule as a basic rule e.g. Matlab’s quad
Basic idea

- Use a basic quadrature rule so that
  - can get 2 estimates of $I$ easily e.g. $I_n^S$ and $I_{2n}^S$
  - can get error estimate e.g. from extrapolation

- Start on the whole interval:
  - if error estimate $< \text{tolerance}$, return better estimate e.g. $I_{2n}^S$ or extrapolated estimate
• if not, subdivide interval into 2 subintervals of half the width and repeat the above on each subinterval recursively

Need a recursion limit to avoid infinite recursion

What tolerance to use on each subinterval? Matlab used to use \( tol/2^L \) where \( L \) is the recursion level (very conservative)

but now uses \( tol \) : usually only one part of the integral causes all the trouble.

Once tolerance is met, keeps as estimate Richardson extrapolant of Simpson’s estimate \( = O(h^6) \) rule for smooth \( f \).
To understand process, I suggest:

- read code of quadtx

- run numerous examples with quadgui

Example
NC methods → quadrature rules $I \approx \sum_{1}^{m} w_i f(x_i)$

1. evaluate $f$ at equispaced points ⇒ can be used on [tabular data]

2. an $m$ point rule can integrate polynomials of degree $\leq m$ exactly (if $m$ is odd)
   
   e.g. Simpson’s rule: 3 point rule integrates cubics exactly

Can do better if we choose both weights $w_i$ and abscissae $x_i$ carefully

→ can integrate a larger class of polynomials exactly
**Gaussian Quadrature**

must be able to evaluate $f$ anywhere (like Adaptive Quadrature)

Start with

$$I = \int_{a}^{b} f(x) \, dx$$

Consider only $I = \int_{-1}^{1} w(x)f(x) \, dx$; $w(x) > 0$, $w$ is a **weight function**

can always do this by change of variable

$$t = \frac{2x - (a + b)}{b - a}$$
and writing
\[
\int_{-1}^{1} g(x) \, dx = \int_{-1}^{1} w(x) \frac{g(x)}{w(x)} \, dx = \int_{-1}^{1} w(x) f(x) \, dx
\]

Now look for a quadrature rule
\[
\int_{-1}^{1} w(x) f(x) \, dx \approx \sum_{i=1}^{m} w_i f(x_i)
\]
where now we can choose both \( \{w_i\} \) and \( \{x_i\} \) to give ‘best approx.’

We choose them to integrate exactly the **largest class of polynomials possible**.
We have $2m$ unknowns

$\Rightarrow$ need $2m$ conditions to specify them

$\Rightarrow$ make integral **exact for polynomials of degree** $2m - 1$

$\rightarrow$ system of nonlinear equations for $\{w_i\}$ and $\{x_i\}$

$\triangleleft$ Example $m = 2, w(x) = 1$
End of Lecture 17
→ 2-point rule that integrates cubics exactly

⇒ composite Gaussian quadrature an attractive method if integrand is piecewise polynomial

e.g spline or Hermite interpolant, finite element solution.
Same sort of thing can be done for integrals like

\[ \int_{-1}^{1} \frac{f(x)}{\sqrt{1 - x^2}} \, dx; \int_{0}^{2\pi} \sin(x)f(x) \, dx \]

Many of these quadrature rules have been solved and weights etc. collected in Tables or (nowadays) software.

Instead of solving by brute force, can use properties of orthogonal polynomials $\rightarrow$ expressions for $x_i$, etc.
For a given interval \([a, b]\) and weight function \(w \geq 0\), a set of functions \(\{\phi_0, \phi_1, \cdots \phi_n\}\) is **orthogonal with respect to** \(w(x)\) iff

\[
\int_{a}^{b} w(x)\phi_i(x)\phi_j(x) \, dx = \gamma_i \delta_{ij}
\]

i.e. we define an **inner product**

\[
(f, g) \equiv \int_{a}^{b} w(x)f(x)g(x) \, dx
\]

then \(\{\phi_0, \phi_1, \cdots \phi_m\}\) forms an orthogonal set iff

\[
(f_i, f_j) = \gamma_i \delta_{ij}
\]

Example \([a, b] = [-1, 1], w(x) = 1\)

\(\Rightarrow\) orthogonal set of polynomials \(\{P_k(x)\}\): the **Legendre polynomials**
Now suppose we want to use an m-point rule to integrate $2m - 1$ degree polynomial exactly for $[a, b] = [-1, 1], w(x) = 1$.

Let $P(x)$ be an arbitrary polynomial of degree $k \leq 2m - 1$. $P$ can be written as

$$P(x) = Q(x)P_m(x) + R(x)$$

where $Q(x), R(x)$ must be polynomials of degree $< m$ (like doing long division of polynomials)

We want to integrate $P$ exactly.

$$\int_{-1}^{1} P(x) \, dx = \int_{-1}^{1} Q(x)P_m(x) \, dx + \int_{-1}^{1} R(x) \, dx$$
Since $Q$ is of degree $< m$, it can be written

$$Q(x) = \sum_{k=0}^{m-1} a_k P_k(x)$$

since the Legendre polynomials form a (orthogonal) basis for $\mathbb{P}_m$.

$$\int_{-1}^{1} Q(x) P_m(x) \, dx = \sum_{k=0}^{m-1} a_k \int_{-1}^{1} P_k(x) P_m(x) \, dx = 0$$

so

$$\int_{-1}^{1} P(x) \, dx = \int_{-1}^{1} R(x) \, dx$$

But we already know how to integrate a degree $m - 1$ polynomial exactly with an $m$-point rule — pick any $m$ points and integrate the (Lagrange) interpolating polynomial.
Hence
\[ \int_{-1}^{1} P(x) \, dx = \int_{-1}^{1} R(x) \, dx = \sum_{k=1}^{m} w_k R(x_k) \]
for any points \( x_k \) and weights \( w_k = \int_{-1}^{1} l_k(x) \, dx \).

But we want an \( m \)-point rule involving \( P \), not \( R \)!

Since \( P(x_k) = Q(x_k)P_m(x_k) + R(x_k) \) we can achieve this by choosing \( \{x_k\} \) to be the \( m \) roots of \( P_m(x) \).

This is always possible because (no proof):

If \( \{\phi_0, \phi_1, \cdots \phi_m\} \) are an orthogonal set of polynomials over \([a, b]\), then each \( \phi_k \) has \( k \) distinct roots in \((a, b)\).
Then

\[ \int_{-1}^{1} P(x) \, dx = \sum_{k=1}^{m} w_k R(x_k) = \sum_{k=1}^{m} w_k P(x_k) \]

i.e. this \( m \)-point rule integrates polynomials of degree \( \leq 2m - 1 \) exactly if the \( \{x_k\} \) are the roots of \( P_m(x) \) and the weights are given by \( w_k = \int_{-1}^{1} l_k(x) \) where \( l_k(x) \) are constructed using the \( \{x_k\}, k = 1..m \).

\( \triangle \) Example \( m = 2 \)

\[ P_2(x) = x^2 - \frac{1}{3} \] so \( x_1, x_2 = \pm \frac{1}{\sqrt{3}} \)

\( \Rightarrow w_1, w_2 = 1 \) as before.
This idea works in general:

if $\{\phi_0, \phi_1, \cdots \phi_m\}$ is a set of orthogonal polynomials over $[a, b]$ w.r.t. $w(x)$ then the quadrature rule

$$\int_a^b w(x)f(x) \, dx \approx \sum w_k f(x_k)$$

integrates polynomials of degree $\leq 2m-1$ exactly if $\{x_k\}$ are the roots of $\phi_m(x)$ and the weights $w_k = \int_{-1}^{1} l_k(x) \, dx$.

We call any such method: **Gaussian quadrature**
## Common Gaussian Quadrature rules

<table>
<thead>
<tr>
<th>Interval</th>
<th>Weight Function $w(x)$</th>
<th>Orthogonal Polynomial $\phi_m(x)$</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[-1, 1]$</td>
<td>$1$</td>
<td>$P_m(x)$</td>
<td>Gauss-Legendre</td>
</tr>
<tr>
<td>$[-1, 1]$</td>
<td>$\frac{1}{\sqrt{1-x^2}}$</td>
<td>$T_m(x)$</td>
<td>Gauss-Chebyshev</td>
</tr>
<tr>
<td>$[0, \infty]$</td>
<td>$e^{-x}$</td>
<td>$L_m(x)$</td>
<td>Gauss-Laguerre</td>
</tr>
<tr>
<td>$[-\infty, \infty]$</td>
<td>$e^{-x^2}$</td>
<td>$H_m(x)$</td>
<td>Gauss-Hermite</td>
</tr>
</tbody>
</table>
Examples of simple usage: remember to change interval!!

Example
Gaussian quadrature cf. NC rules?

Example
Error of Gauss-Legendre quadrature

Hard to derive but for smooth functions, Gaussian quadrature usually converges very fast as $m \to \infty$

\[ \int_{-1}^{1} f(x) = \sum w_i f(x_i) + C 4^{-m} \]

\Rightarrow error decreases **exponentially** with $m$

cf. trapezoid rule: $E_T \sim m^{-2}$; Simpson’s rule: $E_S \sim m^{-4}$
Can we use Gaussian quadrature in adaptive quadrature?

YES but takes some work:

- unlike Simpson’s rule, $I_m, I_{2m}$ have no evaluation points in common → more work to get 2 estimates

  → construct **Kronrod extension** of the Gaussian rule: a higher order quadrature rule that re-uses the \( \{x_k\} \) of the Gaussian rule (with different weights) and adds some extra evaluation points (and weights)

- can’t use extrapolation methods to estimate error so need other ways to estimate error from 2 quadrature estimates
Example Mathematica’s NIntegrate uses adaptive Gauss-Kronrod quadrature

Variants of Gaussian quadrature:

If we demand 1 endpoint is an evaluation point $\rightarrow$ Gauss-Radau quadrature: $m$-point rule integrates polynomial of degree $2m - 2$ exactly

If we demand both endpoints are evaluation points $\rightarrow$ Gauss-Lobatto quadrature: $m$-point rule integrates polynomial of degree $2m - 3$ exactly

Example Matlab’s quadl uses adaptive Gauss-Lobatto–Kronrod quadrature
Improper integrals

2 types of improper integrals:

1. \( f(x) \) unbounded in the interval \([a, b]\)

2. infinite endpoints

We assume that the (improper) integral exists.

No single method works in all cases \(\Rightarrow\) give some general strategies
Singular integrands

Function is well-behaved except it would return $\frac{0}{0}$

a removable singularity

- just redefine function
  - Example

- or use an open quadrature rule
  - Example
Integrable singularity at an endpoint

- Change variable
  - Example

- For some special singularities, use a Gaussian quadrature rule
  - Example

- Treat the singularity analytically
  - Example
Integrable singularity internal to the interval

Split the integral at the singularity

Integrable singularity at unknown position in $[a, b]$

Find out more about $f!!$
Infinite intervals

Change variable

Example
Gaussian quadrature

Example
Truncation

Example
End of Lecture 18
This week: aim to cover

- General properties of IVP solvers
- Euler’s method
- Errors in Euler’s method
Differential Equations

One of the most useful numerical techniques — numerical solution of differential equations.

We do only ODEs ⇒ unknown functions $y_i$ depend only on 1 dependent variable $t$.

Recall that any $n^{th}$ order ODE can be written as a system of 1st order ODEs ⇒ any system of ODEs can be written as a system of 1st order ODEs.

$$\frac{dy_i}{dt} = f_i(t, y_1, \cdots y_n) \quad i = 1..n$$
or in vector form

\[ \frac{dy}{dt} = f(t, y) \]

**Special cases**

If \( f = f(y) \) only, system is **autonomous** (else **nonautonomous**)

If \( f = A(t)y + b(t) \), system is **linear**

If \( b = 0 \), linear system is **homogeneous**

If \( A \) is constant, linear system is **constant-coefficient system**

If only 1 ODE, equation is **scalar**
If the initial conditions

\[ y_i(t_0) = \alpha_i \]

are given at one value of \( t \) (e.g. \( t_0 = 0 \)), we have an Initial Value Problem.

Otherwise, a Boundary Value Problem.

We only cover IVPs.
There are some special methods for 2nd order systems

Example Newton’s equations of motion

\[
\begin{align*}
\ddot{x}_i &= F_i^x(t, x_1, y_1, \cdots z_N) \\
\ddot{y}_i &= F_i^y(t, x_1, y_1, \cdots z_N) \\
\ddot{z}_i &= F_i^z(t, x_1, y_1, \cdots z_N)
\end{align*}
\]

3N 2nd order equations in 3N unknowns → Molecular Dynamics

but usually solve as a system of 1st order ODEs.
The methods used for a system are basically the same as those for a single 1st order ODE

⇒ we lose little by discussing a scalar ODE

\[
\frac{dy}{dt} = f(t, y) \\
y(t_0) = y_0
\]
Finding a numerical solution means:

find a set of values \([y_k]\) at some set of output points \([t_k]\).

Distinguish between the true value: \(y(t_k)\)

and numerical approximation : \(y_k\)

Need numerical methods since most ODEs can’t be solved analytically

\[\text{Example } y' = y^3 + t^2 \text{ can’t be done by Maple.}\]
Some Theory: Existence and uniqueness:

If $f(t, y)$ is cts in $t \in [t_0, t_f]$ and $y \in \mathbb{R}$ and satisfies a **Lipschitz condition**

$$| f(t, y) - f(t, \hat{y}) | \leq L | y - \hat{y} |$$

⇒ the IVP has a **unique solution** for $t \in [t_0, t_f]$.

$\frac{\partial f}{\partial y}$ bounded ⇒ satisfies Lipschitz

Most commonly fails because $\frac{\partial f}{\partial y}$ not bounded for large $y$.

◁ Example
Sensitivity of IVP

The sensitivity of IVPs is quite complicated: we ask for (absolute) change in solution at $t$ due to change in IC or RHS $\rightarrow$ a condition number $\kappa(t)$.

Example: for a scalar nonautonomous ODE $\frac{dy}{dt} = f(t, y)$

$$| \delta y(t) | \leq \exp[\int_{t_0}^{t} J(s) \, ds] \cdot | \delta y_0 |$$

where $J(t) = \frac{\partial f}{\partial y}$ is the Jacobian for this problem

$\Rightarrow \kappa(t) = \exp[\int_{t_0}^{t} J(s) \, ds]$ in this case.
**New feature**: conditioning depends on $t$, can start well-conditioned and become ill-conditioned! or vice versa!

⇒ if $J(t) < 0$ for $t \in [t_0, t_f] \Rightarrow \kappa(t) < 1$ i.e. well-conditioned

$J(t) < 0 \Rightarrow$ neighbouring solutions approach each other (contractive)

Example : for autonomous linear system $\frac{dy}{dt} = Ay + b$

then $J = A \Rightarrow \kappa(t) = \kappa(S) \exp[t\Re(\lambda_{max})]$ in this case

where $S$ is the matrix of eigenvectors of $J$ and $\lambda_{max}$ is its largest eigenvalue.

If $\kappa(S) \gg 1$ but $\Re(\lambda_{max}(A)) < 0$ then starts out ill-conditioned and becomes better!
Stability of IVP

At large times, we can ask whether initially nearby solutions stay nearby (stable) or approach each other (asymptotically stable). This is given by the long time behaviour of $\kappa(t)$.

Example: for autonomous linear system

$$\kappa(t) = \kappa(S) \exp[t \Re(\lambda_{max})]$$

so if $\Re(\lambda_{max}(A)) < 0 \Rightarrow \kappa(t) \rightarrow 0$ as $t \rightarrow \infty$

$\Rightarrow$ autonomous linear system asymptotically stable if $\Re(\lambda_{max}(A)) < 0$

Warning: doesn’t generalize immediately to nonautonomous or nonlinear cases!
All methods start at $t = t_0$ using the IC $y = y_0$

then march a distance $h$ in $t$

$t_0 \to t_0 + h \equiv t_1$

$y_0 \to y_1 \approx y(t_1)$

At $t = t_1$ we have another IVP,

\[
\frac{dy}{dt} = f(t, y) \\
y(t_1) = y_1
\]

so just repeat this procedure until $t = t_f$.

Called **time-stepping**.
If $h$ is constant $\rightarrow$ **fixed step method**

If $h$ is changed from step to step $\rightarrow$ **variable step method**

If, to get from $t_n$ to $t_{n+1}$ we only use $y_n$ $\rightarrow$ **1 step method**

If, to get from $t_n$ to $t_{n+1}$ we use previous values $y_{n-1}, y_{n-2} \ldots$ $\rightarrow$ **multistep method**

We start with fixed step methods but modern codes are usually variable step.
What method to choose depends on:

- discretization error
- stability properties of method
- efficiency e.g. number of func. evals
- ease of use
The simplest method: Euler’s Method

Start with $y_0$. We know $y'(t_0) = f(t_0, y_0)$ so step a distance $h$ with that slope:

$$y_1 = y_0 + hf(t_0, y_0)$$

Now repeat, using in general

**Euler’s Method**

$$y_{n+1} = y_n + hf(t_n, y_n)$$

We are approximating the DE by solving a difference equation → discretization error
3 derivations:

1. Taylor series $\rightarrow$ Runge-Kutta methods (1-step)

2. Numerical differentiation $\rightarrow$ BDF methods (multistep)

3. Quadrature $\rightarrow$ Adams methods (multistep)
1. Taylor series → **local error** $\sim h^2$

2. Use Forward Difference to approximate $y'$
   → **truncation error** $\sim h^2$

3. 
   $$y(t_{n+1}) - y(t_n) = \int_{t_n}^{t_{n+1}} f(\tau, y(\tau)) \, d\tau$$

   use Left Hand rectangle rule! → **truncation error** $\sim h^2$
Let’s try it on some problems:
We observe:

1. the **global error** \( y(t_k) - y_k \) appears to be \( \propto h \) (if method works at all)

2. sometimes it blows up

3. sometimes error grows with \( t \), sometimes not
End of Lecture 19
Errors in Euler’s Method

Really want to understand \textit{global error}

\[ GE_k \equiv y(t_k) - y_k^{EM} \]

Taylor Series ⇒

\[ y(t_{k+1}) = y(t_k) + h y'(t_k) + h^2 y''(\xi)/2 \]
\[ = y(t_k) + h f(t_k, y(t_k)) + h^2 y''(\xi)/2 \]  \hspace{1cm} (3)

\[ y_{k+1} = y_k + h f(t_k, y_k) \]  \hspace{1cm} (5)

EM ⇒

297
\begin{align*}
(2)-(3) \Rightarrow \\
GE_{k+1} = GE_k + h[f(t_k, y(t_k)) - f(t_k, y_k)] + h^2 y''(\xi)/2 \quad (6)
\end{align*}

\text{MVT} \Rightarrow 
\begin{align*}
f(t_k, y(t_k)) - f(t_k, y_k) &= \frac{\partial f}{\partial y} |_{\xi} (y(t_k) - y_k) \\
\equiv J(\xi)GE_k \quad (7)
\end{align*}

298
(4) and (6) \Rightarrow

GE_{k+1} = (1 + hJ)GE_k + h^2 y''(\xi)/2 \quad (9)

In words

Global error at \(t_{k+1}\)

\(= (1 + hJ) \times (\text{Global error at } t_k) \text{ plus local error (LE) at each step.} \)

\((1 + hJ)\) determines error propagation \(\rightarrow\) stability of method
We want errors to dampen out, if possible

⇒ \(| 1 + hJ | < 1\)

Note: errors **must grow if** \( J > 0 \)

\( J > 0 \) ⇒ solutions nearby are diverging and DE is **unstable wrt ICs**

**MORAL:** Absolute errors will **grow** in a region where \( J > 0 \)

⇒ don’t expect accurate numerical solution if \( J \gg 0 \)
\[ J < 0 \Rightarrow \text{solutions nearby are contractive and DE is stable wrt ICs} \]

Here the LE made at each step is damped thereafter
Convergence of Euler’s Method

We would like $GE_k \to 0$ as $h \to 0$.

Same as asking that the Forward Error $\to 0$ as $h \to 0$.

We can do this directly for Euler’s method but I’ll do it in 2 steps:

1. show the Backward Error (residual) $\to 0$ as $h \to 0$. consistency

2. show the discrete problem has finite condition number as $h \to 0$. 0-stability
then use a Big Theorem

\[ \text{Consistency} + \text{Stability} \rightarrow \text{Convergence} \]

that is true for ODE methods and time-dependent PDE methods!

Roughly for ODEs: any sensible method is consistent but not all are 0-stable!
Consistency

A method is **consistent** if exact solution satisfies the difference equation in the limit $h \to 0, n \to \infty, t = t_0 + nh$ fixed.

To check this, we find the **residual** $R$ by substituting the exact solution in to the method: for Euler’s Method

$$R = y(t_{n+1}) - y(t_n) - hf(t_n, y(t_n)) = \frac{1}{2}h^2y''(\xi)$$

$R$ is the **local truncation error**.

For 1-step methods, $\text{LTE} = \text{LE}$. 
We want this residual to measure the backward error: is the exact solution a solution of the difference equation with a different $f$? The answer is YES if we replace $f$ with $f + \frac{R}{h}$ — so the backward error is really $R/h$.

**Definition:** A method is **consistent** if $\lim_{h \to 0} \frac{R}{h} = 0$ or in little-o notation $R = o(h)$.

In our case, $R/h \sim h \to 0 \Rightarrow$ Euler’s Method is consistent.

**Definition:** A method is **consistent of order** $p$ if $\lim_{h \to 0} \frac{R}{h} = O(h^p)$ \Rightarrow Euler’s Method is consistent of order 1.
0-stability

Now need to check that discrete problem doesn’t go ill-conditioned as $h \to 0$. Suppose we change IC by $\delta_0$ and at step $k$ change the RHS by $\delta_k$, is the change in the solution finite?

\[ u_{n+1} = u_n + hf(t_n, u_n); u_0 = A \]
\[ v_{n+1} = v_n + h[f(t_n, v_n) + \delta_n]; v_0 = A + \delta_0 \]
\[ \rightarrow \text{Eq. for the change } v - u \]
\[ e_{n+1} = e_n + h[f(t_n, v_n) - f(t_n, u_n) + \delta_n]; e_0 = \delta_0 \]
\[ |e_{n+1}| \leq |e_n|(1 + hL) + h\delta; e_0 = \delta_0 \]

assuming $f$ is Lipschitz and $|\delta_k| \leq \delta$.  

306
Solve this inequality:

\[
|e_{n+1}| \leq |e_n|(1 + hL) + h\delta \\
\leq |e_{n-1}|(1 + hL)^2 + [1 + (1 + hL)]h\delta \\
\leq |e_{n-2}|(1 + hL)^3 + [1 + (1 + hL) + (1 + hL)^2]h\delta \\
\vdots \\
\leq |e_0|(1 + hL)^{n+1} + \frac{(1 + hL)^{n+1} - 1}{(1 + hL) - 1}h\delta \\
|e_n| \leq e^{nhL}\delta_0 + \frac{e^{nhL} - 1}{L}\delta
\]

using \((1 + hL)^n \leq e^{nhL}\).

Now take the limit \(h \to 0\), \(nh = T = t_f - t_0\)
\[ |e_n| \leq e^{LT} \delta_0 + \frac{e^{LT} - 1}{L} \delta \]

so change remains finite as \( h \to 0 \).

\[ \Rightarrow \text{Euler's Method is 0-stable.} \]

Note: that if \( LT \gg 1 \) the answer is very sensitive to errors \( \Rightarrow \) “ill-conditioned”

Now combine these two results to show that \( |e_n| \to 0 \) as \( h, \delta_0 \to 0 \).

**Definition:** A method is **convergent of order** \( p \) if \( \lim_{h,\delta_0 \to 0} \max |e_n| = O(h^p) \)
Theorem: A method that is consistent of order $p$ and 0-stable is convergent of order $p$.

Sketch of proof: the LTE $R/h$ play the role of the perturbation $\delta$. Therefor if the discrete condition numbers stay finite (0-stability) and the $R/h = O(h^p)$ as $h \to 0$ (consistent), then as $h, \delta_0 \to 0 \Rightarrow |e_n| = O(h^p)$ (convergent)

Example Euler’s Method

Here $R/h = \frac{1}{2}hy''(\xi) \Rightarrow \delta = \frac{1}{2}hM$ where $|y''(\xi)| \leq M$. So, for $\delta_0 \to 0$

$$|GE_n| \leq \frac{hM}{2L}[e^{LT} - 1] = Bh$$

an error bound!

⇒ Euler’s Method is convergent of order 1
1. error $\leq Bh$

   if we halve h, expect GE to halve

2. 

   \[ |GE_n| \leq \frac{hM}{2L} [e^{LT} - 1] = Bh \]  \hspace{1cm} (10)

   $\Rightarrow$ B depends sensitively on LT

   $\Rightarrow$ LT $\gg$ 0 not suitable for numerical solution

This bound v. pessimistic
can do better if \( J < 0 \) (errors are damped)

\[
| G E_k | \leq \frac{hM}{2} (t_k - t_0) = Bh
\]  

for \( h \) s.t. \( |1 + hJ| < 1 \)

BUT if \( |1 + hJ| > 1 \) and \( J < 0 \) we get error growth for a stable problem! — **numerical instability**
Region where Euler Method is stable: *interval of absolute stability*

\[ |1 + hJ| < 1 \Rightarrow -2 < hJ < 0 \]

\[ J < 0 \text{ and } h < \frac{2}{|J|} \]

\[ \Rightarrow \text{upper bound on stepsize (depending on } J\text{) to ensure numerical stability} \]
If $J \ll 0 \Rightarrow$ need very small stepsize to satisfy stability requirement

$\Rightarrow$ stepsize restricted by stability not truncation error!

An ODE where stepsize restricted by stability not truncation error because $J(t_f - t_0) \ll 0$ is called stiff.

Usually a problem is stiff if it has sharp transients.

Stiff problems require special methods!
Effect of roundoff?

In error bound for EM, replace $\frac{hM}{2}$ with $\frac{hM}{2} + \frac{\varepsilon_M}{h}$

$\Rightarrow$ an optimal stepsize, like for numerical differentiation, $h_{opt} \approx \varepsilon_M^{1/2}$

If roundoff errors add randomly: get extra error $\sim \varepsilon_M n^{1/2} \sim \varepsilon_M h^{-1/2}$

$\Rightarrow$ optimal $h$, with

$$h_{opt} \sim \varepsilon_M^{2/3}$$

Usually truncation errors dominate in solving DEs - just remember don’t set $h$ too low!
### Summary: role of $J$

<table>
<thead>
<tr>
<th>$J &gt; 0$</th>
<th>$J &lt; 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>DE unstable wrt ICs</td>
<td>DE stable wrt ICs</td>
</tr>
<tr>
<td>nearby solutions diverge</td>
<td>nearby solutions converge</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$J(t_f - t_0) \gg 1$</th>
<th>$J(t_f - t_0) \approx 1$</th>
<th>$J(t_f - t_0) \approx -1$</th>
<th>$J(t_f - t_0) \ll -1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>don’t expect</td>
<td>$h &gt; 2/</td>
<td>J</td>
<td>$</td>
</tr>
<tr>
<td>good num. sol.</td>
<td>should be OK</td>
<td>special methods</td>
<td></td>
</tr>
<tr>
<td>esp. rel. error</td>
<td>$\Rightarrow$ EM unstable</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary of Euler’s Method

<table>
<thead>
<tr>
<th>Global error</th>
<th>$\le Bh = O(h)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local error</td>
<td>$O(h^2)$</td>
</tr>
<tr>
<td>Stability interval</td>
<td>$-2 &lt; hJ &lt; 0$</td>
</tr>
<tr>
<td>Roundoff</td>
<td>$\Rightarrow h_{opt} \approx \varepsilon^{2/3}$</td>
</tr>
</tbody>
</table>

i.e. a convergent method, OK for nonstiff problems
Beyond Euler’s method

Although it’s easy to understand, Euler’s method is not accurate enough to be useful for ODEs.

**Definition:** A method is of **pth order** if local error \( \sim h^{p+1} \Rightarrow \text{global error} \sim h^p \) (provided it’s stable)

\( \Rightarrow \) Euler’s method is a **first order method**

\( \rightarrow \) we look for **higher-order methods**
There are 2 broad classes of methods: 1-step and multistep

For each class, there are:

- **explicit methods**: easier to program; relatively cheap per step
  fine for nonstiff problems

- **implicit methods**: harder to program; relatively expensive per step
  but some can solve stiff ODEs
4 kinds of higher-order methods

1. explicit 1-step methods (Runge-Kutta methods)

2. explicit multistep methods (Adams methods)

3. multistep stiff solvers (BDF methods)

4. 1-step stiff solvers (Implicit RK methods) - not in 381
End of Lecture 20
This week: aim to cover

- Runge-Kutta methods
- Variable step methods
- Multistep methods
Runge-Kutta methods

Let’s start from the quadrature rule derivation of Euler’s method and try to improve it:

\[ y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(\tau, y(\tau)) \, d\tau \]

and approximate it, not by LH rectangle rule, but by midpoint rule (open NC rule with error \( \sim h^3 \))

\[ y_{n+1} = y_n + hf(t_n + \frac{1}{2}h, y(t_n + \frac{1}{2}h)) + O(h^3) \]

but we don’t know \( y(t_n + \frac{1}{2}h) \)

→ approximate it with error \( \sim h^2 \) is good enough!
→ use EM to get \( Y_2 = y(t_n + \frac{1}{2}h) \)

\[
Y_2 = y_n + \frac{1}{2}hf(t_n, y_n)
\]

which can be done sequentially in terms of slopes \( s_i \) as

\[
s_1 = f(t_n, y_n) \tag{12}
\]

\[
s_2 = f(t_n + \frac{1}{2}h, y_n + \frac{1}{2}hs_1) \tag{13}
\]

\[
y_{n+1} = y_n + hs_2 \tag{14}
\]

**midpoint Euler or explicit midpoint method**
Try again: starting from trapezoid rule:
\[ s_1 = f(t_n, y_n) \quad (15) \]
\[ s_2 = f(t_n + h, y_n + hs_1) \quad (16) \]
\[ y_{n+1} = y_n + \frac{h}{2}(s_1 + s_2) \quad (17) \]

modified Euler or explicit trapezoid method
Both methods:

- are explicit: can compute $s_1, s_2, y_{n+1}$ in turn

- use 2 $f$ evaluations per step: 1 per stage

- try to get a better idea of slope field $f(t, y)$ by sampling at points in $[t_n, t_{n+1}]$

Are there any more? Are they really 2nd order?
Look for explicit 2-stage methods of the form:

\[ s_1 = f(t_n, y_n) \]  \hspace{1cm} (18)
\[ s_2 = f(t_n + c_2h, y_n + ha_{21}s_1) \]  \hspace{1cm} (19)
\[ y_{n+1} = y_n + h(b_1s_1 + b_2s_2) \]  \hspace{1cm} (20)

To find conditions for 2nd order, match with Taylor series for local error starting from \( y(t_n) = y_n \) (in principle):

\[ y(t_{n+1}) = y(t_n) + hy'(t_n) + \frac{1}{2}h^2y''(t_n) + O(h^3) \]
\[ y(t_{n+1}) = y_n + h f(t_n, y_n) + \frac{1}{2} h^2 \frac{d}{dt} f(t, y(t)) \big|_{t_n} + O(h^3) \]

\[ y(t_{n+1}) = y_n + h f(t_n, y_n) + \frac{1}{2} h^2 [f_t + f y'] \big|_{t_n} + O(h^3) \]

\[ y(t_{n+1}) = y_n + h f(t_n, y_n) + \frac{1}{2} h^2 [f_t + f f_y] \big|_{t_n} + O(h^3) \]

Now compare with our 2-stage method: (expand about \((t_n, y_n)\))
\[ s_1 = f_n \equiv f(t_n, y_n) \]

\[ s_2 = f_n + f_{tc} c_2 h + f_y h a_{21} s_1 + O(h^2) \]

\[ y_{n+1} = y_n + h b_1 (f_n) + h b_2 (f_n + f_{tc} c_2 h + f_y h a_{21} s_1 + O(h^2)) \]

\[ = y_n + h(b_1 + b_2) f_n + h^2 (b_2 c_2) f_t + h^2 b_2 a_{21} f_y f_n + O(h^3) \]

For this to match Taylor series to \( O(h^3) \), need:
order conditions

\begin{align*}
  b_1 + b_2 &= 1 \\
  b_2 c_2 &= \frac{1}{2} \\
  b_2 a_{21} &= \frac{1}{2}
\end{align*}

\Rightarrow a_{21} = c_2

3 equations in 4 unknowns \Rightarrow a 1-parameter family of methods:
\[ s_1 = f(t_n, y_n) \]  \hspace{1cm} (24) \\
\[ s_2 = f(t_n + \alpha h, y_n + \alpha h s_1) \]  \hspace{1cm} (25) \\
\[ y_{n+1} = y_n + h((1 - \frac{1}{2\alpha})s_1 + \frac{1}{2\alpha}s_2) \]  \hspace{1cm} (26) 

This derivation proves it’s at least 2nd order; in fact, it’s 2nd order (disagrees with Taylor series at next term)

Any such method is: second order (explicit) Runge-Kutta \textbf{RK2}

\textbullet \hspace{1cm} \textit{Example}

\[ \alpha = 1/3 \rightarrow \text{RK2 with lowest local error (Heun)} \]
We specify this in a **RK tableaux**:

\[
\begin{array}{c|ccc}
  \text{c} & 0 & 0 & 0 \\
  \text{b} & 1 & 1 & 0 \\
  \hline
 1/2 & 1/2 \\
\end{array}
\]

where \(c\) are the **RK nodes**, \(b\) are the **RK weights**, \(A\) is the **RK matrix**.

For explicit methods, \(A\) is nonzero only below diagonal \(\Rightarrow\) can evaluate \(s_1, s_2 \cdots\) in turn.

\(<\text{ Example explicit trapezoid }>

\[
\begin{array}{c|ccc}
  \text{c} & 0 & 0 & 0 \\
  \text{b} & 1 & 1 & 0 \\
  \hline
 1/2 & 1/2 \\
\end{array}
\]
Is higher order worthwhile?

Set tolerance $\tau$, suppose $t_f - t_0 = 1$

For Euler: $GE \sim h \Rightarrow h \sim \tau$

number of func. evals $= 1$ per step $\times$ $n$ steps $= \frac{1}{h} \sim \tau^{-1}$

For RK2: $GE \sim h^2 \Rightarrow h \sim \tau^{1/2}$

number of func. evals $= 2$ per step $\times$ $n$ steps $= \frac{1}{h} \sim 2\tau^{-1/2}$
RK2 wins if $2\tau^{-1/2} < \tau^{-1} \Rightarrow \tau < 1/4$

Moral: For small enough tolerances, a higher order method is worthwhile

What about stability of RK2?
Recall **0-stability**: for fixed $t$ as $h \to 0$, want discrete problem to have bounded errors.

**All RK methods are 0-stable**

hence, from Big Theorem:

\[
\text{Consistency + Stability} \implies \text{Convergence}
\]

\[\rightarrow \text{All RK methods are convergent}\]
While comforting, we run ODE codes at finite $h$. We want numerical solution to have damped errors, when true solutions are contractive $J < 0$.

We stick to **Linear stability** ⇒ consider autonomous case

$$y_1' = Ay_1 + b(t); \quad y_1(0) = y_0$$

Then nearby solution with different IC satisfies

$$y_2' = Ay_2 + b(t); \quad y_2(0) = y_0 + \delta$$

The difference $z$ satisfies

$$z' = Az; \quad z(0) = \delta$$
Assume $A$ is diagonalizable: then $A = SΛS^{-1}$; $\lambda_i \in \mathbb{C}$. So by changing variables

$$w = S^{-1}z$$

we get the system

$$w' = Λw$$

which is decoupled

$$w'_i = \lambda_i w_i; \lambda_i \in \mathbb{C}$$

since $Λ$ is diagonal.

This explains why, for linear stability, we use the model scalar equation

$$y' = λy, \quad λ \in \mathbb{C}$$
For contractive solutions, need DE to be stable $\Rightarrow Re(\lambda) < 0$

for numerical solution to behave qualitatively the same $\rightarrow$ we demand

$$|y_{n+1}| < |y_n|$$

**A-stability** — also ensures errors $w_i$ don't grow
For Euler’s Method:

\[ y_{n+1} = y_n + hf(t_n, y_n) = y_n + h\lambda y_n = (1 + h\lambda)y_n \]

We call \( \{ h\lambda \in \mathbb{C} : |1 + h\lambda| < 1 \} \) the region of absolute stability.

Its intersection with real axis = \((-2, 0)\) is the interval of absolute stability.
Do same for RK2: apply method to the model equation

\[
\begin{align*}
    s_1 &= f(t_n, y_n) = \lambda y_n \\ 
    s_2 &= \lambda(y_n + \alpha hs_1) = \lambda(y_n + \alpha h\lambda y_n) \\ 
    y_{n+1} &= y_n + h((1 - \frac{1}{2\alpha})s_1 + \frac{1}{2\alpha}s_2) \\ 
    &= y_n + h((1 - \frac{1}{2\alpha})\lambda y_n + \frac{1}{2\alpha}\lambda(y_n + \alpha h\lambda y_n)) \\ 
    &= y_n[1 + h\lambda + \frac{1}{2}(h\lambda)^2]
\end{align*}
\]  

→ Region of absolute stability: \(| 1 + h\lambda + \frac{1}{2}(h\lambda)^2 | < 1\)

→ Interval of absolute stability: \((-2, 0)\) (again)

**Moral:** Use RK2 for better accuracy, not improved stability
Note: exact solution

\[ y(t_{n+1}) = e^{\lambda h}y_n \]

\[ = [1 + h\lambda + \frac{1}{2}(h\lambda)^2 + O(h^3)]y_n \]

→ RK2 is only 2nd order, not higher
**A-stability**

for large $t$ at fixed $h$, want numerical solution to behave $\sim$ exact solution $\Rightarrow$ want errors to dampen

If region of A-stability includes $Re(\lambda) < 0$ (whole LH of complex plane)

$\rightarrow$ method is A-stable

**No explicit RK method is A-stable**

since region of absolute stability given by $|P(h\lambda)| < 1$ where $P$ is some polynomial, which must $\rightarrow \infty$ as $\lambda \rightarrow -\infty$
Similarly look for 3rd order methods using 3 stages:

\[
\begin{align*}
  s_1 &= f(t_n, y_n) \tag{32} \\
  s_2 &= f(t_n + c_2 h, y_n + h a_{21} s_1) \tag{33} \\
  s_3 &= f(t_n + c_3 h, y_n + h a_{31} s_1 + h a_{32} s_2) \tag{34} \\
  y_{n+1} &= y_n + h (b_1 s_1 + b_2 s_2 + b_3 s_3) \tag{35}
\end{align*}
\]

match with Taylor series → 3 1-parameter families, all RK3 with 3 stages
Similarly look for 4th order methods using 4 stages:

\[ \begin{array}{c|cccc}
0 & & & & \\
1/2 & 1/2 & & & \\
1/2 & 0 & 1/2 & & \\
1 & 0 & 0 & 1 & \\
\hline
1/6 & 1/3 & 1/3 & 1/6 & \\
\end{array} \]

Can go on, but for \( p > 4 \) need \( s > p \)

\[ \begin{array}{c|cccc}
\text{Example for RK4} & \text{classical} & RK4 \\
\end{array} \]

\[ \begin{array}{c|cccc}
\text{Example for RK5, need 6 stages} & & & & \\
\end{array} \]
Effect of roundoff

For each method, GE $\sim h^p$ after $n \sim 1/h$ steps: truncation error

If roundoff errors add randomly: get extra error $\sim \varepsilon_M n^{1/2} \sim \varepsilon_M h^{-1/2}$

$\rightarrow$ optimal $h$ just like numerical differentiation, with

$$h_{\text{opt}} \sim \varepsilon_M^{2/(2p+1)}$$

$\rightarrow h_{\text{opt}} \sim 10^{-5}, 10^{-4}, 10^{-3}$ for RK3, RK4, RK5 in double precision
The diagram illustrates the effect of roundoff for RK3, showing the global error at $t=1$ as a function of $-\log_{10}(h)$, where $h$ is likely a step size or a parameter related to the discretization of the problem. The $y$-axis represents the global error, which decreases as $-\log_{10}(h)$ increases, indicating that the error decreases as the step size $h$ decreases. This is a common trend in numerical methods where smaller step sizes generally lead to more accurate solutions at the expense of increased computational cost.
effect of roundoff for RK4

global error at t=1

-log10(h)
Effect of roundoff for RK5

Global error at t=1

-log10(h)

-16
-15
-14
-13
-12
-11
-10
-9
-8

-log10(h)
## Summary of RK methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Order</th>
<th>Local error</th>
<th>Fevals</th>
<th>$h_{opt}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Euler = RK1</td>
<td>1</td>
<td>$h^2$</td>
<td>1</td>
<td>$\sim 10^{-10}$</td>
</tr>
<tr>
<td>RK2</td>
<td>2</td>
<td>$h^3$</td>
<td>2</td>
<td>$\sim 5 \times 10^{-7}$</td>
</tr>
<tr>
<td>RK3</td>
<td>3</td>
<td>$h^4$</td>
<td>3</td>
<td>$\sim 10^{-5}$</td>
</tr>
<tr>
<td>RK4</td>
<td>4</td>
<td>$h^5$</td>
<td>4</td>
<td>$\sim 10^{-4}$</td>
</tr>
<tr>
<td>RK5</td>
<td>5</td>
<td>$h^6$</td>
<td>6</td>
<td>$\sim 10^{-3}$</td>
</tr>
</tbody>
</table>
End of Lecture 21
So far, everything has been fixed-step.

Armed with 2 methods of different order $\rightarrow$ estimate of local error
$\rightarrow$ control stepsize to achieve a tolerance $\tau$

$\rightarrow$ variable-step methods

Want 2 estimates to share Fevals (cf. adaptive quadrature) $\Rightarrow$ design methods which share stages

2 RK methods of different order with same $c, A$ (same evaluation points) $\rightarrow$
Embedded Runge-Kutta methods

Example Matlab’s ode23, ode45

Basic idea: control size of local error by using 2 estimates from methods of different order $p, p + 1$

$y_{n+1}^p$ has local error $\sim Ch^{p+1}$

$y_{n+1}^{p+1}$ has local error $\sim \bar{C}h^{p+2}$

for usual values of $h$, $\bar{C}h^{p+2} \ll Ch^{p+1}$ so we estimate error in $y_{n+1}^p$ (worse method) by

$$\text{err} = |y_{n+1}^p - y_{n+1}^{p+1}|$$

and demand that $\text{err} < \tau$
If $\text{err} < \tau$, keep that step, using $y_{n+1}^{p+1}$ (better estimate)

else

cut down stepsize $h$ so $\text{err} < \tau$ with new stepsize

end

Control stepsize using error estimate of worse method but keep better estimate

Hope this makes up for controlling local error, not global error.

How to rescale $h$?
We want \( \text{err} < \tau \) and we know \( \text{err} \sim h^{p+1} \)

\[ \Rightarrow \text{we will achieve tolerance with new stepsize} = qh_{old} \]

\[
\frac{\text{err}_{new}}{\text{err}_{old}} \sim \frac{C(qh)^{p+1}}{Ch^{p+1}} = q^{p+1} = \frac{\tau}{\text{err}}
\]

so we choose

\[ q = 0.8 \left( \frac{\tau}{\text{err}} \right)^{1/(p+1)} \]

where 0.8 is a safety factor.
Example ode23tx

uses 3rd order 3-stage RK3

and (4-stage!) RK2 which uses $s_1, s_2, s_3$ from RK3 (no extra work)

and $s_4 = f(t_{n+1}, y_{n+1})$

Note: $s_4 \rightarrow s_1$ on next step (FSAL) so this costs nothing extra if step is accepted (i.e. most of the time) → a 3rd order method + error estimator for $\sim$ 3 stages of work!

In fact, don’t bother forming $y^p$ at all — just form estimator $|y_3 - y_2|$
Example ode45

uses a 5th order 6-stage RK

+ 4th order 7-stage FSAL RK

→ 5th order method + error estimator for ~ 6 stages of work!

Embedded RK methods are good nonstiff 1-step solvers — prob. first methods to try.
Other major class of nonstiff solvers use previous values of \( y_k \)

**Multistep methods** esp. **Adams methods**

Go back to quadrature formula:

\[
y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(\tau, y(\tau)) \, d\tau
\]

**Basic idea:** use previous \( f \) values \( f_k \equiv f(t_k, y_k), k = n, n - 1, \ldots \) to construct an interpolant

integrate interpolant (extrapolation) to get \( y_{n+1} \)

**Adams-Bashforth methods** — a family of **explicit** multistep methods
Example Use $f_n$ only $\rightarrow$ Euler $= AB1$

Use $f_n, f_{n-1}$ $\rightarrow$ linear interpolant $\rightarrow$ 2nd order 2-step method AB2

\[ y_{n+1} = y_n + h \left( \frac{3}{2} f_n - \frac{1}{2} f_{n-1} \right) \]

Note: we need 2 starting values $y_0, y_1$ to start this

Multistep methods are not self-starting

Same idea $\rightarrow$ AB3, AB4 etc.

Why bother?
AB methods need only 1 Feval per step

⇒ very efficient cf. RK methods!

BUT check stability using model equation $y' = \lambda y$ again

$$y_{n+1} = y_n + h\lambda \left[ \frac{3}{2}y_n - \frac{1}{2}y_{n-1} \right]$$

a 2nd order recurrence relation with 2 lin. indep. solutions

$$y_k = c_1 r_1^k + c_2 r_2^k$$

where $r_1, r_2$ roots of

$$r^2 - (1 + \frac{3}{2}h\lambda)r + \frac{1}{2}h\lambda = 0$$
→ for A-stability, we need

\[ |r_1(h\lambda)| < 1 \text{ and } |r_2(h\lambda)| < 1 \]

⇒ stability analysis more complicated for multistep methods

Exact solution \( = e^{\lambda t} = (e^{\lambda h})^k \approx (1 + \lambda h + ..)^k \)

Larger root \( r_1(h\lambda) = 1 + h\lambda + \cdots \) (the principal root) approximates the true solution.

Smaller root \( r_2(h\lambda) = h\lambda + \cdots \) is a parasitic root that can only cause problems.
For a k-step method, get 1 principal root and \( k - 1 \) parasitic roots.

**Theorem:** For a multistep method to be 0-stable, all the parasitic roots must satisfy \( |r_j| < 1 \) as \( h \to 0 \).

For Adams methods, all the parasitic roots \( r_j \to 0 \) as \( h \to 0 \)

\( \Rightarrow \) **All Adams methods are 0-stable**

\( \Rightarrow \) **All Adams methods are convergent**
For Adams-Bashforth methods, regions of absolute stability shrink rapidly with increasing order.

what to do?

Look at **implicit methods**
Back to derivation but now include $f_{n+1}$ in interpolant

$\rightarrow$ a family of implicit multistep methods — **Adams-Moulton methods**

$\rhd$ Example 0th degree interpolant thru $f_{n+1}$: use RH rectangle rule

$y_{n+1} = y_n + hf_{n+1} = y_n + hf(t_{n+1}, y_{n+1})$

**Backward Euler method** = AM0 (1st order implicit method)
After this, get k-step methods with order $k + 1$

Example put linear interpolant thru $f_n, f_{n+1}$, integrate over $[t_n, t_{n+1}]$ (trapezoid rule)

→ implicit trapezoid rule = AM1

$$y_{n+1} = y_n + \frac{h}{2}[f_n + f_{n+1}] = y_n + \frac{h}{2}[f(t_n, y_n) + f(t_{n+1}, y_{n+1})]$$

Error of trapezoid rule $\sim h^3 \rightarrow$ 2nd order method
Example: put quadratic interpolant thru $f_{n+1} f_n, f_{n-1}$, integrate over $[t_n, t_{n+1}] \rightarrow \text{AM2}$

$$y_{n+1} = y_n + \frac{h}{12}[-f_{n-1} + 8f_n + 5f_{n+1}]$$

a 2-step 3rd order method

and so on

**The good news**: regions of absolute stability are much bigger than for AB methods

**The bad news**: need to solve nonlinear equations to convergence to get these stability regions
In practice, don’t solve AM methods to convergence

instead just do 1 iteration of the fixed point iteration

\[ y_{n+1}^{(k+1)} = y_n + \frac{h}{2}[f(t_n, y_n) + f(t_{n+1}, y_{n+1}^{(k)})] \]

but need initial guess!

use corresponding AB method (explicit) of same order to get initial guess \( y_{n+1}^{(0)} \)

→ **Predictor-Corrector pair**
Use AB (explicit) to get initial guess $\hat{y}_{n+1}$ — predictor

then use AM method (1 iteration) to get $y_{n+1}$ — corrector

Example AB2-AM1 (both 2nd order)

$\hat{y}_{n+1} = y_n + h\left[\frac{3}{2}f_n - \frac{1}{2}f_{n-1}\right] \text{ Predict}$

$\hat{f}_{n+1} = f(t_{n+1}, \hat{y}_{n+1}) \text{ Evaluate}$

$y_{n+1} = y_n + \frac{h}{2}[f_n + \hat{f}_{n+1}] \text{ Correct}$

$f_{n+1} = f(t_{n+1}, y_{n+1}) \text{ Evaluate (ready for next step)}$

PECE
Overall → an explicit method with 2 Fevals per step but much bigger stability regions (than AB)

In practice, Adams codes are variable-step (messy) and variable order — not for amateurs

Example Mathematica's NDSolve (500,000 lines of C)

Matlab’s ode113 — Adams methods of orders 1–13 (700 lines of Matlab)
## Summary of nonstiff solvers

<table>
<thead>
<tr>
<th>1-step (RK)</th>
<th>Multistep (Adams)</th>
</tr>
</thead>
<tbody>
<tr>
<td>explicit</td>
<td>PECE</td>
</tr>
<tr>
<td>expensive for high order $p$</td>
<td>easy to get high order</td>
</tr>
<tr>
<td>$\geq p$ Fevals per step</td>
<td>2 Fevals per step</td>
</tr>
<tr>
<td>stability region grows with order</td>
<td>stability region shrinks with order</td>
</tr>
<tr>
<td>0-stable $\Rightarrow$ convergent</td>
<td>0-stable $\Rightarrow$ convergent</td>
</tr>
<tr>
<td>variable-step $\rightarrow$ simple code</td>
<td>variable step/order $\rightarrow$ complex code</td>
</tr>
<tr>
<td>self-starting</td>
<td>start with low order or RK</td>
</tr>
<tr>
<td>general purpose solver</td>
<td>best if problem very smooth or tight tolerances</td>
</tr>
<tr>
<td></td>
<td>or $f$ very expensive</td>
</tr>
</tbody>
</table>
End of Lecture 22
This week: aim to cover

- Stiff solvers
- Revision/exam
**Stiff solvers**

Most of the methods we have seen so far have had finite regions of absolute stability

⇒ if $\Re(hJ) \ll 0$ stepsize will need to be tiny for accurate numerical solution

⇒ $h$ determined by stability, not accuracy

What happens if you try a nonstiff variable-step solver (e.g. ode23tx) on a stiff problem?
If $h$ is outside stability region, 2 estimates will vary a lot (due to error growth) → solver thinks $h$ is too big (for accuracy reasons) so cuts $h$ down

⇒ solver keeps going happily, just with very small steps

**Moral: a nonstiff solver will solve a stiff problem — just takes a long time to do it!**
Ideally, we would like a method to be stable $\forall \Re(h\lambda) < 0$ i.e. $\textbf{A-stable}$.

Example

Backward Euler (Implicit Trapezoid) methods are A-stable, but only 1st (2nd) order resp.
Can we do better? NO

**Theorem**: (Dahlquist) No A-stable multistep method has order $> 2$

Is A-stability really what we want? Problems with sharp transients typically have $\lambda$ near the negative real axis, not near the imaginary axis (**oscillatory ODEs**) $\Rightarrow$ A-stability is too strong

$\Rightarrow$ **weaken A-stability** to require

$$|y_{n+1}| < |y_n|$$

for $\lambda h$ in a wedge about negative real axis i.e. doesn’t include region near imaginary axis $\rightarrow$ **$A(\alpha)$-stability**
OR What if $\lambda \to -\infty$?

We want numerical solution to be strongly damped in this case i.e. $\frac{|y_{n+1}|}{|y_n|} < 1$ is too weak

$\Rightarrow$ strengthen A-stability to require

$\frac{|y_{n+1}|}{|y_n|} \to 0$ (not just $< 1$) as $\lambda \to -\infty$

$\Rightarrow$ method is L-stable

$\triangle$ Example
The first common class of methods that satisfy these definitions are BDF methods (Gear 1971), which we get by approximating $y'$ not $\int_{t_n}^{t_{n+1}} f \, d\tau$.

Start from

$$y'(t_{n+1}) = f(t_{n+1}, y_{n+1})$$

$\Rightarrow$ we’ll have only 1 $f$ i.e. $f_{n+1} \rightarrow$ an implicit method

Now approx, LHS by deriv. of polynomial interpolant thru $t_{n+1}, t_n, t_{n-1} \cdots$

$\triangleleft$ Example Use $t_{n+1}, t_n \rightarrow$ Backward Euler $= \text{BDF1}$
Use $t_{n+1}, t_n, t_{n-1}$→

$$y_{n+1} - \frac{4}{3}y_n + \frac{1}{3}y_{n-1} = \frac{2}{3}hf_{n+1}$$

= BDF2 — a 2nd order A-stable method

Sim. → BDF3, BDF4 with stability regions showing $A(\alpha)$ stability
BDF methods have traded in A-stability near imaginary axis for L-stability → first general purpose stiff solvers

Note: BDF methods not 0-stable for $k > 6$ ⇒ only use BDF1–5.

→ variable-step variable-order codes based on BDF

Example Mathematica’s NDSolve switches between Adams methods and BDF methods when it thinks DE has become stiff.

Matlab’s ode15s uses similar methods and has an option to use BDF
These stability regions only hold if we solve nonlinear equations to convergence – how to do this?

Example use Backward Euler to illustrate

Fixed point iteration?

\[ y_{n+1} = y_n + hf(t_{n+1}, y_{n+1}) \equiv g(y_{n+1}) \]

Fixed point iteration only converges if \(|g'(x^*)| < 1\) (a contraction mapping)
In our case \( g' = h \frac{\partial f}{\partial y} = hJ \) so the restriction \( |g'(x^*)| < 1 \) ⇒
\[
|hJ| < 1
\]
which then puts the same restrictions on \( h \) that we just tried to lift

**Moral:** cannot use fixed-point iteration to solve implicit equations for stiff problems

→ try Newton’s method
Write Backward Euler as

\[ F(y_{n+1}) = y_{n+1} - y_n - hf(t_{n+1}, y_{n+1}) = 0 \]

Newton’s Method is

\[ x^{k+1} - x^k \equiv \Delta x^k = -\frac{F(x^k)}{F'(x^k)} \]

\[ \Delta y_{n+1}^k = -\frac{y_{n+1}^k - y_n - hf(t_{n+1}, y_{n+1}^k)}{1 - hJ(y_{n+1}^k)}; \quad y_{n+1}^{k+1} = y_{n+1}^k + \Delta y_{n+1}^k \]

This usually takes only a few iterations, since \( y_n \) is a good initial guess.
For a system of ODEs this generalizes to

$$(I - hJ)_k \Delta y_{n+1}^k = -(y_{n+1}^k - y_n - hf(t_{n+1}, y_{n+1}^k))$$

and the biggest expense is to form the Jacobian matrix and solve a new system at every iteration

⇒ instead approximate $(I - hJ)_k$ to cut down expense

△ Example use $(I - hJ)_0$, LU factorize this once ⇒ each iteration needs forward/back substitution only

→ modified Newton iteration ~ chord method for systems

lose quadratic convergence but still converges (linearly) quite fast
Is that all to ODE solvers? NO

**Implicit RK solvers:**

- are all convergent

- can be A-stable and high-order

- can be L-stable and high-order

- can be proved stable for nonlinear nonautonomous problems

**BUT take a lot more work! :-((**
End of Subject!