Py4Sci Notes 2021

Created by Anton Leagre

Some remarks:

- These notes will be in English.
- As these are mainly written to prepare for the oral theory exam, no notes will contain long-winded examples or code snippets. Proofs will be included where necessary.
- At the end of each chapter there's a summary of the different algorithms and their advantages/ disadvantages
- Copyright disclaimer: If you got these notes by paying, you got scammed, as sharing notes for profit is illegal as per the UGent Education and Examination code.

1 Numerical Limitations

1.1 Basic concepts

Absolute Error = $\text{approximate value} - \text{true value}$

Relative Error = $\frac{\text{absolute error}}{\text{true value}}$

relative error of about $10^{-p} \iff p$ correct **significant digits** (the leading nonzero digit and all following digits).

Precision: the number of digits with which a number is expressed.

Accuracy: the number of *correct* significant digits in an approximation of the desired quantity.

Truncation error: The difference between the true result and the result given by an algorithm using exact arithmetic. It is due to approximations such as truncating an infinite series or replacing derivatives by finite differences...

Error in the algorithm ITSELF.

Rounding error: The difference between the result produced by a given algorithm using exact arithmetic and the same algorithm, using finite-precision, rounded arithmetic.

Error due to FLOATING POINT ROUNDING.

1.2 Floating-point number systems

$\pm \left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_{p-1}}{\beta^{p-1}} \right) \beta^E$
where d_i and E are integers such that $0 \leq d_i \leq \beta - 1$ and $L \leq E \leq U$.

symbol	name
β	Base or radix
p	Precision
$[L, U]$	Exponent range

A floating-point system is **normalized** if the leading digit d_0 always equals 1 (unless the number represented is zero).

This is advantageous because

- Each number has a unique representation.
- No digits are wasted on leading zeros, thereby maximizing precision.
- In a binary system ($\beta = 2$), the leading bit is always 1, and thus need not be stored, thereby gaining an additional bit of precision.

1.3 Properties

A floating-point number is finite and discrete. The number of normalized floating-point numbers in a given system is
$2(\beta-1)\beta^{p-1}(U-L+1)+1$

- 2 choices of sign
- $(\beta-1)$ choices for the leading digit of the mantissa (can't be zero, otherwise leading zero)
- β^{p-1} because there are β choices for the remaining $p-1$ digits of the mantissa
- $(U-L+1)$ possible values for the exponent
- $+1$ because the numbr could be zero

The smallest positive normalized number (the **underflow level**) equals β^L

The largest number (the **overflow level**) equals $\beta^{U+1}(1-\beta^{-p})$

Floating-point numbers are not uniformly distributed throughout their range, but are equally spaced only between successive powers of β.

1.4 Good practices

Cancellation

- Avoid subtracting two almost identical numbers

Addition

- Avoid adding small and large numbers
- Perform a sequence of additions ordered from the smallest number to the largest

1.5 Summary

- Basic concepts

Absolute error, relative error, precision, accuracy, truncation error, rounding error

- Floating point numbers
  The system is representable by
  - a base or radix (often 2)
  - a precision
  - a exponent range
    A number in a system than has
  - a mantissa, sequence of numbers
  - an exponent in the exponent range
  - sometimes, extra sign bit

- Good practices
    - cancellation
        - avoid substracting two almost identical numbers.
    - addition
        - avoid adding small and large numbers
        - perform a sequence of additions ordered from the smallest number to the largest

2 Linear Systems

2.1 Introduction

2.2 Solving Linear systems

2.2.1 Strategy

A lower triangular system $\mathbf{Lx} = \mathbf{b}$ can be solved by **forward substitution**, mathematically expressed as

$$x_1 = b_1/l_{11}, \quad x_i = \left(b_i - \sum_{j=1}^{i-1} l_{ij}x_j\right)/l_{ii} \quad \text{for} \quad i = 2, \ldots, n$$

An upper triangular system $\mathbf{Ux} = \mathbf{b}$ can be solved by **backward substitution**, mathematically expressed as

$$x_n = b_n/u_{nn}, \quad x_i = \left(b_i - \sum_{j=1+i}^{n} u_{ij}x_j\right)/u_{ii} \quad \text{for} \quad i = n-1, \ldots, 1$$

We need non-singular linear transformations that make an arbitrary matrix triangular.

2.2.2 Gauss Transformation

Gauss transformations make all entries of a vector below an index a_k zero:

$$\mathbf{M_{ka}} = \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & -m_{k+1} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -m_n & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_k \\ a_{k+1} \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} a_1 \\ \vdots \\ a_k \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

with $m_i = a_i/a_k \quad i = k+1, \cdots, n$

We also have that

- $\mathbf{M}_k = \mathbf{I} - \mathbf{m}_k\mathbf{e}_k^T$, where $\mathbf{m}_k = [0, \cdots, 0, m_{k+1}, \cdots, m_n]$ and $\mathbf{e}_k$ is the kth column of the identity matrix
- $\mathbf{M}_k^{-1} = \mathbf{I} + \mathbf{m}_k\mathbf{e}_k^T$, which means that $\mathbf{M}_k^{-1}$, denoted as $\mathbf{L}_k$, is the same as $\mathbf{M}_k$, except that the signs of the mulipliers are reversed.

2.2.3 LU Factorization

We make matrix A upper triangular by multiplying Gauss tranformations to A at each row.

$$\mathbf{MA} = \mathbf{M_n M_{n-1}} \dots \mathbf{M_1 A} = \mathbf{U}$$

We can factor A as follows:

$$\mathbf{A} = \mathbf{M^{-1}MA} = \mathbf{LU}$$

with $\mathbf{L} = \mathbf{M^{-1}}$ lower triangular, because $\mathbf{M}$, as a product of lower triangular matrices, is still lower triangular, and $\mathbf{L} = \mathbf{M^{-1}}$ the inverse of of lower triangular matrix is again still lower triangular. In fact the explicit formula for $\mathbf{L}$ is just

$$L = \begin{pmatrix} 1 & & & & & \\ \ell_{2,1} & \ddots & & & & \\ & \ddots & 1 & & & \\ \vdots & \cdots & \ell_{n+1,n} & 1 & & \\ & & \vdots & \ddots & \ddots & \\ \ell_{N,1} & \cdots & \ell_{N,n} & \cdots & \ell_{N,N-1} & 1 \end{pmatrix}$$

in which the only essential difference is the minus signs and $l = 1/m$.

2.2.4 Partial pivoting

Problems with Gauss elimination:

1. The process breaks down if the leading diagonal entry of the matrix is zero at any stage.

- This issue is trivial to solve: switch rows with a row that has a non-zero element: **pivoting**.

2. In finite-precision arithmetic we wish to limit the size of the multipliers so that previous rounding errors do not get amplified.

- The multipliers will never exceed 1 in magnitude if for each column we choose the entry of the largest magnitude on or below the diagonal as a pivot. Such a policy is called **partial pivoting**

2.2.5 Gauss-Jordan Elimination

A similar transformation that makes the entries below AND above zero; reducing A to a diagonal form.

Typically, it is not used because

- The final solution phase is computationally somewhat cheaper because of the diagonal form of the matrix, but this does not suffice make up for the additional cost in the elimination phase.

However, it might be desirable in some situations:

- In an implementation on parallel computers, the workload remains the same throughout the elimination phase and the final solutions can all be calculated at once.
- It can also be used to calculate the inverse of a matrix explicitely by initializing the right-hand side of the matrix as the identity matrix $\mathbf{I}$.

2.3 Special types of Linear Systems

Some systems allow faster calculations

- **Symmetric**: $\mathbf{A} = \mathbf{A}^T$, i.e. $a_{ij} = a_{ji}$ for all i, j
- **Positive definite**: $\mathbf{x^{TAx}} > 0$ for all $\mathbf{x} \neq 0$
- **Banded**: $a_{ij} = 0$ for all $\|i - j\| > \beta$, with β the **bandwidth** of $\mathbf{A}$
- **Sparse**: most entries of $\mathbf{A}$ are zero

2.3.1 Cholesky Factorization

If a matrix $\mathbf{A}$ is symmetric and positive definite, then an LU factorization can be arranged so that $\mathbf{U} = \mathbf{L}^T$, meaning that $\mathbf{A} = \mathbf{L}\mathbf{L}^T$, where $\mathbf{L}$ is lower triangular.

We can get the elements of $\mathbf{L} = \mathbf{U}^T$ by comparing $\mathbf{A}$ and $\mathbf{L}\mathbf{L}^T$

In 2D

- $l_{11} = \sqrt{a_{11}}$
- $l_{21} = a_{21}/l_{11}$
- $l_{22} = \sqrt{a_{22} - l_{21}^2}$

The diagonal entries are always positive, thus the roots are well defined.

The Cholesky factorization has a few very attractive properties:

- The n square roots are all of positive numbers, so the algorithm is well-defined
- Pivoting is not required
- Only the lower triangle of $\mathbf{A}$ is accessed, and hence the strict upper triangular portion need not be stored
- Only about $n^3/6$ multiplications and a similar number of additions are required.

Thus Cholesky factorization requires only about half as much storage and work as general LU-factorization.

2.3.2 Computational Complexity

As shown in the examples below:

- LU factorization of an $n \times n$ matrix takes<about $n^3/3$ floating point operations (flops), THIS IS NOT CORRECT AFAIK $n^3/3$ Multiplications and additions, total $2n^3/3$

- A complete matrix inversion takes about n^3 flops and thus is more expensive
- Solving an LU-factorized system using forward and backward substitution takes about n^2 flops. For large systems, this is neglegible compared to the factorization phase.
- Cramer's rule (in which the system is solved using ratios of determinants) is astronomically expensive

2.4 Sensitivity and Conditioning

2.4.1 Vector norms

p-norms

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} \|x_i\|^p \right)^{1/p}$$

Important cases:

- **1-norm** or **Manhattan norm**

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} \|x_i\|$$

- **2-norm** or **Euclidean norm**

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} \|x_i\|^2}$$

- ∞**-norm** (the limit for $p \to \infty$):

$$\|\mathbf{x}\|_\infty = \max_{1 \le i \le n} \|x_i\|$$

2.4.2 Matrix norms

The norm of an $m \times n$ matrix $\mathbf{A}$ is given by

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

Two cases that are easy to calculate are:

- $\|\mathbf{A}\|_1$, which corresponds the maximum absolute *column* sum of the matrix:

$$\|\mathbf{A}\|_1 = \max_j \sum_{i=1}^{m} \|a_{ij}\|$$

- $\|\mathbf{A}\|_\infty$, which corresponds the maximum absolute *row* sum of the matrix:

$$\|\mathbf{A}\|_\infty = \max_i \sum_{j=1}^{n} \|a_{ij}\|$$

Properties

- $\|\mathbf{A}\| > 0$ if $\mathbf{A} \neq \mathbf{0}$
- $\|\gamma\mathbf{A}\| = \|\gamma\| \cdot \|\mathbf{A}\|$, for any scalar γ
- $\|\mathbf{A} + \mathbf{B}\| \leq \|\mathbf{A}\| + \|\mathbf{B}\|$
- $\|\mathbf{AB}\| \leq \|\mathbf{A}\| \cdot \|\mathbf{B}\|$
- $\|\mathbf{Ax}\| \leq \|\mathbf{A}\| \cdot \|\mathbf{x}\|$, for any vector $\mathbf{x}$

2.4.3 Condition number

Measure of how close a matrix is to being singular.

$$\text{cond}(\mathbf{A}) = \|\mathbf{A}\| \cdot \|\mathbf{A}^{-1}\|$$

Minimum value is 1 (for identity matrix). Is ∞ (by definition) for singular matrices.

A condition number close to 1 corresponds to a well-posed problem, whereas a very large condition number tells you that a solution of a linear system will change drastically for small changes in the input data.
We quantify this in the next paragraph

2.4.4 Error Estimation

let $\mathbf{x}'$ be the solution to the perturbed system $\mathbf{Ax}' = \mathbf{b} + \Delta\mathbf{b}$, and define the difference between both solutions $\mathbf{x}' - \mathbf{x}$ as $\Delta\mathbf{x}$.

This results in

$$\mathbf{Ax}' = \mathbf{A}(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{Ax} + \mathbf{A}\Delta\mathbf{x} = \mathbf{b} + \Delta\mathbf{b}$$

Consequently, $\mathbf{A}\Delta\mathbf{x} = \Delta\mathbf{b}$, and hence $\Delta\mathbf{x} = \mathbf{A}^{-1}\Delta\mathbf{b}$.

Taking norms, and using the properties listed above we find:

- $\|\mathbf{b}\| = \|\mathbf{Ax}\| \leq \|\mathbf{A}\| \cdot \|\mathbf{x}\|$ or $\|\mathbf{x}\| \geq \|\mathbf{b}\|/\|\mathbf{A}\|$
- $\|\Delta\mathbf{x}\| = \|\mathbf{A}^{-1}\Delta\mathbf{b}\| \leq \|\mathbf{A}^{-1}\| \cdot \|\Delta\mathbf{b}\|$

Combining both equalities gives

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \text{cond}(\mathbf{A})\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|}$$

A similar reasoning (which is left as an **exercise**) learns us that for deviations $\mathbf{E}$ in the matrix $\mathbf{A}$, such that $(\mathbf{A} + \mathbf{E})\mathbf{x}' = \mathbf{b}$, we find

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}'\|} \leq \text{cond}(\mathbf{A})\frac{\|\mathbf{E}\|}{\|\mathbf{A}\|}$$

2.4.5 Residual

The **residual** $\mathbf{r}$ of an approximate solution $\mathbf{x}'$ of the system $\mathbf{Ax} = \mathbf{b}$ is defined as

$$\mathbf{r} = \mathbf{b} - \mathbf{Ax}'$$

Or relative residual:

$$\frac{\|\mathbf{r}\|}{(\|\mathbf{A}\| \cdot \|\mathbf{x}'\|)}$$

$$\|\Delta\mathbf{x}\| = \|\mathbf{x}' - \mathbf{x}\| = \|\mathbf{A}^{-1}(\mathbf{Ax}' - \mathbf{b})\| = \| - \mathbf{A}^{-1}\mathbf{r}\| \leq \|\mathbf{A}^{-1}\| \cdot \|\mathbf{r}\|$$

Dividing both sized by $\|\mathbf{x}'\|$ and filling in the definition of $\text{cond}(\mathbf{A})$, we find

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}'\|} \leq \text{cond}(\mathbf{A})\frac{\|\mathbf{r}\|}{\|\mathbf{A}\| \cdot \|\mathbf{x}'\|}$$

Therefore, a small residual implies a small relative error in the solution *only* if the matrix $\mathbf{A}$ has a small condition number.

2.5 Summary

- Solving linear systems
    - Diagonal systems can be solved by substitution
    - Lower triangular systems can be solved by forward substitution
    - Upper triangular systems can be solved by backward substitution
- Transformations
    - Gauss transformations
        - make all entries of a vector below a certain index zero
        - Repeated Gauss transforms can make any matrix upper triangular.
        - Problems:
            1. The process breaks down if the leading diagonal entry of the matrix is zero at any stage.
            - This issue is trivial to solve: switch rows with a row that has a non-zero element: **pivoting**.
            2. In finite-precision arithmetic we wish to limit the size of the multipliers so that previous rounding errors do not get amplified.
            - The multipliers will never exceed 1 in magnitude if for each column we choose the entry of the largest magnitude on or below the diagonal as a pivot. Such a policy is called **partial pivoting**
    - Gauss Jordan transformations
        - make all entries of a vector above and below a certain index zero

        - Repeated Gauss Jordan transforms can make any diagonal.
        - The final solution phase is computationally somewhat cheaper because of the diagonal form of the matrix, but this does not suffice make up for the additional cost in the elimination phase.
        - In an implementation on parallel computers, the workload remains the same throughout the elimination phase and the final solutions can all be calculated at once.

    - It can also be used to calculate the inverse of a matrix explicitely by initializing the right-hand side of the matrix as the identity matrix $\mathbf{I}$.

    - Cholesky factorization

        - For symmetric positive definite systems.
        - Pivoting is not required
        - Only the lower triangle of $\mathbf{A}$ is accessed, and hence the strict upper triangular portion need not be stored
        - Only about $n^3/6$ multiplications and a similar number of additions are required.

- Complexity

    - LU factorization of an $n \times n$ matrix takes $n^3/3$ multiplications and additions, total $2n^3/3$
    - A complete matrix inversion takes about n^3 flops and thus is more expensive
    - Solving an LU-factorized system using forward and backward substitution takes about n^2 flops. For large systems, this is neglegible compared to the factorization phase.
    - Cramer's rule (in which the system is solved using ratios of determinants) is astronomically expensive

3 Linear Least squares

3.1 Introduction

Used when you have a system $\mathbf{Ax} = \mathbf{b}$ in which $\mathbf{A}$ no longer is a square matrix but an $m \times n$ matrix with $m > n$. (more equations than unknowns). We call this an *overdetermined* problem.

As there exists no exact solution, we want to minimize the residual $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$ as function of $\mathbf{x}$.

The best norm to use is the Euclidean norm (least *squares*).

3.2 Normal equations

We define

$\phi(\mathbf{x}) = \|\mathbf{r}\|_2^2 = \mathbf{r}^T\mathbf{r} = (\mathbf{b} - \mathbf{Ax})^T(\mathbf{b} - \mathbf{Ax}) = \mathbf{b}^T\mathbf{b} - \mathbf{x}^T\mathbf{A}^T\mathbf{b} - \mathbf{b}^T\mathbf{Ax} + \mathbf{x}^T\mathbf{A}^T\mathbf{Ax} = \mathbf{b}^T\mathbf{b} - 2\mathbf{x}^T\mathbf{A}^T\mathbf{b}$

To minimize this function, we want to find a point which satisfies $\nabla\phi(\mathbf{x}) = \mathbf{0}$
which is true for

$$\mathbf{0} = \nabla\phi(\mathbf{x}) = 2\mathbf{A}^T\mathbf{Ax} - 2\mathbf{A}^T\mathbf{b}$$

where we used $\frac{\mathrm{d}}{\mathrm{dx}}(\mathbf{a^T Qx}) = 2\mathbf{Qx}$.

In other words, to minimize $\mathbf{x}$ for ϕ we need to satisfy the $n \times n$ symmetric linear system

$$\mathbf{A}^T\mathbf{Ax} = \mathbf{A}^T\mathbf{b}$$

This system is known as the **normal equations**. The solution is unique if and only if the columns of A are linearly independent, i.e., rank $(\mathbf{A}) = n$. In this case the matrix $\mathbf{A}^T\mathbf{A}$ is positive definite. When some columns of $\mathbf{A}$ are linearly dependent, a manifold of solutions exists, i.e. rank $(\mathbf{A}) < n$.

3.3 Problem Transformations

Problem: $\mathrm{cond}(\mathbf{A}^T\mathbf{A}) = [\mathrm{cond}(\mathbf{A})]^2$
Solution: Find ways of solving that don't require explicit calculation of
$\mathbf{A}^T\mathbf{A}$

We can again abuse triangular systems, but can't use Gauss Eliminations because they don't preserve the Euclidean norm.

3.3.1 Orthogonal Transformations

A square real matrix $\mathbf{Q}$ is *orthogonal* if its columns are *orthonormal*, meaning that $\mathbf{Q^T Q} = \mathbf{I}$.

Such an **orthogonal transformation** $\mathbf{Q}$ preserves the Euclidean norm of any vector $\mathbf{v}$:

$$\|\mathbf{Qv}\|_2^2 = (\mathbf{Qv})^T\mathbf{Qv} = \mathbf{v}^T\mathbf{Q}^T\mathbf{Qv} = \mathbf{v^Tv} = \|\mathbf{v}\|_2^2$$

These preserve norms and thus don't amplify errors, BUT are usually more expensive than simpler tranformations.

3.3.2 Triangular LQ problems and QR Factorization

Of form

$$\begin{bmatrix}\mathbf{R}\\ \mathbf{O}\end{bmatrix}\mathbf{x} \cong \begin{bmatrix}\mathbf{c_1}\\ \mathbf{c_2}\end{bmatrix}$$

with $\mathbf{R}$ an $n \times n$ upper triangular matrix.
The residual is given by

$$\|\mathbf{r}\|_2^2 = \|\mathbf{c_1} - \mathbf{Rx}\|_2^2 + \|\mathbf{c_2}\|_2^2$$

If we solve the triangular system $\mathbf{Rx} = \mathbf{c_1}$ (which can easily be achieved with back-substitution) we have found the least squares solution $\mathbf{x}$ and we can conclude that the minimum sum of squares is

$$\|\mathbf{r}\|_2^2 = \|\mathbf{c_2}\|_2^2.$$

Tranforming a system to triangular form is accomplished by **QR factorization**, which, for an $m \times n$ matrix $\mathbf{A}$ with $m > n$ has the form

$$\mathbf{A} = \mathbf{Q}\begin{bmatrix}\mathbf{R}\\ \mathbf{O}\end{bmatrix}$$

where $\mathbf{Q}$ is an $m \times m$ orthogonal matrix and $\mathbf{R}$ is an $n \times n$ upper triangular matrix.

The transformed right-hand side then reads

$$\mathbf{Q}^T\mathbf{b} = \begin{bmatrix}\mathbf{c_1}\\ \mathbf{c_2}\end{bmatrix}$$

The residual equals

$$\|\mathbf{r}\|_2^2 = \|\mathbf{b} - \mathbf{Ax}\|_2^2 = \|\mathbf{b} - \mathbf{Q}\begin{bmatrix}\mathbf{R}\\ \mathbf{O}\end{bmatrix}\mathbf{x}\|_2^2 = \|\mathbf{Q^{Tb}} - \begin{bmatrix}\mathbf{R}\\ \mathbf{O}\end{bmatrix}\mathbf{x}\|_2^2 = \|\mathbf{c_1} - \mathbf{Rx}\|_2^2 + \|\mathbf{c_2}\|_2^2$$

A few common choices for QR factorization are:

- Householder transformations
- Givens transformations
- Gram-Schmidt orthogonalization

3.3.3 Householder Transforms

Is given by

$$\mathbf{H} = \mathbf{I} - 2\frac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}}$$

with $\mathbf{v}$ a nonzero vector.
It can be shown that $\mathbf{H} = \mathbf{H}^{-1} = \mathbf{H}^{\mathbf{T}}$, which means that $\mathbf{H}$ is orthogonal and symmetric.

If we split up a given m-vector $\mathbf{a}$ as

$$\mathbf{a} = \begin{bmatrix}\mathbf{a}_1 \\ \mathbf{a}_2\end{bmatrix}$$

where $\mathbf{a}_1$ is a $(k-1)$-vector with $1 \leq k < m$.

If we then take the householder vector to be

$$\mathbf{v} = \begin{bmatrix}\mathbf{0} \\ \mathbf{a}_2\end{bmatrix} - \alpha\mathbf{e}_k$$

where $\alpha = -\text{sign}(a_k)\|\mathbf{a}_2\|_2$, then the resulting Householder transformation annihilates the last $m-k$ components of $\mathbf{a}$.

Similarly to the LU composition, we get

$$\mathbf{H}_n \dots \mathbf{H}_1\mathbf{A} = \begin{bmatrix}\mathbf{R} \\ \mathbf{O}\end{bmatrix}$$

We also define

$$\mathbf{Q}^T = \mathbf{H}_n \dots \mathbf{H}_1 \qquad \text{or, equivalently} \qquad \mathbf{Q} = \mathbf{H}_n^T \dots \mathbf{H}_1^T$$

such that

$$\mathbf{A} = \mathbf{Q}\begin{bmatrix}\mathbf{R} \\ \mathbf{O}\end{bmatrix}$$

To solve the least squares system $\mathbf{A}\mathbf{x} \cong \mathbf{b}$, we solve the equivalent system

$$\begin{bmatrix}\mathbf{R} \\ \mathbf{O}\end{bmatrix}\mathbf{x} \cong \mathbf{Q}^T\mathbf{b} = \begin{bmatrix}\mathbf{c_1} \\ \mathbf{c}_2\end{bmatrix}$$

3.4 Rank deficiency

If $\text{rank}(\mathbf{A}) < n$. You can still perform a QR factorisation of $\mathbf{A}$ but the upper triangular matrix will be singular. This means that multiple $\mathbf{x}$ vectors with the same minimal norm exist. Such situations arise when using a wrong model, a corrupt data source, or a poorly designed experiment.

3.5 Singular Value Decomposition

We decompose $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathbf{T}}$$

where $\mathbf{U}$ is an $m \times m$ orthogonal matrix, $\mathbf{V}$ is an $n \times n$ orthogonal matrix, and $\mathbf{\Sigma}$ is an $m \times n$ diagonal matrix, with

$$\sigma_{ij} = \begin{cases} 0, & \text{for } i \neq j. \\ \sigma_i \geq 0, & \text{for } i = j. \end{cases}$$

The diagonal entries σ_i are called the **singular values** of $\mathbf{A}$ and are usually ordered so that $\sigma_{i-1} \geq \sigma_i, i = 2, \ldots, \min\{m, n\}$. The columns $\mathbf{u_i}$ of $\mathbf{U}$ and $\mathbf{v_i}$ of $\mathbf{V}$ are the corresponding left and right **singular vectors**.

The least-squares solution to $\mathbf{Ax} \cong \mathbf{b}$ of minimum Euclidean norm is given by

$$\mathbf{x} = \sum_{\sigma_i \neq 0} \frac{\mathbf{u_i^T b}}{\sigma_i} \mathbf{v}_i$$

The matrix norm corresponding to the Euclidean vector norm is equal to the largest singular value of the matrix,

$$\|\mathbf{A}\|_2 = \max_{\mathbf{x} \neq \mathbf{0}} \frac{\|\mathbf{Ax}\|_2}{\|\mathbf{x}\|_2} = \sigma_{\max}$$

The condition number of an arbitrary matrix $\mathbf{A}$ is given by the ratio

$$\operatorname{cond}_2(\mathbf{A}) = \frac{\sigma_{\max}}{\sigma_{\min}}$$

Note that we find $\operatorname{cond}_2(\mathbf{A}) = \infty$ for singular matrices, because there, $\sigma_{\min} = 0$.

The rank of a matrix is equal to the number of nonzero singular values it has.

The **pseudoinverse** of a general matrix $\mathbf{A}$ is given by

$$\mathbf{A}^+ = \mathbf{V}\mathbf{\Sigma}^+\mathbf{U^T}$$

- If the matrix $\mathbf{A}$ is square and nonsingular this definition agrees with $\mathbf{A}^{-1}$.
- In all cases, the solution to a least squares problem $\mathbf{Ax} \cong \mathbf{b}$ is given by $\mathbf{A}^+\mathbf{b}$.

An other (computationally less good) way to find the pseudo-inverse can be obtained via the normal equations

$$\mathbf{A}^T\mathbf{Ax} = \mathbf{A}^T\mathbf{b}$$

we see that

$$\mathbf{x} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}$$

is a solution of the least squares problem $\mathbf{Ax} \cong \mathbf{b}$.

Consequently, the pseudoinverse $\mathbf{A}^+$ is also given by

$$\mathbf{A}^+ = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T$$

3.6 Sensitivity and Condition Number

Let's now also generalize the expression,

$$\frac{\|\Delta \mathbf{x}\|}{\|\mathbf{x}'\|} \leq \text{cond}(\mathbf{A}) \frac{\|\mathbf{r}\|}{\|\mathbf{A}\| \cdot \|\mathbf{x}'\|}$$

which we found for square systems.

We start from the solution $\mathbf{x}$ to the least squares problem, i.e.

$$\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 \text{ is minimal}$$

Similarly, denote with $\mathbf{x}' = \mathbf{x} + \mathbf{\Delta x}$ the solution to a problem with a perturbed right hand side $\mathbf{b} + \mathbf{\Delta b}$:

$$\|\mathbf{A}(\mathbf{x} + \mathbf{\Delta x}) - (\mathbf{b} + \mathbf{\Delta b})\|_2 \text{ is minimal}$$

From the normal equations we know that

$$\mathbf{A}^T\mathbf{A}(\mathbf{x} + \mathbf{\Delta x}) = \mathbf{A^T}(\mathbf{b} + \mathbf{\Delta b})$$

Thus,

$$\mathbf{\Delta x} = \mathbf{A}^+ \mathbf{\Delta b}$$

Taking norms and dividing by $\|\mathbf{x}\|_2$ we find

$$\frac{\|\mathbf{\Delta x}\|_2}{\|\mathbf{x}\|_2} \leq \frac{\|\mathbf{A}^+\|_2 \|\mathbf{\Delta b}\|_2}{\|\mathbf{x}\|_2}$$

Using the definition $\text{cond}(\mathbf{A}) = \|\mathbf{A}\|_2 \cdot \|\mathbf{A}^+\|_2$ and multiplying both the denominator and enumerator in the right-hand-side with $\|\mathbf{b}\|_2$ we find

$$\frac{\|\mathbf{\Delta x}\|_2}{\|\mathbf{x}\|_2} \leq \frac{\text{cond}(\mathbf{A})}{\|\mathbf{A}\|_2} \frac{\|\mathbf{b}\|_2 \|\mathbf{\Delta b}\|_2}{\|\mathbf{b}\|_2 \|\mathbf{x}\|_2} \leq \text{cond}(\mathbf{A}) \frac{\|\mathbf{\Delta b}\|_2}{\|\mathbf{b}\|_2} \frac{\|\mathbf{b}\|_2}{\|\mathbf{A}\mathbf{x}\|_2} = \text{cond}(\mathbf{A}) \frac{1}{\cos(\theta)} \frac{\|\mathbf{\Delta b}\|_2}{\|\mathbf{b}\|_2}$$

where $\cos(\theta) = \frac{\|\mathbf{A}\mathbf{x}\|_2}{\|\mathbf{b}\|_2}$ denotes the cosine of the angle θ between the vectors $\mathbf{b}$ and $\mathbf{A}\mathbf{x}$.

> To see why this is true, start from the inner product
>
> $$(\mathbf{A}\mathbf{x})^T \mathbf{b} = \|\mathbf{A}\mathbf{x}\|_2 \|\mathbf{b}\|_2 \cos(\theta)$$
>
> rearrange to isolate $\cos(\theta)$
>
> $$\cos(\theta) = \frac{(\mathbf{A}\mathbf{x})^T \mathbf{b}}{\|\mathbf{A}\mathbf{x}\|_2 \|\mathbf{b}\|_2}$$
>
> make use of the normal equation to find
>
> $$\cos(\theta) = \frac{\mathbf{x}^T \mathbf{A}^T \mathbf{b}}{\|\mathbf{A}\mathbf{x}\|_2 \|\mathbf{b}\|_2} = \frac{\mathbf{x}^T \mathbf{A}^T \mathbf{A}\mathbf{x}}{\|\mathbf{A}\mathbf{x}\|_2 \|\mathbf{b}\|_2} = \frac{(\mathbf{A}\mathbf{x})^T (\mathbf{A}\mathbf{x})}{\|\mathbf{A}\mathbf{x}\|_2 \|\mathbf{b}\|_2}$$

> Recognize that the numerator represents the square of the Euclidean norm of the vector $\mathbf{Ax}$
>
> $$\cos(\theta) = \frac{\|\mathbf{Ax}\|_2^2}{\|\mathbf{Ax}\|_2\|\mathbf{b}\|_2} = \frac{\|\mathbf{Ax}\|_2}{|\mathbf{b}\|_2}$$

In contrast to the result for square matrices, it does not longer hold that the sensitivity of the solution only depends on the matrix $\mathbf{A}$, but now also depends on the right-hand-side vector $\mathbf{b}$.

3.7 Summary

- The easiest method to implement are the normal equations (which only require matrix multiplications and Cholesky decomposition). However, this method is computationally quite expensive and the error is proportional to $[\text{cond}(\mathbf{A})]^2$, which means it can break down quite easily
- The most efficient and accurate orthogonalization method (for dense matrices at least) is typically the Householder method. For square systems, it requires about the same amount of work as the normal equations, but for strongly overdetermined systems, it becomes only about half as efficient. On the other hand, it is much more broadly applicable due to its better accuracy.
- SVD is the most expensive method, but also offers superb robustness and reliability.

4 Eigenvalue problems

4.1 Introduction

Equations of the form

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$$

This often comes down to finding roots of the characteristic polynomial of A

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0$$

Calculating the roots of its charactersistic polynomial is not a good numerical way to find the eigenvalues of a matrix of nontrivial size:

- Computing the coefficients of the characteristic polynomial for a large matrix is in itself already expensive
- The coefficients of the characteristic polynomial can be highly sensitive to small perturbations in $\mathbf{A}$ -> unstable
- Rounding errors in finding the characteristic polynomial can destroy the accuracy of the roots
- Computing the roots of a polynomial of high degree is a nontrivial and substantial task (Fundamental theory of Algebra)

We can exploit similarity these properties for more efficient algorithms:

- If $\mathbf{A}$ is symmetric/Hermitian, all its eigenvalues are real.
- **Shift**: if $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$ and σ any scalar, then $(\mathbf{A} - \sigma\mathbf{I}) = (\lambda - \sigma)\mathbf{x}$; The eigenvalues are shifted by σ, but the eigenvectors remain unchanged.
- **Inversion**: $\mathbf{A}^{-1}$ has the same eigenvectors as $\mathbf{A}$, and eigenvalues $1/\lambda$
- **Powers**: $\mathbf{A}^k$ has the same eigenvectors as $\mathbf{A}$, and eigenvalues λ^k
- **Polynomials**: for a general polynomial $p(t)$, $p(\mathbf{A})\mathbf{x} = p(\lambda)\mathbf{x}$. Thus the eigenvalues of a polynomial in a matrix $\mathbf{A}$ are given by the same polynomial, evaluated at the eigenvalues of $\mathbf{A}$ and the corresponding eigenvectors remain the same as those of $\mathbf{A}$.
- **Similarity**: A matrix $\mathbf{B}$ is *similar* to a matrix $\mathbf{A}$ if there exists an invertible matrix $\mathbf{T}$ such that

  $$\mathbf{B} = \mathbf{T}^{-1}\mathbf{A}\mathbf{T}$$

  It follows that:

  $$\mathbf{B}\mathbf{y} = \lambda\mathbf{y} \Rightarrow \mathbf{T}^{-1}\mathbf{A}\mathbf{T}\mathbf{y} = \lambda\mathbf{y} \Rightarrow \mathbf{A}\mathbf{T}\mathbf{y} = \lambda\mathbf{T}\mathbf{y}$$

  In other words, $\mathbf{B} = \mathbf{T}^{-1}\mathbf{A}\mathbf{T}$ has the same eigenvalues as $\mathbf{A}$, but systematically transforms its eigenvectors.

4.2 Calculating Eigenvalues and Eigenvectors

4.2.1 Power iteration

Repeatedly aplying $\mathbf{A}$ to an arbitrary non-zero vector.

Assuming that $\mathbf{A}$ has a unique eigenvalue λ_1 of maximum modulus, with corresponding eigenvector $\mathbf{v}_1$, power iteration converges to a multiple of $\mathbf{v}_1$.

> **PROOF**
>
> Assume that we can express the starting vector $\mathbf{x}_0$ as a linear combination $\mathbf{x}_0 = \sum_{j=1}^{n} \alpha_j \mathbf{v}_j$, with $\mathbf{v}_j$ the eigenvectors of $\mathbf{A}$.
>
> $$\mathbf{x}_k = \mathbf{A}\mathbf{x}_{k-1} = \mathbf{A}^2\mathbf{x}_{k-2} = \ldots = \mathbf{A}^k\mathbf{x}_0$$
> $$= \mathbf{A}^k \sum_{j=1}^{n} \alpha_j \mathbf{v}_j = \sum_{j=1}^{n} \alpha_j \mathbf{A}^k \mathbf{v}_j = \sum_{j=1}^{n} \lambda_j^k \alpha_j \mathbf{v}_j$$
> $$= \lambda_1^k \left(\alpha_1 \mathbf{v}_1 + \sum_{j=2}^{n} (\lambda_j/\lambda_1)^k \alpha_j \mathbf{v}_j \right)$$

Power iteration usually works well in practice, but might fail:

- The starting vector $\mathbf{x}_0$ may have *no* component in the dominant eigenvector $\mathbf{v}_1$. In practice this is very unlikely and is mitigated after a few iterations due to rounding errors that introduce such a component.
- There may be more than 1 eigenvalue with the same maximum modulus, in which case the algorithm might converge to a linear combination of the corresponding eigenvectors.
- For a real matrix and real starting vector, the iteration can never converge to a complex vector.

Geometric growth of the components at each iteration risks overflow or underflow, so in practice the approximate eigenvector is rescaled to have norm 1 at every iteraction. Then, $\mathbf{x}_k \to \mathbf{v}_1 / \|\mathbf{v}_1\|_\infty$ and $\|\mathbf{y}_k\|_\infty \to \|\lambda_1\|$.

The convergence rate of power iteration is linear (and proportional with $\|\lambda_2/\lambda_1\|$, where λ_2 is the eigenvalue with second largest modulus.

4.2.2 Inverse Iteration

For some applications we're interested in the smallest eigenvalue of a matrix. Then we can make use of the fact that eigenvalues of $\mathbf{A}^{-1}$ are $1/\lambda$. This suggests to use power iteration on the inverse of $\mathbf{A}$, but the inverse of $\mathbf{A}$ does not need to be calculated explictly.

Instead, the equivalent system of linear equations is solved at each iteration using the triangular factors resulting from e.g. LU-factorization of $\mathbf{A}$, which need only to be calculated once. Using $\mathbf{L}$ and $\mathbf{U}$, we can then efficiently solve $\mathbf{A}\mathbf{y} = \mathbf{x}$ using forward and backward substitution.

Inverse iteration converges to the eigenvector corresponding to the largest eigenvalue of $\mathbf{A}^{-1}$, which is the smallest eigenvalue of $\mathbf{A}$.

By shifting the matrix $\mathbf{A}$ to $\mathbf{A} - \sigma\mathbf{I}$, all eigenvalues are also shifted by σ.
In case of reverse iteration this approach gives some flexibility in which eigenvalue is found because we converge to the eigenvalue closest to σ. Also, when the shift is already a close approximation of the eigenvalue, the convergence is very rapid.

4.2.3 Rayleigh Quotient Iteration

Given an approximate eigenvector $\mathbf{x}$ for a real matrix $\mathbf{A}$, finding the best estimate for the corresponding eigenvalue λ can be considered as a linear least squares approximation problem:

$$\mathbf{x}\lambda \cong \mathbf{A}\mathbf{x}$$

It's solution, the **Rayleigh quotient** is given by

$$\lambda = \frac{\mathbf{x^T A x}}{\mathbf{x^T x}}$$

This is a better approximation for the eigenvalue than the one obtained at each stage in the power iteration algorithm.

Given an approximate eigenvector, the Rayleigh quotient provides a good estimate for the corresponding eigenvalue. Conversely, inverse iteration converges very rapidly to an eigenvector if an approximate eigenvalue is used as shift. When combining these ideas we arrive at **Rayleigh quotient iteration**.

4.2.4 Deflation

The process of **deflation** removes a known eigenvalue from a matrix, so that further eigenvalues and eigenvectors can be determined.

This can be achieved by letting $\mathbf{u_1}$ be any vector such that $\mathbf{u_1^T x_1} = \lambda_1$.
Then the matrix $\mathbf{A} - \mathbf{x_1 u_1^T}$ has eigenvalues $0, \lambda_2, \ldots, \lambda_n$.

We're not going to look deeper into this procedure because

- it becomes increasingly cumbersome and numerically less accurate to find eigenvalues using deflation (so that inverse iteration using the estimated eigenvalues as a shift are necessary)
- there are better ways to find many eigenvalues of a matrix.

4.3 QR Iteration

In practice, the fastest and most used method to find the eigenvalues of a matrix is **QR-iteration**.
Starting from a matrix $\mathbf{A}$, we define the following sequence:

$$\mathbf{A}_m = \mathbf{Q}_m \mathbf{R}_m$$
$$\mathbf{A}_{m+1} = \mathbf{R}_m \mathbf{Q}_m$$

This sequence will converge to a triangular matrix with the eigenvalues of $\mathbf{A}$ on its diagonal, or a near-triangular form, which easily allows to calculate the eigenvalues.

> This isn't in the theory PDF's but how it works is like this:
> First observe that $\mathbf{A}_m$ and $\mathbf{A}_{m+1}$ have the same eigenvalues, as
>
> $$A_{m+1} = R_m Q_m = Q_m^{-1} Q_m R_m Q_m = Q_m^{-1} A_m Q_m = Q_m^{\mathsf{T}} A_m Q_m$$
>
> which is a similarity transform. Then realize that what we're basically doing is power iteration *but with all eigenvectors at once*.
> We need to orthonormalize all vectors at each iteration step, otherwise all eigenvalues would just converge to a multiple of the largest eigenvalue. Explict QR factorization prohibits the eigenvalues from converging in this way. Also see V&F cursus for more details.

4.4 Summary

- Power iteration
  - Works well in practice
  - The starting vector $\mathbf{x}_0$ may have *no* component in the dominant eigenvector $\mathbf{v}_1$. In practice this is very unlikely and is mitigated after a few iterations due to rounding errors that introduce such a component.
  - There may be more than 1 eigenvalue with the same maximum modulus, in which case the algorithm might converge to a linear combination of the corresponding eigenvectors.
  - For a real matrix and real starting vector, the iteration can never converge to a complex vector.
- Inverse iteration
  - Inverse iteration converges to the eigenvector corresponding to the largest eigenvalue of $\mathbf{A}^{-1}$, which is the smallest eigenvalue of $\mathbf{A}$.
- Rayleigh Quotient iteration
  - better approximation for the eigenvalue than the one obtained at each stage in the power iteration algorithm.
- QR iteration
  - Calculates all eigenvalues at once
  - In practice the fastest and most used method

5 Nonlinear Equations

5.1 Introduction

In analogy to linear equations, where a system of equations is written as $\mathbf{Ax} = \mathbf{b}$, we could write down a system of nonlinear equations as $\mathbf{f(x)} = \mathbf{y}$.

However, it is more customary to subtract $\mathbf{y}$ from $\mathbf{f(x)}$ so the equation that needs to be solved is expressed as $\mathbf{f(x)} = \mathbf{0}$.

A solution value $\mathbf{x}$ such that $\mathbf{f(x)} = \mathbf{0}$ is called a **root** of the equation, and a **zero** of the function $\mathbf{f}$.
This problem thus is referred to as **root finding** or **zero finding**.

5.2 Number of solutions

This not entirely trivial.

> **Examples**
>
> Even in 1 dimension, many different cases are possible:
>
> - $e^x + 1 = 0$ has no solution.
> - $e^{-x} - x = 0$ has one solution.
> - $x^2 - 4\sin(x) = 0$ has two solutions.
> - $x^3 - 6x^2 + 11x - 6 = 0$ has three solutions.
> - $\sin(x) = 0$ has infinitely many solutions.

For a nonlinear equation it is possible to have degenerate solutions, which are called **multiple roots**. Generally, for a smooth function f, if $f(x^*) = f'(x^*) = f''(x^*) = \cdots = f^{(m-1)}(x^*) = 0$ and $f^m(x^*) \neq 0$, then x^* is a root of **multiplicity** m.

If $m = 1$, then the solution is not degenerate and is called a **simple root**.

Geometrically, this means that the curve defined by f has a horizontal tangent at the x-axis.

5.3 Sensitivity

In one dimension the condition number for the root-finding problem of f near x^* is $\frac{1}{\|f'(x^*)\|}$.

For functions for which $f'(x)$ is small near the root, the error in the root finding problem can be substantial.

At a multiple root x^*, $f'(x^*) = 0$, so the condition number of a multiple root is infinite.
Intuitively this is clear because a small change in the parameters of f can cause the multiple

root to disappear or split up in more than one root.

5.4 Convergence Rates and Stopping Criteria

The **convergence rate** is the effectiveness with which a certain algorithm reaches it s solution.

To solve a nonlinear equation, one often has the choice between several iterative methods, with different converge rates. The total cost of solving the system, depends on the amount of iterations necessary to reach the solution with the desired accuracy AND the computational complexity of a single iteration.

Let $\mathbf{e}_k = \mathbf{x}_k - x^*$ be the error at iteration k, where $\mathbf{x}_k$ is the approximate solution at iteration k and x^* the (usually unknown) true solution.

An iterative method is said to converge with rate r if

$$\lim_{k \to \infty} \frac{\|\mathbf{e}_{k+1}\|}{\|\mathbf{e}_k\|^r} = C$$

for some finite constant $C > 0$.

Interesting cases are:

- $r = 1$ and $C < 1$: *linear* convergence
- $r > 1$: *superlinear* convergence
- $r = 2$: *quadratic* convergence
- $r = 3$: *cubic* convergence

In an iterative method, the solution gains an additional r number of correct digits as compared to the previous iteration.

The convergence of a certain algorithm tells us that we zoom in on the correct solution at a certain rate, but it doesn't tell us the current accuracy of our solution at any given iteration.

Therefore, we don't know whether we reached a solution that is sufficiently close to the real solution to decide that we can stop the algorithm.

More often than not, it's not trivial to define a suitable **stopping criterion**.
A reasonable way is to look at the relative change in the solutions for successive iterations $\|\mathbf{x}_{k+1} - \mathbf{x}_k\| / \|\mathbf{x}_k\| < \varepsilon$, and check that this quantity becomes smaller than a predefined **error tolerance** ε.

A sensible value for ε *might* be (but this really depends on your specific problem) the double precision accuracy of 10^{-16}.

5.5 Solving Nonlinear equations in 1D

5.5.1 Bisection method

We look for a (short) interval $[a, b]$ in which f changes sign.
Such a **bracket** ensures that the function must take a zero value somewhere within this interval.

The **bisection method** begins with an initial bracket and then iteratively reduces its length until the desired accuracy is reached.

At every iterations, the function is evaluated at the **midpoint** of the interval, such that half of the interval can be discarded, based on the sign of the function value at the midpoint.

The bisection method makes no use of the magnitudes of the function values, and as a result it is certain to converge, but very slowly. At each iteration, the bound on the possible error is reduced by half, meaning that it converges linearly with $r = 1$ and $C = 0.5$.

Given a starting interval $[a, b]$, the length of the interval after k iterations is $(b - a)/2^k$, so that achieving an error tolerance of ε requires

$$\left\lceil \log_2 \left(\frac{b-a}{\varepsilon} \right) \right\rceil$$

iterations, regardless of the particular function f involved.

5.5.2 Fixed Point Iteration

Given a function $g : \mathbb{r} \to \mathbb{R}$, a value x such that $x = g(x)$ is called a **fixed point** of the function g, since x remains unchanged when g is applied to it.

This problem is important because many iterative algorithms for solving nonlinear equations (see below) are based on iterations of the form

$$x_{k+1} = g(x_k)$$

The simplest way to characterize the behavior of an iterative scheme $x_{k+1} = g(x_k)$ for a fixed-point problem $x = g(x)$ is to look at the derivative of g in the solution x^*.
It is a rule that if $x^* = g(x)$ and $\|g'(x^*)\| < 1$, then the iterative scheme is **locally convergent**.
If however $\|g'(x^*)\| > 1$, then the scheme diverges for every initial value different from x^*.

Proof

If x^* is a fixed point, then the error at the k-th iteration is

$$e_{k+1} = x_{k+1} - x^* = g(x_k) - g(x^*)$$

There exist a point θ_k between x_k and x^* for which

$$g(x_k) - g(x^*) = g'(\theta_k)(x_k - x^*)$$

so

$$e_{k+1} = g'(\theta_k)e_k$$

> We do not know the value of θ_k, but if $\|g'(x^*)\| < 1$, then by starting the iteration sufficiently close to x^*, there exists a constant C for which $\|g'(\theta_k)\| \leq C < 1$, for $k = 0, 1, \ldots$.
>
> Thus we have
>
> $$\|e_{k+1}\| \leq C\|e_k\| \leq \ldots \leq C^k\|e_{e_0}\|$$
>
> As $C < 1$ implies $C^k \to 0$, also $\|e_k\| \to 0$ and the sequence converges.

The convergence rate of the iterative scheme is linear with $C = \|g'(x^*)\|$. The smaller this constant, the faster the convergence. Ideally, we have $\|g'(x^*)\| = 0$, in which case the Taylor expansion gives

$$g(x_k) - g(x^*) = g''(\xi_k)(x_k - x^*)/2$$

with ξ_k between x_k and x^*. This yields

$$\lim_{k \to \inf} \frac{\|e_{k+1}\|}{\|e_k\|}^2 = \frac{g''(x^*)}{2}$$

In this case the *rate of convergence becomes quadratic* In the next sections we'll see methods to systematically choose g to reach this quadratic convergence.

5.5.3 Newton's method

We start from the truncated Taylor series

$$f(x + h) \approx f(x) + hf'(x)|_{x_0}$$

which is a linear function of h that approximates f near a given x.
It's zero is easily determined to be $h = -f(x)/f'(x)|_{x_0}$, assuming that $f'(x)|_{x_0} \neq 0$.
Because the zeros of both functions are not identical, this procedure is repeated in an iterative scheme, called **Newton's method**

This method can be seen as a systematic way of transforming a nonlinear equation $f(x) = 0$ into a fixed-point problem $x = g(x)$, where

$$g(x) = x - f(x)/f'(x)$$

To study the convergence of this scheme, we determine the derivative

$$g'(x) = f(x)f''(x)/(f'(x))^2$$

- For simple roots ($f(x^*) = 0$ and $f'(x^*) \neq 0$), $g'(x^*) = 0$. Thus the asymptotic convergence rate of Newton's method is quadratic.
- For a multiple root with multiplicity m, it is only linearly convergent, with constant $C = 1 - (1/m)$.

Take note that these convergences are only local and it may not converge at all unless started sufficiently close to the solution.

5.5.4 Secant method

One drawback of Newton's method is that both the function and its derivative needs to be explicitly and evaluated at every iteration. In the **Secant method** the derivative is replaced by a finite difference approximation on successive iterates:

$$f'(x_k) = \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}$$

Requires only one new function evaluation per iteration, but has the disadvantage of requiring two starting guesses and converging more slowly (subquadratically but still faster than linear with $r \approx 1.618$).

The lower cost per iteration often more than offsets the larger number of iterations required, such that the total cost of finding a root is often less for the secant method than for Newton's method.

5.5.5 Inverse Interpolation

The secant method fits a straight line to two values of the function for each iteration.
Its convergence rate can be improved (but not made to exceed $r = 2$) by fitting a higher order polynomial instead of a straight line.

This has however the drawbacks that the zeros of the fitted polynomial might be difficult to compute, or might not exist at all.

Instead, we can use **inverse interpolation** where, instead of fitting a polynomial to values $f(x_k)$ as function of the values x_k, we do the opposite:
we fit a polynomial p to the values x_k as function of the values $f(x_k)$.
The next approximate solution is than simply $p(0)$.

The most used implementation of this idea is **inverse quadratic interpolation** where a parabola is fitted through the values obtained at the last 3 iterations.
Similar to the secant method this only requires one additional function evaluation per iteration, but requires a little more memory and overhead in fitting the parabola.
This algorithm has a converge rate of $r \approx 1.839$.

5.5.6 More than one root

All methods we saw until now zoom in on a single root of the function under study.
Sometimes we're interested in all of the roots of e.g. a polynomial function.

For a polynomial $p(x)$ of degree n, we want to find all n zeros (which might be complex).

To this end we can resort to several methods:

- Use one of the methods shown above to find one root x_1 and then deflate the polynomial $p(x)$ to $p(x)/(x-x_1)$ which has a degree that is one lower and repeat the process.
  Note that it's a good idea to zoom in on each of the obtained roots using the approximate values used this way to avoid any numerical errors introduced in the deflating process.
- Use a dedicated (complex) routine specifically designed for this purpose.
  These work by isolating the roots of a polynomial in the complex plane, and then refining in a way similar to the bisection method to zoom in on each of the roots.
- Form the **companion matrix** of the given polynomial and use an eigenvalue routine to find its eigenvalues, which are also the roots of the polynomial.

5.6 Systems of nonlinear equations

Systems of nonlinear equations are more difficult to solve than single non-linear equations for a number of reasons:

- A much wider range of behavior is possible, so we don't get as far with theoretical analysis of the existence and number of solutions.
- There is no simple way to bracket a desired solution.
- Computational overhead increases rapidly with the dimension of the problem.

One method that works in more dimensions is Newton's method:

For a differentiable vector function $\mathbf{f}$, the truncated Taylor series reads:

$$\mathbf{f}(\mathbf{x}+\mathbf{s}) \approx \mathbf{f}(\mathbf{x}) + \mathbf{J}_f(\mathbf{x})\mathbf{s}$$

where $\mathbf{J_f(x)}$ is the Jacobian matrix of $\mathbf{f}$ with elements

$$\{\mathbf{J_f(x)}\}_{ij} = \frac{\partial f_i(\mathbf{x})}{\partial x_j}$$

If $\mathbf{s}$ satisfies the linear system $\mathbf{J}_f(\mathbf{x})\mathbf{s} = -\mathbf{f(x)}$, then $\mathbf{x}+\mathbf{s}$ is taken as an approximate zero of $\mathbf{f}$.

The computational cost of Newton's method in n dimensions is substantial:

- Evaluating the Jacobian matrix (or approximating it) requires n^2 function evaluations.
- Solving the system $\mathbf{J}_f(\mathbf{x})\mathbf{s} = -\mathbf{f(x)}$, for instance using LU-factorization, costs $\mathcal{O}(n^3)$ operations.

5.7 Summary

- Nonlinear equations in 1D
    - Bisection method

        - Sure to converge, but really slow.
    - Fixed point iteration
        - only works for a particular equation
        - convergence is linear (most of the time)
    - Newton's method
        - quadratic convergence
        - for a multiple root with multiplicity m, only linearly convergent with constant $C = 1 - (1/m)$
        - only local convergence
        - works in multiple dimensions
    - Secant method
        - derivative doesn't need to be calculated analytically
        - requires two starting guesses
        - converges subquadratically
        - lower cost per iteration than Newton's method, but more iterations needed (most of the time this method is less costly than Newton)
    - Inverse interpolation
        - requires only one additional function evaluation per iteration, but requires more memory and overhead.
- Systems of non-linear equations
    - A much wider range of behavior is possible, so we don't get as far with theoretical analysis of the existence and number of solutions.
    - There is no simple way to bracket a desired solution.
    - Computational overhead increases rapidly with the dimension of the problem.

6 Optimization

6.1 Introduction

Given a function $f : \mathbb{R}^n \to \mathbb{R}$, and a set $S \subseteq \mathbb{R}^n$, we seek $\mathbf{x}^* \in S$ such that f attains a minimum on S at $\mathbf{x}^*$, i.e. $f(\mathbf{x}^*) \leq f(\mathbf{x})$ for all $\mathbf{x} \in S$.

Such a point $\mathbf{x}^*$ is called a **minimizer**, or simply a **minimum** of f. A maximum of f is a minimum of $-f$, so it suffices to consider minimization.

The **objective function** f may be linear of nonlinear, and it is usually assumed to be differentiable.
The set S is usually defined by a set of equations and inequalities, called **constraints**, which may be linear of nonlinear.
Any vector $\mathbf{x} \in S$, i.e. that satisfies the constraints, is called a **feasible point**, and S is called the **feasible set**.
If $S = \mathbb{R}^n$, the problem is **unconstrained**

A general **continuous** optimization problem (note that we will not address **discrete** optimization problems) has the form

$$\min_{\mathbf{x}} f(\mathbf{x}) \quad \text{subject to} \quad \mathbf{g}(\mathbf{x}) = \mathbf{0} \quad \text{and} \quad \mathbf{h}(\mathbf{x}) \leq \mathbf{0}$$

where $f : \mathbb{R}^n \to \mathbb{R}$, $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^m$ and $\mathbf{h} : \mathbb{R}^n \to \mathbb{R}^p$.

Optimization problems are classified by the properties of the functions involved.
For example if f, $\mathbf{g}$ and $\mathbf{h}$ are all linear, the we have a **linear programming** problem.
If any of them are nonlinear, we have a **nonlinear programming** problem.

What constitutes a solution to an optimization problem?
A **global minimum** satisfies $f(\mathbf{x}^*) \leq f(\mathbf{x})$ for *any* feasible point $\mathbf{x}$.
Finding such a global minimum, or even verifying that a point is a global minimum is difficult unless the problem has special properties.

Most optimization methods use local information, such as derivatives, and consequently are designed to find a **local minimum**.
Often the best one can do to find a global minimum is use a very large set of starting points, widely scattered throughout the feasible set.
The lowest minimum found this way has a good (but not perfect) chance of being the global minimum.

The **first-order necessary condition** for a minimum is that the **gradient** of the objective function f is zero.

$$\nabla f(\mathbf{x}^*) = 0$$

Consider the **Hessian matrix** of f.
This is a matrix-valued function $\mathbf{H}_f$ (only defined if f is twice differentiable).

$$\mathbf{H}_f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1^2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_2^2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_n^2} \end{bmatrix}$$

We can then classify critical points as follows:

At a critical point $\mathbf{x}^*$, where $\nabla f(\mathbf{x}) = \mathbf{0}$, if $\mathbf{H}_f(\mathbf{x}^*)$ is...

- Positive definite, then $\mathbf{x}^*$ is a minimum of f
- Negative definite, then $\mathbf{x}^*$ is a maximum of f
- Indefinite, then $\mathbf{x}^*$ is a saddle point of f
- Singular, then various pathological situations can occur

This is called the **second-order sufficient condition**.

6.2 Optimization in 1D

A function $f : \mathbb{R} \to \mathbb{R}$ is **unimodal** on an interval $[a, b]$ if there is a unique $x^* \in [a, b]$ such that $f(x^*)$ is the minimum value of f on $[a, b]$, and for any $x_1, x_2 \in [a, b]$ with $x_1 < x_2$,

$x_2 < x^*$ implies $f(x_1) > f(x_2)$ and $x_1 > x^*$ implies $f(x_1) < f(x_2)$.

Thus, $f(x)$ is strictly decreasing for $x \leq x^*$ and strictly increasing for $x \geq x^*$.
This property will allow us to refine an interval containing a solution by computing sample values of the function within the interval and discarding portions of the interval according to the function values obtained, analogous to bisection for solving nonlinear equations.

6.2.1 Golden Section Search

Suppose f is unimodal on $[a, b]$, and let $x_1, x_2 \in [a, b]$ with $x_1 < x_2$.
By comparing the function values $f(x_1)$ and $f(x_2)$ we can exclude a subinterval, either $(x_2, b]$ or $[a, x_1)$ because we know that the minimum lies within the remaining subinterval.

To make consistent progress in reducing the length of the interval containing the minimum, each pair of points in the new interval should have the same relative position as the old pair in the old interval.

To accomplish this objective, we choose the relative positions of the two points to be τ and $1 - \tau$, where $\tau^2 = 1 - \tau$, so that $\tau = (\sqrt{(5)} - 1)/2 \approx 0.618$ (the "golden ratio") and $1 - \tau \approx 0.382$. The complete procedure converges linearly to a local minimum *if* the function is unimodal within the initial bracket.

The golden section search for optimization is analogous to the bisection method to solve a nonlinear equation. Similarly, we do not make use of the function values, other than to compare them.

6.2.2 Successive parabolic interpolation

The simplest approach of fitting a second order polynomial to the problem is **successive parabolic interpolation**, where the function is evaluated at three points, and a parabola is fitted to the resulting function values.
The minimum of the parabola is used as a new approximate value of the minimum.

This algorithm is not guaranteed to converge, but if it is started reasonable close to a minimum it converges superlinearly with convergence rate $r \approx 1.324$

6.2.3 Newton's method

Instead of fitting a parabola to the points, we can also obtain a local quadratic approximation based on a truncated Taylor expansion.

$$f(x+h) \approx f(x) + f'(x)h + \frac{1}{2}f''(x)h^2$$

The minimum of this function is given by $-f'(x)/f''(x)$, which we can use to find the minimum of the objective function in an iterative way.

This method is equivalent to Newton's method for solving nonlinear equations, and also has a quadratic convergence rate.
However, unless it is started sufficiently close to the desired minimum it might not converge at all, or converge to a maximum or inflection point instead.

6.3 Unconstrained optimization in ND

6.3.1 Direct Search

Analogous to the golden section search for one-dimensional optimization, in direct search methods for multidimensional optimization the objective function values are only *compared* to each other.
However, in contrast to the golden section search, they do not retain the convergence guarantee.

For example: **Nelder and Mead**.
To seek the minimum of a function $f : \mathbb{R}^n \to \mathbb{R}$, the function is first evaluated at $n+1$ starting points.
These $n+1$ starting points form a *simplex* meaning that no three points are colinear (e.g. a simplex in two dimensions, has three points which form a triangle).
A new point is generated along the straight line connecting the point with the highest function

value (the *worst* point) and the centroid of the remaining n points.
This new point then replaces the worst point and the process is repeated until convergence.

Direct search methods are especially useful for nonsmooth objective functions, for which few other methods are applicable, and they can be effective when n is small, but they tend to be quite expensive when n is larger than two or three.
One advantage of direct search methods is that they can easily be parallelized.

6.3.2 Steepest Descent

The negative gradient of a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ points *downhill* and locally, $-\nabla f(\mathbf{x})$ is the direction of *steepest descent*.
Thus, the negative gradient is a potentially fruitful direction in which to seek points having lower function values.

The maximum possible benefit from movement in any downhill direction is to attain the minimum of the objective function along that direction.
For any fixed $\mathbf{x}$ and direction $\mathbf{s} = -\nabla f(\mathbf{x})$, we can define a function $\phi : \mathbb{R}^n \to \mathbb{R}$:

$$\phi(\alpha) = f(\mathbf{x} + \alpha\mathbf{s})$$

In this way the problem of minimizing the objective function f along the direction of $\mathbf{s}$ from $\mathbf{x}$ is seen to be a one-dimensional optimization problem.
Once a minimum is found in a certain direction, the negative gradient is computed at this new point and the process is repeated until convergence.
This process of minimizing an objective function only along a fixed line in $\mathbb{R}^n$ is called a *line search*.

The steepest descent method is very reliable in that it can always make progress provided the gradient is nonzero. However, the resulting iterations can zigzag back and forth, making very slow progress.
In general the convergence rate of steepest descent is only linear.

6.3.3 Newton's Method

Again we make a local quadratic approximation, which can be obtained from a truncated Taylor series expansion

$$f(\mathbf{x} + \mathbf{s}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x})^T\mathbf{s} + \frac{1}{2}\mathbf{s}^T\mathbf{H}_f(\mathbf{x})\mathbf{s}$$

where $\mathbf{H}_f(\mathbf{x})$ is the *Hessian matrix*.
This quadratic function in $\mathbf{s}$ is minimized when

$$\mathbf{H}_f(\mathbf{x})\mathbf{s} = -\nabla f(\mathbf{x}).$$

The convergence rate of Newton's method for unconstrained optimization is normally quadratic but the method is unreliable unless started close enough to the solution.

While Newton's method does not require a line search, it may still be advisable to perform a line search along the direction of the Newton step in order to make the method more robust.

Newton's method usually converges very rapidly once it nears a solution, but it requires a substantial amount of work per iteration.
Specifically, for a problem with a dense Hessian matrix, each iteration requires $\mathcal{O}(n^2)$ scalar function evaluations to form the gradient and the Hessian matrix while $\mathcal{O}(n^3)$ arithmetic operations are required to solve the linear system for the Newton step. s

6.3.4 Quasi-Newton Methods

Many variants of Newton's method have been developed to reduce its overhead or improve its reliability, or both. These *quasi-Newton methods* have the general form

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{B}_k^{-1} \nabla f(\mathbf{x}_k)$$

where α_k is a line search parameter and $\mathbf{B}_k$ is some approximation of the Hessian matrix obtained in any number of ways, including secant updating, finite differences, periodic reevaluation, or neglecting some terms in the true Hessian of the objective function.

Many quasi-Newton methods are more robust than the pure Newton method and have considerably lower overhead per iteration yet remain superlinear (though not quadratic).

6.3.5 Secant updating Methods

Several secant updating formulas for unconstrained minimization have been developed that not only preserve symmetry in the approximate Hessian matrix but also preserve positive definiteness.
Symmetry reduces the amount of work and storage required by about half, and positive definiteness guarantees that the resulting quasi-Newton step will be a descent direction.
In practice, a factorization of $\mathbf{B}_k$ is updated rather than $\mathbf{B}_k$ itself, so that the linear system for the quasi-Newton step can be solved at a cost per iteration of $\mathcal{O}(n^2)$ rather than $\mathcal{O}(n^3)$ operations.

Unlike Newton's method for optimization, no second derivatives are required.
And most of these methods are often started with $\mathbf{B}_0 = \mathbf{I}$, which means the first step is along the negative gradient (i.e. along the direction of steepest descent) and then second derivative information is gradually built up in the approximate Hessian matrix by updating over successive iterations.

6.3.6 Conjugate Gradient method

The conjugate gradient method is another alternative to Newton's method that does not require explicit second derivatives.
Indeed, unlike secant updating methods, the conjugate gradient method does not even store

an approximation to the Hessian matrix, which makes it especially **suitable for very large problems**.

As the name suggests, the conjugate gradient method also uses gradients, but in contrast to the steepest descent method it avoids repeatedly searching in the same directions by modifying the new gradient at each step to remove components in previous directions.
The resulting sequence of *conjugate* (i.e. orthogonal in the inner product $(\mathbf{x}, \mathbf{y}) = \mathbf{x}^T \mathbf{H}_f \mathbf{y}$) search directions implicitly accumulates information about the Hessian matrix as iterations proceed.

Theoretically, the conjugate gradient method is exact after at most n iterations for a quadratic objective function in n dimensions, but it is usually quite effective for more general unconstrained optimization problems as well.
It is common to restart the algorithm after every n iterations by restarting to use the negative gradient at the current point.

6.4 Nonlinear Least Squares

Least squares data fitting can be viewed as an optimization problem.
Given data points $(t_i, y_i), i = 1, \ldots, m$, we wish to find the vector $\mathbf{x} \in \mathbb{R}^n$ of parameters that gives the best fit to the model function $f(t, \mathbf{x})$, where $f : \mathbb{R}^{n+1} \to \mathbb{R}$.
Previously, we only considered cases in which the model function f was linear in the components of $\mathbf{x}$ but now we are in a position to consider *nonlinear least squares* as a special case of nonlinear optimization.

If we define the components of the *residual* function $\mathbf{r} : \mathbb{R}^n \to \mathbb{R}^m$ by

$$r_i(\mathbf{x}) = y_i - f(t_i, \mathbf{x}), \quad i = 1, \ldots, m,$$

then we wish to minimize the function

$$\phi(\mathbf{x}) = \frac{1}{2} \mathbf{r}(\mathbf{x})^T \mathbf{r}(\mathbf{x})$$

i.e. the sum of squares of the residual components (the factor 1/2 is inserted for later convenience and has no effect on the optimal value for $\mathbf{x}$).
If we apply Newton's method and $\mathbf{x}_k$ is an approximate solution, then the Newton step $\mathbf{s}_k$ is given by the linear system

$$\mathbf{H}_\phi(\mathbf{x}_k) \mathbf{s}_k = -\nabla \phi(\mathbf{x}_k)$$

where the gradient vector and Hessian matrix of ϕ are given by

$$\nabla \phi(\mathbf{x}) = \mathbf{J}^T(\mathbf{x}) \mathbf{r}(\mathbf{x})$$

and

$$\mathbf{H}_\phi(\mathbf{x}) = \mathbf{J}^T(\mathbf{x}) \mathbf{J}(\mathbf{x}) + \sum_{i=1}^{m} r_i(\mathbf{x}) \mathbf{H}_{r_i}(\mathbf{x})$$

in which $\mathbf{J}^T(\mathbf{x})$ is the Jacobian matrix of $\mathbf{r}(\mathbf{x})$, and $\mathbf{H}_{r_i}(\mathbf{x})$ denotes the Hessian matrix of the component function $r_i(\mathbf{x})$.

The Newton step $\mathbf{s}_k$ is thus given by the linear system

$$\left(\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x_k}) + \sum_{i=1}^{m} r_i(\mathbf{x}_k)\mathbf{H}_{r_i}(\mathbf{x}_k)\right)\mathbf{s}_k = -\mathbf{J}^T(\mathbf{x}_k)\mathbf{r}(\mathbf{x}_k)$$

The m Hessian matrices $\mathbf{H}_{r_i}$ of the residual components are usually inconvenient and expensive to compute. Fortunately, we can exploit the special structure of this problem to avoid computing them in most cases, as we will see next.

6.4.1 Gauss-Newton Method

Note that in $\mathbf{H}_\phi$ each of the Hessian matrices $\mathbf{H}_{r_i}$ is multiplied by the corresponding residual component r_i, which should be small at a solution, provided that the model function fits the data reasonably well. This observation motivates the *Gauss-Newton method* for nonlinear least squares in which the terms involving $\mathbf{H}_{r_i}$ are dropped from the Hessian and the linear system

$$\big(\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x_k})\big)\mathbf{s}_k = -\mathbf{J}^T(\mathbf{x}_k)\mathbf{r}(\mathbf{x}_k)$$

determines an approximate Newton step $\mathbf{s}_k$ at each iteration.

We recognize this system as the normal equations for the $m \times n$ linear least squares problem

$$\mathbf{J}(\mathbf{x}_k)\mathbf{s}_k \cong -\mathbf{r}(\mathbf{x}_k)$$

which can be solved more reliably by orthogonal factorization of $\mathbf{J}(\mathbf{x}_k)$.
The next approximate solution is then given by $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$ and the process is repeated until convergence.
In effect, the Gauss-Newton method replaces a nonlinear least squares problem by a sequence of linear least squares problems whose solutions converge to the solution of the original nonlinear problem.

If the residual components at the solution are relatively large, then the terms omitted from the Hessian matrix may not be negligible, in which case the Gauss-Newton approximation may be inaccurate and convergence is no longer guaranteed.
In such cases, it may be best to use a general nonlinear optimization method that takes into account the full Hessian matrix.

6.4.2 Levenberg-Marquardt Method

The *Levenberg-Marquardt method* is a useful alternative when the Gauss-Newton method yields an ill-conditioned or rank-deficient linear least squares subproblem.
At each iteration of this method, the linear system for the step $\mathbf{s}_k$ is of the form

$$\big(\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x_k}) + \mu_k\mathbf{I}\big)\mathbf{s}_k = -\mathbf{J}^T(\mathbf{x}_k)\mathbf{r}(\mathbf{x}_k)$$

where μ_k is a nonnegative scalar parameter chosen by some strategy. The corresponding linear least squares problem to be solved is

$$\begin{bmatrix} \mathbf{J}(\mathbf{x}_k) \\ \sqrt{\mu_k}\,\mathbf{I} \end{bmatrix} \mathbf{s}_k \cong \begin{bmatrix} -\mathbf{r}(\mathbf{x}_k) \\ \mathbf{0} \end{bmatrix}$$

This method can be interpreted as replacing the terms omitted from the true Hessian by a scalar multiple of the identity matrix or as using a weighted combination of the Gauss-Newton step and the steepest descent direction. With a suitable strategy for choosing the parameter μ_k, typically based on a trust-region approach, the Levenberg-Marquardt method can be very robust in practice, and it forms the basis for several effective software packages for solving nonlinear least squares problems.

6.5 Constrained Optimization

- **Equality constraints** which are of the general form $\mathbf{g}(\mathbf{x}) = \mathbf{0}$
- **Inequality constraints** which are of the general form $\mathbf{h}(\mathbf{x}) \leq \mathbf{0}$

Inequality constraints may be irrelevant to the solution and a given inequality constraint $h_i(\mathbf{x}) \leq 0$ is said to be *active* or *binding* at a feasible point $\mathbf{x} \in \mathbf{S}$ if $h_i(\mathbf{x}) = 0$. Naturally, equality constraints are always active.

Both equality-constrained and inequality-constrained problems can be solved using *Lagrange multipliers* although the optimality conditions become more complicated when inequality constraints are involved.

The minimize function of `scipy.optimize` provides several algorithms for constrained minimization.

6.6 Summary

- Optimization in 1D
    - Golden section search
        - The procedure converges linearly to a local minimum *if* the function is unimodal within the initial bracket.
          -The golden section search for optimization is analogous to the bisection method to solve a nonlinear equation.
        - We do not make use of the function values, other than to compare them.
    - Succesive parabolic interpolation
        - This algorithm is not guaranteed to converge, but if it is started reasonable close to a minimum it converges superlinearly with convergence rate $r \approx 1.324$
    - Newton's method

        - This method is equivalent to Newton's method for solving nonlinear equations, and also has a quadratic convergence rate.
        - unless it is started sufficiently close to the desired minimum it might not converge at all, or converge to a maximum or inflection point instead.

- Unconstrained optimization in ND

    - Direct search

        - Direct search methods are especially useful for nonsmooth objective functions, for which few other methods are applicable, and they can be effective when n is small, but they tend to be quite expensive when n is larger than two or three.
        - One advantage of direct search methods is that they can easily be parallelized.

    - Steepest descent

        - The steepest descent method is very reliable in that it can always make progress provided the gradient is nonzero. However, the resulting iterations can zigzag back and forth, making very slow progress.
        - In general the convergence rate of steepest descent is only linear.

    - Newton's method

        - The convergence rate of Newton's method for unconstrained optimization is normally quadratic but the method is unreliable unless started close enough to the solution.
        - Newton's method usually converges very rapidly once it nears a solution, but it requires a substantial amount of work per iteration.
        - for a problem with a dense Hessian matrix, each iteration requires $\mathcal{O}(n^2)$ scalar function evaluations to form the gradient and the Hessian matrix while $\mathcal{O}(n^3)$ arithmetic operations are required to solve the linear system for the Newton step. s

    - Quasi-Newton methods

        - Many quasi-Newton methods are more robust than the pure Newton method and have considerably lower overhead per iteration yet remain superlinear (though not quadratic).

    - Secant updating methods

        - preserve symmetry in the approximate Hessian matrix but also preserve positive definiteness.
        - Symmetry reduces the amount of work and storage required by about half, and positive definiteness guarantees that the resulting quasi-Newton step will be a descent direction.

        - Unlike Newton's method for optimization, no second derivatives are required.
    - Conjugate gradient method
        - does not even store an approximation to the Hessian matrix, which makes it especially **suitable for very large problems**.
        - Theoretically, the conjugate gradient method is exact after at most n iterations for a quadratic objective function in n dimensions, but it is usually quite effective for more general unconstrained optimization problems as well.
- Nonlinear Least squares
    - Gauss-Newton method
        - In effect, the Gauss-Newton method replaces a nonlinear least squares problem by a sequence of linear least squares problems whose solutions converge to the solution of the original nonlinear problem.
    - Levenberg-Marquardt method
        - the Levenberg-Marquardt method can be very robust in practice, and it forms the basis for several effective software packages for solving nonlinear least squares problems
- Constrained optimization
    - In general can be solved using Lagrange multipliers.
    - Efficient algorithms are in scipy.

7 Interpolation

7.1 Introduction

There are many different purposes for which you might want to use interpolation:

- Plotting a smooth curve through discrete data points
- Reading between the lines of a table
- Differentiating or integrating tabular data
- Evaluating a mathematical function quickly and easily
- Replacing a complicated function by a simple one

For given data
$(t_i, y_i), i = 1, \ldots, m,$

with $t_1 < t_2 < \ldots < t_m$, we seek a function $f : \mathbb{R} \to \mathbb{R}$ such that

$f(t_i) = y_i, i = 1, \ldots, m,$

For a given set of data points $(t_i, y_i), i = 1, \ldots, m$,, an interpolant is chosen from the space of functions spanned by a suitable set of **basis functions** $\phi_1(t), \ldots, \phi_n(t)$.

The interpolating function f is therefore expressed as a linear combination of these basis functions

$$f(t) = \sum_{j=1}^{n} x_j \phi_j(t)$$

where the parameters x_j are to be determined. Requiring that f interpolate the data (t_i, y_i) means that

$$f(t) = \sum_{j=1}^{n} x_j \phi_j(t_i) = y_i$$

which is a system of linear equations that we can write in matrix form as
$\mathbf{A}\mathbf{x} = \mathbf{y}$,
where the entries of the **basis matrix** $\mathbf{A}$ are given by $a_{ij} = \phi_j(t_i)$, the components of the right-hand-side vector $\mathbf{y}$ are the known data points y_i, and the components of the vector $\mathbf{x}$ the unknown parameters x_j we want to determine.

7.2 Polynomial Interpolation of discrete data

7.2.1 Monomial basis

To interpolate n data points, we choose $k = n - 1$ so that the dimension of the space will match the number of data points. An obvious basis for $\mathbb{P}_{n-1}$ is given by the first n **monomials**

$$\phi_j(t) = t^{j-1}$$

for which a given polynomial $p_{n-1} \in \mathbb{P}_{n-1}$ has the form

$$p_{n-1}(t) = x_1 + x_2 t + \cdots + x_n t^{n-1}$$

The system of equations we want to solve is

$$\mathbf{Ax} = \begin{bmatrix} 1 & t_1 & t_1^2 & \cdots & t_1^{n-1} \\ 1 & t_2 & t_2^2 & \cdots & t_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & t_n & t_n^2 & \cdots & t_n^{n-1} \end{bmatrix} \begin{bmatrix} x1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \mathbf{y}$$

Such a matrix, whose columns are subsequent powers of a variable is called a **Vandermonde matrix**.

Although it is not singular, it is often *nearly singular* because the functions become increasingly difficult to distinguish as the degrees increase (see figure below). This makes the the columns of the Vandermonde matrix almost linearly dependent.

In addition to the computational cost of determining the interpolating polynomial, the cost of evaluating it is also an important factor!

When represented in the monomial basis, a polynomial

$$p_{n-1}(t) = x_1 + x_2 t + x_3 t^2 + \cdots + x_n t^{n-1}$$

can be evaluated very efficiently using **Horner's method** also known as **Nested evaluation** or **synthetic division**:

$$p_{n-1}(t) = x_1 + t(x_2 + t(x_3 + t(\cdots(x_{n-1} + x_n t)\cdots)))$$

which requires only n summations and n additions.

7.2.2 Lagrange Interpolation

For a given set of data points $(t_i, y_i), i = 1, \ldots, m$,
the **Langrange basis functions** for $\mathbb{P}_{n-1}$ are given by

$$l_j(t) = \frac{\prod_{k=1, k\neq j}^{n}(t - t_k)}{\prod_{k=1, k\neq j}^{n}(t_j - t_k)}$$

It can be seen that

- $$l_j(t) \text{ is a polynomial of degree } n-1$$
- $$l_j(t_i) = \begin{cases} 1, & \text{if} \quad i = j. \\ 0, & \text{if} \quad i \neq j. \end{cases}$$

which means that for this basis the matrix of the linear system $\mathbf{Ax} = \mathbf{y}$ is the identity matrix $\mathbf{I}$.
The interpolating polynomial then is

$$p_{n-1}(t) = y_1 l_1(t) + y_2 l_2(t) + \cdots + y_n l_n(t)$$

which is easy to construct.

6.5.1 Newton Interpolation

For a given set of data points $(t_i, y_i), i = 1, \ldots, m$,
the **Newton basis functions** for $\mathbb{P}_{n-1}$ are given by

$$\pi_j(t) = \prod_{k=1}^{j-1} (t - t_k)$$

Note that we assign $\pi_j(t) = 1$ for $j = 1$

The interpolating polynomial then has the form

$$p_{n-1}(t) = x_1 + x_2(t - t_1) + x_3(t - t_1)(t - t_2) + \cdots x_n(t - t_1) \ldots (t - t_{n-1})$$

From the definition it can be seen that the basis matrix $\mathbf{A}$ is lower triangular, so the system $\mathbf{Ax} = \mathbf{y}$ can efficiently be solved by forward substitution.

You can also use Horner's method to evaluate the polynomial:

$$p_{n-1}(t) = x_1 + (t - t_1)\left[x_2 + (t - t_2)\left[x_3 + (t - t_3)\left[\cdots (x_{n-1} + x_n(t - t_{n-1}))\cdots\right]\right]\right]$$

Another useful property of the newton basis functions is that the interpolant can be constructed *incrementally* as more data points are added.

If $p_j(t)$ is a polynomial of degree $j - 1$ which interpolates j data points, then for any constant x_{j+1}

$$p_{j+1}(t) = p_j(t) + x_{j+1}\pi_{j+1}(t)$$

is a polynomial of degree j that also interpolates the same j points. The free parameter x_{j+1} can be
chosen so that $p_{j+1}(t)$ interpolates the new data points y_{j+1} as

$$x_{j+1} = \frac{y_{j+1} - p_j(t_{j+1})}{\pi_{j+1}(t_{j+1})}$$

6.6 Polynomial interpolation of a continuous function

Interpolants of a high degree

- can be expensive to determine or evaluate (depending on the basis chosen)
- by definition a polynomial of degree n has $n - 1$ extrema, and thus many "wiggles"
  Even if the polynomial passes through all required data points, it may fluctuate wildly in between these points and does not approximate an underlying function at all.

Polynomial interpolants of increasing degree converge to the function in the middle of the interval, but diverge near the endpoints. More satisfactory results can be obtained if our sample points are bunched near the ends of the interval.

One good way to achieve this is to use the **Chebysev points** which are defined on the interval $[-1, 1]$, but can transformed to an arbitrary interval.

$$t_i = \cos\left(\frac{(2i-1)\pi}{2k}\right), \qquad i = 1, \ldots, k$$

6.7 Piecewise Polynomial Interpolation

Consider a set of ordered data points (x_i, y_i) in the interval $[a, b]$. A function S is called a spline interpolation of degree k if

- S is defined on the interval $[a, b]$
- the rth derivative of S is continuous in the interval $[a, b]$ for $0 \leq r \leq k-1$
- S is a polynomial of degree $\leq k$ in every subinterval between the knots

The most used form of spline interpolation is *Cubic interpolation*.

A spline with n knots has $(n-1)$ piecewise polynomials of degree k that interpolate the data. The number of free parameters thus is $(k+1)(n-1)$.
For instance, a linear polynomial($k = 1$, and with 2 free parameters per polynomial) has $2(n-1)$ free parameters and a **cubic spline** has $4(n-1)$.

- Interpolating the data requires $2(n-1)$ equations, because each of the $(n-1)$ polynomials must match the two data points at either end of the subinterval.
- Requiring the derivative to be continuous gives $(n-2)$ additional equations, because there are $(n-2)$ *interior* data points.
- Requiring that the second derivative is also continuous gives $(n-2)$ additional equations.

The spline of lowest degree that has sufficient variables to satisfy all these equations is a cubic spline ($4n-4$ variables for $4n-6$ conditions). The remaining two variables can be fixed in a number of ways:

- **clamped cubic spline**: Specifying the first derivative of the endpoints
- **natural spline**: forcing the second derivative to be zero at the endpoints
- **not-a-knot**: the third derivative of the spline is continuous at the one-but-outermost data points
- **periodic spline**: forcing equality of the first as well as second derivatives of the two outermost points (if the spline is to be periodic)

6.8 Summary

Interpolation of discrete data

- Monomial Interpolation
    - System of equations is a Vandermonde matrix (which is often nearly singular: difficult to distinguisch large powers).
    - Polynomial can be evaluated quickly using Horner's method.
- Lagrange Interpolation
    - System of equations is the identity matrix.
    - Polynomial is easy to construct.
- Newton Interpolation
    - System of equations is (lower) triangular -> forward substitution.
    - Polynomial can be evaluated quickly using a (modified) Horner's method.
    - Interpolant can be constructed *incrementally* as more data points are added.

Interpolation of a continuous function

- can be expensive to determine or evaluate (depending on the basis chosen).
- by definition a polynomial of degree n has $n-1$ extrema, and thus many "wiggles".
- Polynomial interpolants of increasing degree converge to the function in the middle of the interval, but diverge near the endpoints.
- Use the *Chebysev points* to concentrate the sampling near the edges of the interval.

Piecewise interpolation

- The most used form of spline interpolation is *Cubic interpolation*.
- Interpolating the data requires $2(n-1)$ equations, because each of the $(n-1)$ polynomials must match the two data points at both ends of the subinterval.
- Requiring the derivative to be continuous gives $(n-2)$ additional equations, because there are $(n-2)$ *interior* data points.
- Requiring that the second derivative is also continuous gives $(n-2)$ additional equations.
- The spline of lowest degree that has sufficient variables to satisfy all equations is a cubic spline ($4n-4$ variables for $4n-6$ conditions). The remaining two variables can be fixed in a number of ways.

7 Integration and Differentiation

7.1 Integration

Riemann sums are in theirselves a way to approximate an integral, but there are better numerical methods.

7.2 Existence, Uniqueness, Conditioning

- For all practical purposes, a function is integrable if it is bounded (no singularities) with at most a finite number of points of discontinuity within the interval of integration.
- Since all the Riemann sums defining the Riemann integral of a given function on a given interval must have the same limit, uniqueness of the Riemann integral is built into its definition.
- Because integration is an averaging or smoothing process that tends to dampen the effect of small changes in the integrand, integration problems are typically well-behaved with a small condition number.

7.3 Numerical Quadrature

Essentially quadrature is interpolation:

- The integrand function f is evaluated at the points x_i, $i = 1, \ldots, n$.
- The polynomial of degree $n-1$ that interpolates the function values at those points is determined.
- The integral of the interpolant is then taken as an approximation to the integral of the original function.

To find the weights that integrate the first n polynomials exactly, we can use the method of undetermined coefficients:

$$w_1 \cdot 1 + w_2 \cdot 1 + \cdots + w_n \cdot 1 = \int_a^b 1 dx = b - a$$

$$w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_n \cdot x_n = \int_a^b x dx = (b^2 - a^2)/2$$

$$\vdots$$

$$w_1 \cdot x_1^{n-1} + w_2 \cdot x_2^{n-1} + \cdots + w_n \cdot x_n^{n-1} = \int_a^b x^{n-1} dx = (b^n - a^n)/n$$

In matrix form this becomes:

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_n \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{n-1} & x_2^{n-1} & \cdots & x_n^{n-1} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} b-a \\ (b^2-a^2)/2 \\ \vdots \\ (b^n-a^n)/n \end{bmatrix}$$

7.3.1 Accuracy and other concepts

By construction, an n-point interpolatory quadrature rule integrates each of the first $n-1$ monomial basis functions exactly, and hence by linearity it integrates any polynomial of degree at most $n-1$ exactly.

A quadrature rule is said to be of **degree** d if it is exact (i.e., the error is zero) for every polynomial of degree d but is not exact for some polynomial of degree $d+1$.

An n-point interpolatory quadrature rule is of degree at least $n-1$.

The significance of the degree is that it conveniently characterizes the **accuracy** of a given rule. If Q_n is an interpolatory quadrature rule, and p_{n-1} is the polynomial of degree at most $n-1$ interpolating an integrand f at the nodes $x_1, \cdots, x_n$, then we get the following error bound for the approximate integral:

$$\|I(f) - Q_n(f)\| \leq \frac{1}{4} h^{n+1} \|f^{(n)}\|_\infty$$

where $h = \max\{x_{i+1} - x_i : i = 1, \ldots, n-1\}$.

the preceding general bound already indicates that we can obtain higher accuracy by taking n larger, or h smaller, or both.

A sequence of quadrature rules is said to be **progressive** if the nodes of Q_{n1} are a subset of those of Q_{n2} for $n_2 > n_1$, so that they don't need to be re-evaluated.

There are **simple quadrature rules**, in which a single rule is applied over the entire given interval.
There are also **composite (or compound) quadrature rules**, which is equivalent to using piecewise polynomial interpolation on the original interval (using different quadrature rules on each subinterval).

7.3.2 Newton-Cotes Quadrature

An n-point **open Newton-Cotes rule** has nodes

$$x_i = a + i(b-a)/(n+1) \quad , \quad i = 1, \cdots, n$$

and an n-point **closed Newton-Cotes rule** has nodes

$$x_i = a + (i-1)(b-a)/(n-1) \quad , \quad i = 1, \cdots, n$$

7.3.2.1 The midpoint rule

Interpolating the function value at the midpoint of the interval by a polynomial of degree zero (i.e., a constant) gives the one-point open Newton-Cotes rule known as **the midpoint rule**:

$$M(f) = (b-a) f\left(\frac{a+b}{2}\right)$$

7.3.2.2 The trapezoid rule

Interpolating the function values at the two endpoints of the interval by a polynomial of degree one (i.e., a straight line) gives the two-point closed Newton-Cotes rule known as **the trapezoid rule**:

$$T(f) = \frac{b-a}{2}(f(a) + f(b))$$

7.3.2.3 Simpson's rule

Interpolating the function values at the two endpoints and the midpoint by a polynomial of degree two (i.e., a quadratic) gives the three-point closed Newton-Cotes rule known as **Simpson's rule**:

$$S(f) = \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)$$

7.3.2.4 Error Analysis

The error in the midpoint quadrature rule can be estimated using a Taylor series expansion about the midpoint $m = (a+b)/2$ of the interval $[a, b]$:

$$f(x) = f(m) + f'(m)(x-m) + \frac{f''(m)}{2}(x-m)^2 + \frac{f'''(m)}{6}(x-m)^3 + \frac{f^{(4)}(m)}{24}(x-m)^4 + \cdots$$

Integrating this expression from a to b, the odd-order terms drop out, yielding

$$I(f) = f(m)(b-a) + \frac{f''(m)}{24}(b-a)^3 + \frac{f^{(4)}(m)}{1920}(b-a)^5 + \cdots = M(f) + E(f) + F(f)$$

where $E(f)$ and $F(f)$ represent the first two terms in the error expansion for the midpoint rule.

For the trapezoid rule, take $x = a$ and $x = b$ into the Taylor series, and simplify.

$$I(f) = T(f) - 2E(f) - 4F(f) - \cdots$$

$$T(f) - M(f) = 3E(f) + 5F(f) + \cdots$$

$$E(f) \approx \frac{T(f) - M(f)}{3}$$

1. The midpoint rule is about twice as accurate as the trapezoid rule, despite being based on a polynomial interpolant of degree one less.

2. The difference between the midpoint rule and the trapezoid rule can be used to estimate the error in either of them

3. Halving the length of the interval decreases the error in either rule by a factor of about 1/8

For Simpson's rule, an appropriately weighted combination of the midpoint and trapezoid rules eliminates the leading term, $E(f)$, from the error expansion,

$$I(f) = \frac{2}{3}M(f) + \frac{1}{3}T(f) - \frac{2}{3}F(f) + \cdots = S(f) - \frac{2}{3}F(f) + \cdots$$

In general, for any odd value of n, an n-point Newton-Cotes rule has degree one greater than that of the polynomial interpolant on which it is based due to cancellation of positive and negative errors.

The midpoint rule integrates linear polynomials exactly (degree 1). the error for Simpson's rule depends on the fourth and higher derivatives in the Taylor expansion, which vanish for cubic as well as quadratic polynomials, so that Simpson's rule is of degree three rather than two.

7.3.2.5 Drawbacks

- The interpolation of a continuous function at equally spaced points by a high-degree polynomial may suffer from unwanted oscillation, and as the number of interpolation points grows, convergence to the underlying function is not guaranteed.
- it can be shown that every n-point Newton-Cotes rule with $n \leq 11$ has at least one negative weight and that the sum of all weights tends to infinity as $n \to \infty$. Thus they are ill-onditioned
- The presence of large positive and negative weights also means that the value of the integral is computed as a sum of large quantities of differing sign, and hence substantial cancellation is likely in finite-precision arithmetic.

We can't use Newton-Cotes with large subdivions:
In practice, therefore, Newton-Cotes rules are usually restricted to a modest number of points, and if higher accuracy is required, then the interval is subdivided and composite quadrature is used.

7.3.3 Clenshaw-Curtis Quadrature

Using the Chebysev points, we can achieve higher accuracy.
Efficient implementations of quadrature rules based on the Chebyshev points using FFT have become known as **Clenshaw-Curtis quadrature**.

It can be shown that the resulting weights are always positive for any n, and that the resulting approximate values converge to the exact integral as $n \to \infty$.
Thus, quadrature rules based on the Chebyshev points are extremely attractive in that they are always stable and more accurate then N-C for the same number of nodes.

Nevertheless, the degree of an n-point rule is only $n-1$, which is well below the maximum possible.

7.3.4 Gaussian Quadrature

In Gaussian quadrature, both the nodes and the weights are optimally chosen to maximize the degree of the resulting quadrature rule.

In general, for each n there is a unique n-point Gaussian rule, and it is of degree $2n-1$.
Gaussian quadrature rules therefore have the highest possible accuracy for the number of nodes used, but they are significantly more difficult to derive than Newton-Cotes rules.
The nodes and weights can still be determined by the method of undetermined coefficients, but the resulting system of equations is nonlinear.

- For any n the Gaussian **nodes are symmetrically** placed about the midpoint of the interval; for odd values of n the midpoint itself is always a node.

- **nodes are usually irrational numbers** even when the endpoints a and b are rational.
  This feature makes Gaussian rules relatively inconvenient for hand computation, compared with simple Newton-
  Cotes rules. Therefore use tabulated values (but's let's be real, calculating integrals by hand in 2021?)

- Gaussian quadrature weights and nodes are derived for some specific interval and thus any other interval of integration must be transformed into
  the standard interval for which the nodes and weights have been tabulated.

$$\int_{\alpha}^{\beta} f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

  using

$$t = \frac{(b-a)x + a\beta - b\alpha}{\beta - \alpha}$$

$$I(g) \approx \frac{b-a}{\beta-\alpha} \sum_{i=1}^{n} w_i g\left(\frac{(b-a)x_i + a\beta - b\alpha}{\beta - \alpha}\right)$$

I t can be shown that the resulting weights are always positive for any n, so that Gaussian quadrature rules are always stable and the resulting approximate values converge to the exact integral as $n \to \infty$.

for $m \neq n$, G_m and G_n have no nodes in common (except for the midpoint when m and n are both odd). Thus, Gaussian rules are not progressive.

Avoiding this additional work is the motivation for Kronrod quadrature rules.
Such rules come in pairs: an n-point Gaussian rule G_n and a $(2n+1)$-point Kronrod rule K_{2n+1} whose nodes are optimally chosen subject to the constraint that all of the nodes of G_n are

reused in K_{2n+1}.
Thus, n of the nodes used in K_{2n+1} are prespecified, leaving the remaining $n+1$ nodes, as well as all $2n+1$ of the weights (including those corresponding to the nodes of G_n), free to be chosen to maximize the degree of the resulting rule.
The rule K_{2n+1} is therefore of degree $3n+1$, whereas a true $(2n+1)$-point Gaussian rule would be of degree $4n+1$.
Thus, there is a tradeoff between accuracy and efficiency.

In using a Gauss-Kronrod pair, the value of K_{2n+1} is taken as the approximation to the integral, and a realistic but conservative estimate for the error, based partly on theory and partly on experience, is given by

$$(200\|G_N - K_{2n+1}\|)^{1.5}$$

Because they efficiently provide both high accuracy and a reliable error estimate, Gauss-Kronrod rules are among the most effective quadrature methods available. The pair of rules (G_7 , K_{15}), in particular, has become a commonly used standard.

7.3.5 Composite Quadrature

This approach is equivalent to using piecewise polynomial interpolation on the original interval and then integrating the piecewise interpolant to approximate the integral.

A **composite**, or **compound**, quadrature rule on a given interval $[a,b]$ results from subdividing the interval into k subintervals, typically of uniform length $h=(b-a)/k$, applying an n-point simple quadrature rule Q_n in each subinterval, and then taking the sum of these results as the approximate value of the integral.

If the rule Q_n is open, then evaluating the composite rule will require kn evaluations of the integrand function.
If Q_n is closed, on the other hand, then some of the points are repeated, so that only $k(n-1)+1$ evaluations of the integrand are required.

- In principle, by taking k sufficiently large it is possible to achieve arbitrarily high accuracy (up to the limit of the arithmetic precision) using a composite rule, even with an underlying rule Q_n of low degree, although this may not be the most efficient lway to attain a given evel of accuracy.

- Composite quadrature rules also offer a particularly simple means of estimating the error by using different levels of subdivision, which can easily be made progressive.

7.3.6 Adaptive Quadrature

A typical adaptive quadrature strategy works as follows:

1. First we need a pair of quadrature rules, say Q_{n1} and Q_{n2}, whose difference provides an error estimate.

A few examples are

- The trapezoid and midpoint rules, whose difference overestimates the error in the more accurate rule by a factor of three (see above).
- Greater efficiency is usually obtained with rules of higher degree, however, such as the Gauss-Kronrod pair (G_7 ,K_{15}).
- Another alternative is to use a single rule at two different levels of subdivision; Simpson's rule is a popular choice in this approach.

  The pair of rules should be progressive.

2. apply both rules Q_{n1} and Q_{n2} on the initial interval of integration $[a, b]$.
3. If the resulting approximate values for the integral differ by more than the desired tolerance, divide the interval into two or more subintervals and repeat the procedure on each subinterval.
4. If the tolerance is met on a given subinterval, then no further subdivision of that subinterval will be required.
5. If the tolerance is not met on a given subinterval, then the subdivision process is repeated again, and so on until the tolerance is met on all subintervals.

Such a strategy leads to a nonuniform sampling of the integrand function that places many sample points in regions where the function is difficult to integrate and relatively few points where the function is easily integrated.

7.4 Other integration problems

7.4.1 Tabular data

If we only have limited points at which we know the value of $f(x)$. We can use piecewise cubic spline interpolation.

7.4.2 Improper integrals

- unbounded intervals:
  - Replace any infinite limit of integration by a finite value.
    Such a finite limit should be chosen carefully so that the omitted tail is negligible or its contribution to the integral can be estimated. But the remaining finite interval should not be so wide that an adaptive quadrature routine will be fooled into sampling the integrand badly.
  - Transform the variable of integration so that the new interval is finite. Typical transformations include $x = -\log t$ or $x = t/(1 - t)$.
    Care must be taken not to introduce singularities or other difficulties by such a transformation.

    - Use a quadrature rule, such as Gauss-Laguerre or Gauss-Hermite, that is designed for an unbounded interval.

- singularities

    - adaptive quadrature is not a good idea: Even if the routine is lucky enough to avoid evaluating the integrand at the singularity, an adaptive quadrature routine will generally be extremely inefficient for an integrand having a singularity because polynomials, which never have vertical asymptotes, cannot efficiently approximate functions that do (recall that our error bounds depend on higher derivatives of the integrand, which will inevitably be large near a singularity).

    - The better solution is to remove the singularity analytically by a transformation, which is not always so trivial to find

7.4.3 Double integrals

- The only viable method is Monte Carlo integration (see infra).
- Quadrature rules become exponentially expensive.

7.5 Numerical differentiation

For a function that has discrete values, fit a polynomial and calculate it's derivative analytically.

For smooth functions, there are other techniques:

7.5.1 Finite difference approximations

These are based on the Taylor expansions

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2}h^2 + \frac{f'''(x)}{6}h^3 + \cdots$$

and

$$f(x-h) = f(x) - f'(x)h + \frac{f''(x)}{2}h^2 - \frac{f'''(x)}{6}h^3 + \cdots$$

7.5.1.1 First derivative

- Solving the first series for $f'(x)$, we obtain the **forward difference formula**

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{f''(x)}{2}h + \cdots \approx \frac{f(x+h) - f(x)}{h}$$

  This approximation is first-order accurate since the dominant remainder of the series is $\mathcal{O}(h)$.

- Similarly, we obtain the **backward difference formula** from the second series:

$$f'(x) = \frac{f(x) - f(x-h)}{h} - \frac{f''(x)}{2}h + \cdots \approx \frac{f(x) - f(x-h)}{h}$$

- Subtracting the second series from the first gives the **centered difference formula**

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{f'''(x)}{6}h^2 + \cdots \approx \frac{f(x+h) - f(x-h)}{2h}$$

  which is second-order accurate.

7.5.1.2 Second derivative

- Finally, adding the two series together gives a centered difference formula for the second derivative

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} - \frac{f^{(4)}(x)}{12}h^2 + \cdots \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

  which is also second-order accurate.

7.6 Richardson extrapolation

We make the value of the integral/derivative a function of the step size, and calculate its limiting value as we take the step size to zero.

Let $F(h)$ denote the value obtained with step size h, using a method for which we know the scaling behavior as $h \to 0$ (i.e. the order of the method).

Starting from

$$F(h) = a_0 + a_1 h^p + \mathcal{O}(h^r)$$

as $h \to 0$ for some p and r, with $r > p$.

Suppose that we have computed F for two step sizes, say, h and h/q for some positive integer q.
Then we have

$$F(h) = a_0 + a_1 h^p + \mathcal{O}(h^r)$$

and

$$F(h/q) = a_0 + a_1 (h/q)^p + \mathcal{O}(h^r) = a_0 + a_1 q^{-p} h^p + \mathcal{O}(h^r)$$

This system of two linear equations in the two unknowns a_0 and a_1 is easily solved to obtain

$$a_0 = F(h) + \frac{F(h) - F(h/q)}{q^{-p} - 1} + \mathcal{O}(h^r)$$

Thus, the accuracy of the improved value, a_0, is $\mathcal{O}(h^r)$ rather than $\mathcal{O}(h^p)$.

If $F(h)$ is known for several values of h, then the extrapolation process can be repeated to produce still more accurate approximations, up to the limitations imposed by finite-precision arithmetic.

7.6.1 Romberg integration

For any integer $k \leq 0$, let $T_{k,0}$ denote the approximation to the integral $\int_a^b f(x)dx$ given by the composite trapezoid rule with step size $h_k = (b-a)/2^k$.
Then for any integer j, $j = 1, \ldots, k$, we can recursively define the successive extrapolated values

$$Tk, j = \frac{4^j T_{k,j-1} - T_{k-1,j-1}}{4^j - 1}$$

which form a triangular array

$$\begin{matrix} T_{0,0} & & & & \\ T_{1,0} & T_{1,1} & & & \\ T_{2,0} & T_{2,1} & T_{2,2} & & \\ T_{3,0} & T_{3,1} & T_{3,2} & T_{3,3} & \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{matrix}$$

So given $T_{k,0}$, we can get better results by extrapolating.

7.7 Summary

- Newton-Cotes Quadrature
    - Midpoint rule, Trapezoid rule and Simpson's rule
    - In general, for any odd value of n, an n-point Newton-Cotes rule has degree one greater than that of the polynomial interpolant on which it is based due to cancellation of positive and negative errors.
    - The midpoint rule is about twice as accurate as the trapezoid rule, despite being based on a polynomial interpolant of degree one less, the difference between the midpoint rule and the trapezoid rule can be used to estimate the error in either of them
    - Halving the length of the interval decreases the error in either rule by a factor of about 1/8
    - Drawbacks:
        - The interpolation of a continuous function at equally spaced points by a high-degree polynomial may suffer from unwanted oscillation, and as the number of interpolation points grows, convergence to the underlying function is not guaranteed.
        - it can be shown that every n-point Newton-Cotes rule with $n \leq 11$ has at least one negative weight and that the sum of all weights tends to infinity as $n \to \infty$. Thus they are ill-onditioned

        - The presence of large positive and negative weights also means that the value of the integral is computed as a sum of large quantities of differing sign, and hence substantial cancellation is likely in finite-precision arithmetic.

- Clenshaw-Curtis Quadrature

    - The weights are always positive for any n, and that the resulting approximate values converge to the exact integral as $n \to \infty$.
    - The degree of an n-point rule is only $n - 1$, which is well below the maximum possible.

- Gaussian Quadrature

    - In general, for each n there is a unique n-point Gaussian rule, and it is of degree $2n - 1$.
    - Gaussian quadrature rules have the highest possible accuracy for the number of nodes used, but they are significantly more difficult to derive than Newton-Cotes rules.
    - For any n the Gaussian nodes are symmetrically placed about the midpoint of the interval; for odd values of n the midpoint itself is always a node.
    - nodes are usually irrational numbers even when the endpoints a and b are rational.
    - Gaussian quadrature weights and nodes are derived for some specific interval and thus any other interval of integration must be transformed into the standard interval for which the nodes and weights have been tabulated.
    - for $m \neq n$, G_m and G_n have no nodes in common (except for the midpoint when m and n are both odd). Thus, Gaussian rules are not progressive.

- Konrod Quadrature

    - Resolves the issue of Gauss Quadrature not being progressive.
    - The rule K_{2n+1} is therefore of degree $3n + 1$, whereas a true $(2n + 1)$-point Gaussian rule would be of degree $4n + 1$.
    - They provide both high accuracy and a reliable error estimate.

- Composite Quadrature

    - In principle, by taking k sufficiently large it is possible to achieve arbitrarily high accuracy (up to the limit of the arithmetic precision) using a composite rule, even with an underlying rule Q_n of low degree, although this may not be the most efficient lway to attain a given evel of accuracy.
    - Composite quadrature rules also offer a particularly simple means of estimating the error by using different levels of subdivision, which can easily be made progressive.

- Adaptive Quadrature

    - Leads to a nonuniform sampling of the integrand function that places many sample points in regions where the function is difficult to integrate and relatively few points where the function is easily integrated.

- Differentiation

    - Interpolation
      For data that is discrete or non-smooth
    - Finite difference approximations
      For smooth functions
      - forward and backward difference formrula are first order accurate
      - centered difference formula is second order accurate

- Richardson Extrapolation

    - The extrapolation process can be repeated to produce still more accurate approximations, up to the limitations imposed by finite-precision arithmetic.
    - Example: Romberg integration
      First, composite trapezoid rule with step size $h_k = (b-a)/2^k$, then Richardson extrapolation.

8 ODE's and BVP's

8.1 Introduction

8.2 Numerically solving ODE's

8.2.1 Euler Forward method

The simplest example of this approach is **Euler's method**, for which the approximate solution at time $t_{k+1} = t_k + h_k$ is given by

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(t_k, \mathbf{y}_k)$$

Euler's method can be derived in several ways:

- **Finite diffference appoximation**

If we replace the derivative $\mathbf{y}'(t)$ in the ODE $\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$ by a first-order forward difference approximation (see notebook on integration and differentiation), we obtain an algebraic equation

$$\frac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h_k} = \mathbf{f}(t_k, \mathbf{y}_k)$$

which gives Euler's method when solved for y_{k+1} .

- **Taylor Series**

Consider the Taylor series

$$\mathbf{y}(t+h) = \mathbf{y}(t) + h\mathbf{y}'(t) + 1/2h^2\mathbf{y}''(t) + \cdots$$

Euler's method results from taking $t = t_k$, $h = h_k$, $\mathbf{y}'(t_k) = \mathbf{f}(t_k, \mathbf{y}_k)$, and dropping terms of second and higher order.

8.2.2 Accuracy and Stability

- **Rounding error** , which is due to the finite precision of floating-point arithmetic
- **Truncation error** (or **discretization error**), which is due to the method used, and which would remain, even if all arithmetic were performed exactly
- **Global error** is the cumulative overall error

$$\mathbf{e}_k = \mathbf{y}_k - \mathbf{y}(t_k)$$

where $\mathbf{y}_k$ is the computed solution at t_k and $\mathbf{y}(t)$ is the true solution of the ODE passing through the initial point (t_0 ,$\mathbf{y}_0$).

- **Local error** is the error made in one step of the numerical method,

$$\mathbf{l}_k = \mathbf{y}_k - \mathbf{u}_{k-1}(t_k)$$

- **Accuracy**

The accuracy of a numerical method is said to be of order p if

$$\mathbf{l}_k = \mathcal{O}(h_k^{p+1})$$

- **Stability**

Recall that a solution to an ODE is stable if perturbations of the solution do not diverge away from it over time. Similarly, a numerical method is said to be stable if small perturbations do not cause the resulting numerical solution to diverge away without bound.

Such divergence of numerical solutions could be caused by instability of the solution to the ODE, but can also be caused by the numerical method itself, even when the solutions to the ODE are stable.

8.2.3 Euler backward method

The simplest example is the **Euler backward** method

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(t_{k+1}, \mathbf{y}_{k+1})$$

The backward Euler method is implicit because we must evaluate f with the argument y_{k+1} *before we know its value*.

We can do this with a nonlinear solver, as we have seen previously.

Why do we do this?

The answer is that implicit methods generally have a significantly larger stability region than comparable xplicit methods.

To determine the stability and accuracy of the backward Euler method, we apply it to the scalar test ODE $y' = \lambda y$, obtaining

$$y_{k+1} = y_k + h\lambda y_{k+1}$$

or

$$(1 - h\lambda) y_{k+1} = y_k$$

so that

$$y_k = \left(\frac{1}{1 - h\lambda} \right)^k y_0$$

.

Thus, for the backward Euler method to be stable we must have

$$\left| \frac{1}{1 - h\lambda} \right| \leq 1$$

which holds for any $h > 0$ when Re(λ) < 0.

8.2.4 Stiffness

Stiffness is a concept that can be defined a number of ways. For us, the most meaningful way is it correspondence to the physics behind the problems we are investigating.

If a system contains dynamics on very different timescales, like a slow relaxation towards a certain equilibrium, but with rapid oscillations around it, or with very strongly damped transients, then it is considered stiff.

Mathematically, a stable ODE $\mathbf{y}_0 = \mathbf{f}(t, \mathbf{y})$ is stiff if its Jacobian
matrix $\mathbf{J}_f$ has eigenvalues that differ greatly in magnitude.

Euler's forward method, for example, is extremely inefficient for solving a stiff equation because of its small stability region. The unconditional stability of the implicit backward Euler method, on the other hand, makes it suitable for stiff problems.

8.2.5 Runge-Kutta methods

Runge-Kutta methods are single-step methods that are similar in motivation to Taylor series methods but do not involve explicit computation of higher derivatives.
Instead, Runge-Kutta methods replace higher derivatives by finite difference approximations based on values of f at points between t_k and t_{k+1}.

Runge-Kutta methods have a number of virtues.

- To proceed to time t_{k+1}, they require no history of the solution prior to time t_k
- which makes them self-starting at the beginning of the integration
- and also makes it easy to change the step size during the integration
- These features also make Runge-Kutta methods relatively easy to program, which accounts in part for their popularity.

In general, the approximate solution of the ODE

$$\frac{dy}{dt} = f(t, y)$$

at time $t + h$, given its value y on time t, is obtained by taking s intermediate evaluations of f at times c_i,

$$k_i = f\left(t + c_i h, y_n + h \sum_{j=1}^{s} a_{ij} k_j\right)$$

and then adding them to y_n with the correct weights b_i:

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i$$

These formulas can compactly be represented by a Butcher tableau as follows:

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\ c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\ \hline & b_1 & b_2 & \cdots & b_s \end{array}$$

If the tableau only contains elements below the diagonal, than it corresponds to an **explicit** solver. Otherwise it is an **implicit** solver.

For example:

- **Euler Forward**

$$\begin{array}{c|c} 0 & \\ \hline & 1 \end{array}$$

- **Euler Backward**

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array}$$

- **4th order Runke-Kutta solver**

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

8.2.5.1 Adaptive step size

Classical Runge-Kutta methods require a **fixed step**

When a solver contains not only a solution of order $\mathcal{O}(N)$ but also a solution of $\mathcal{O}(N-1)$, we can use the difference between both solutions as an approximation of the error size ϵ on the solution.
This error depends on the size of the time step h, and given a certain error tolerance τ, it is possible to suggest a h for the next time step which is a large as possible, while still maintaining the level of accuracy required as follows:

$$h_{\text{optimal}} = h_{\text{current}} \left(\frac{\epsilon}{h_{\text{current}} \tau} \right)^{(1/N)}$$

These solvers can therefore use an **adaptive step size**.

A simple example of such a solver is **Heun's method**.
In this method, the first order accurate solution is found using Euler's forward method

$$\tilde{y}_{n+1} = y_n + hf(t, y_n)$$

Afterwards, the second order solution is found:

$$y_{n+1} = y_n + \frac{h}{2}\Big(f(t, y_n) + f(t+h, \tilde{y}_{n+1})\Big)$$

The difference between both solutions is an estimate of the error.

The butcher tableau looks like this:

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & \frac{1}{2} & \frac{1}{2} \\ & 1 & 0 \end{array}$$

Probably the most used embedded pair method was developed by Dormand and Prince. This is a 5th order accurate solver with a 4th order embedded error estimate.

$$\begin{array}{c|ccccccc} 0 & & & & & & & \\ \frac{1}{5} & \frac{1}{5} & & & & & & \\ \frac{3}{10} & \frac{3}{40} & \frac{9}{40} & & & & & \\ \frac{4}{5} & \frac{44}{45} & -\frac{56}{15} & \frac{32}{9} & & & & \\ \frac{8}{9} & \frac{19372}{6561} & -\frac{25360}{2187} & \frac{64448}{6561} & -\frac{212}{729} & & & \\ 1 & \frac{9017}{3168} & -\frac{355}{33} & \frac{46732}{5247} & \frac{49}{176} & -\frac{5103}{18656} & & \\ 1 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & \\ \hline & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & 0 \\ & \frac{5179}{57600} & 0 & \frac{7571}{16695} & \frac{393}{640} & -\frac{9209}{339200} & \frac{187}{2100} & \frac{1}{40} \end{array}$$

8.2.6 First-same-as-last (FSAL) property

C onsider the **Bogacki-Shampine** method, which is a third order method with embedded second order solution.
This method seems to require 4 function evaluations per step.

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{3}{4} & 0 & \frac{3}{4} & & \\ 1 & \mathbf{\frac{2}{9}} & \mathbf{\frac{1}{3}} & \mathbf{\frac{4}{9}} & \\ \hline & \mathbf{\frac{2}{9}} & \mathbf{\frac{1}{3}} & \mathbf{\frac{4}{9}} & 0 \\ & \frac{7}{24} & \frac{1}{4} & \frac{1}{3} & \frac{1}{3} \end{array}$$

However, contrary to Heun's method, in the Bogacki-Shampine method the last evaluation of step n corresponds with the first evaluation of step $n+1$ (which is shown as the identical lines in bold), thus effectively reducing the number of evaluations per step to 3.

This property is called is the **first-same-as-last (FSAL)** property and makes this solver 4/3 times more efficient compared to the case when it didn't have the FSAL property.

8.2.7 Performance

At first sight, it seems to pay off to implement increasingly complex and higher-order solvers. However, for each additional order, a number of extra evaluations per step are necessary, as shown in the table below.

solver	Euler	Heun12	RK3	Bogacki − Shampine23	RK4	Dormand − Prince45	Fehlb
$\frac{\#\text{evaluations}}{\text{step}}$	1	2	3	4	4	6	8

There also exists a theoretical limit which order can be achieved by a certain number of evaluations per step.

$\mathcal{O}(\text{solver})$	1	2	3	4	5	6	7	8
$\frac{\#\text{evaluations}}{\text{step}}$	1	2	3	4	6	7	9	11

Note that the Bogacki-Shampine(when not considering FSAL) and the Fehlberg methods appear to be suboptimal, but this stems from the fact that they also have a lower order solution embedded, which further increases the number of conditions their numbers in the Butcher tableau have to fulfill, and consequently require more variables and thus more evaluations per step.

A second point to take into account is the memory usage of these solvers. The Seventh order Fehlberg method is only slightly faster than the Sixth order Fehlberg method, but uses 13 evaluations per step, as compared to 8.
It thus requires almost twice the amount of memory.
Especially in GPU-software, where the memory bandwidth often is a limiting factor, such considerations need to be taken into account.

8.2.8 Extrapolation methods

Extrapolation methods are based on the use of a single-step method to integrate the ODE over a given interval, $t_k \leq t \leq t_{k+1}$, using several different step sizes h_i and yielding results denoted by $\mathbf{Y}(h_i)$.
This gives a discrete approximation to a function $Y(h)$, where $\mathbf{Y}(0) = \mathbf{y}(t_{k+1})$.

An interpolating polynomial or rational function $\hat{\mathbf{Y}}(h)$ is fit to these data, and $\hat{\mathbf{Y}}(0)$ is then taken as the approximation to $\mathbf{Y}(0)$.

This is the same as *Richardson Interpolation*

Extrapolation methods are capable of achieving very high accuracy, but they are much less efficient and less flexible than other methods for ODEs, so they are used mainly when extremely high accuracy is required and cost is not a significant factor.

8.2.9 Multistep methods

Whereas Runge-Kutta methods only use information of one previous point (i.e. a single step method), **Multistep methods** use information at more than one previous point to estimate the solution at the next point.

One of the most popular *explicit* multistep methods is the fourth-order **Adams-Bashforth** method, which uses information of 3 previous time steps, next to the current one:

$$\mathbf{y}_{k+1} = \mathbf{y}_k + \frac{h}{24}\left(55\mathbf{y}'_k - 59\mathbf{y}'_{k-1} + 37\mathbf{y}'_{k-2} - 9\mathbf{y}'_{k-3}\right)$$

Derivation

This derivation considers a scalar function y, but the results can be applied componentwise to nonscalar functions as well.

We derive a multistep method of the form

$$\mathbf{y}_{k+1} = \alpha\mathbf{y}_k + h(\beta_0\mathbf{y}'_k + \beta_1\mathbf{y}'_{k-1} + \beta_2\mathbf{y}'_{k-2} + \beta_3\mathbf{y}'_{k-3})$$

To determine the 5 coefficients $\alpha, \beta_0, \beta_1, \beta_2$ and β_3, we require that this formula exactly integrates the first 5 monomials $1, t, t^2, t^3$ and t^4.

$$\begin{aligned}
1 &= \alpha + h(\beta_0 \cdot 0 + \beta_1 \cdot 0 + \beta_2 \cdot 0 + \beta_3 \cdot 0) \\
t_{k+1} &= \alpha t_k + h(\beta_0 \cdot 1 + \beta_1 \cdot 1 + \beta_2 \cdot 1 + \beta_3 \cdot 1) \\
t_{k+1}^2 &= \alpha t_k^2 + h(\beta_0 2t_k + \beta_1 2t_{k-1} + \beta_2 2t_{k-2} + \beta_3 2t_{k-3}) \\
t_{k+1}^3 &= \alpha t_k^3 + h(\beta_0 3t_k^2 + \beta_1 3t_{k-1}^2 + \beta_2 3t_{k-2}^2 + \beta_3 3t_{k-3}^2) \\
t_{k+1}^4 &= \alpha t_k^4 + h(\beta_0 4t_k^3 + \beta_1 4t_{k-1}^3 + \beta_2 4t_{k-2}^3 + \beta_3 4t_{k-3}^3)
\end{aligned}$$

Because this method needs to work for *any* value of t_k and h, we can conventiently choosing $t_k = 0$ and $h = 1$.

It then follows that $t_{k+1} = 1$, $t_{k-1} = -1$, $t_{k-2} = -2$ and , $t_{k-3} = -3$

The first equation in the system thus becomes

$$1 = \alpha \cdot 1 + h(0)$$

From which it follows that $\alpha = 1$. The remaining system of equations thus reduces to

$$\begin{aligned}
1 &= \beta_0 + \beta_1 + \beta_2 + \beta_3 \\
1 &= -2\beta_1 - 4\beta_2 - 6\beta_3 \\
1 &= 3\beta_1 + 12\beta_2 + 27\beta_3 \\
1 &= 1 - 4\beta_1 - 32\beta_2 - 108\beta_3
\end{aligned}$$

Which we can solve using `linalg.solve` (as seen in the linear systems notebook) to find the coefficients of the Adams-Bashforth method.

One of the most popular *implicit* multistep methods is the fourth-order **Adams-Moulton** method, which uses information of 3 previous time steps, next to the current one:

$$\mathbf{y}_{k+1} = \mathbf{y}_k + \frac{h}{24}\left(9\mathbf{y}'_{k+1} + 19\mathbf{y}'_k - 5\mathbf{y}'_{k-1} + \mathbf{y}'_{k-2}\right)$$

Just like for single-step methods, implicit multistep methods are usually more accurate and stable than explicit multistep methods, but they require an initial guess to solve the resulting (usually nonlinear) equation for $\mathbf{y}_{k+1}$.
A good initial guess is conveniently supplied by an explicit method, so the explicit and implicit methods can be used as a **predictor-corrector pair**.
One of the most used pairs is the Adams-Bashforth predictor and Adams-Moulton corrector shown above.

A few properties of multistep methods worth knowing

- How do we get initial steps?
  One strategy is to use a single-step method, which requires no past history, to generate solution values at enough points to begin using a multistep method.

- Changing step size is complicated, since the interpolation formulas are most conveniently based on equally spaced intervals for several consecutive points, so multistep methods are not idealy suited for adaptive step sizes.

- A good local error estimate can be determined from the difference between the predictor and the corrector.

- Implicit methods have a much greater region of stability than explicit methods but must be iterated to convergence to realize this benefit fully (e.g., a PECE scheme is actually explicit, albeit in a somewhat complicated way).

- A properly designed implicit multistep method can be very effective for solvingstiff equations.

8.2.10 Multivalue methods

Multivalue methods are a direct extension of multistep methods which allow adaptive step sizes (at the cost of more function evaluations per step and a more complicated implementation).

8.3 BVP's for ODE's

Differential equations have more than one solution, boundary conditions impose a single unique solution.

The two-point boundary value problem for the second-order scalar ODE

$$u'' = f(t, u, u')$$

with $a < t < b$

and boundary conditions

- $$u(a) = \alpha$$
- $$u(b) = \beta$$

is equivalent to the first oder system of ODEs:

$$\begin{bmatrix} y_1' \\ y_2' \end{bmatrix} = \begin{bmatrix} y_2 \\ f(t(y1, y2) \end{bmatrix}$$

with $a < t < b$

and with separated boundary conditions

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} y_1(a) \\ y_2(a) \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} y_1(b) \\ y_2(b) \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

For the general first-order two-point boundary value problem

$$\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$$

with $a < t < b$
and boundary conditions

$$\mathbf{g}(\mathbf{y}(a), \mathbf{y}(b)) = \mathbf{0}$$

let $\mathbf{y}(t; \mathbf{x})$ denote the solution to the associated initial value problem with initial condition $\mathbf{y}(a) = x$.
For a given $\mathbf{x}$, the solution $\mathbf{y}(t; \mathbf{x})$ of the IVP is a solution of the BVP if

$$\mathbf{h}(\mathbf{x}) \equiv \mathbf{g}(\mathbf{x}, \mathbf{y}(b; \mathbf{x})) = \mathbf{0}$$

8.3.1 Shooting method

Replaces a BVP with a sequence of initial value problems.
The general first-order two-point boundary value problem is equivalent to the system of nonlinear algebraic equations

$$\mathbf{h}(\mathbf{x}) \equiv \mathbf{g}(\mathbf{x}, \mathbf{y}(b; \mathbf{x})) = \mathbf{0}$$

You can solve it by solving the nonlinear system $\mathbf{h}(\mathbf{x}) = \mathbf{0}$ using a non-linear solver.

Let's take

$$u'' = f(t, u, u')$$

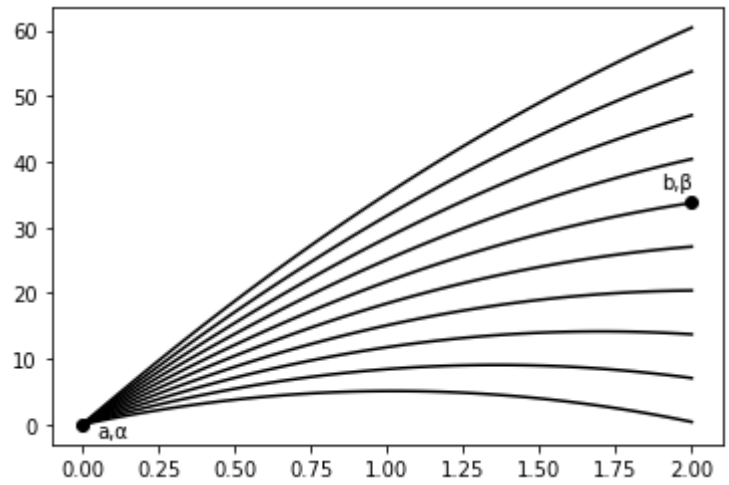
with $a < t < b$

and boundary conditions

- $$u(a) = \alpha$$
- $$u(b) = \beta$$

This is not an IVP, because we get $u(b)$ instead of $u'(a)$.
We can guess a value for the initial slope, solve the resulting IVP, and then check to see if the computed solution value at $t = b$ matches the desired boundary value, $u(b) = \beta$.

The motivation for the name **shooting method** should now be obvious: we keep adjusting our aim until we hit the target.



For a second order BVP:

$$\mathbf{g}(\mathbf{y}(a), \mathbf{y}(b)) = \begin{bmatrix} y_1(a) - \alpha \\ y_1(b) - \beta \end{bmatrix} = \mathbf{0}$$

Thus, the nonlinear system to be solved is

$$\mathbf{h}(\mathbf{x}) = \begin{bmatrix} y_1(a; \mathbf{x}) - \alpha \\ y_1(b; \mathbf{x}) - \beta \end{bmatrix} \mathbf{0}$$

where x is the initial value.

The first component of $\mathbf{h}(\mathbf{x})$ will be zero if $x_1 = \alpha$, and the initial slope x_2 remains to be determined so that the second component of $\mathbf{h}(\mathbf{x})$ will be zero.

In effect, therefore, we must solve the scalar nonlinear equation in x_2 ,

$$h_2(\alpha, x_2) = y_1(b; \alpha, x_2) = 0$$

for which we can use a one-dimensional zero finder.

This method is conceptually simple and easy to implement, but has drawbacks:

- it inherits the stability issues of an IVP (even though the BVP might be stable)
- It is hard to get convergence, because the IVP might be ill-conditioned, or the IVP might not exist for stome starting guess of the initial value (become unbounded).

We can get around this by diving the interval into a mesh and requiring continuity at the mesh points (*mulptiple shooting*), but it results in a larger and more complicated system of nonlinear equations to solve.

8.4 Summary

- Euler methods
  Use a truncated Taylor expansion

  - Euler Forward
    - Easiest algorithm to implement
    - Inefficient for solving stiff ODEs
  - Euler Backward
    - Implicit methods have larger stability regions compared to explicit methods
    - can solve stiff ODEs efficiently

- Runge-Kutta methods
  Use finite difference approximations of higher derivatives based on values between the the points t_k and t_{k+1}

  - Self-starting at the beginning of the integration
  - Easy to change the step size during the integration
  - Easy to program
  - Fixed step

- Adaptive step size
  Used when a solver contains a solution of order $\mathcal{O}(N)$ and a solution of $\mathcal{O}(N-1)$.

  - Heun's method
    Uses Euler forwards method and calculates a second order solution.
  - Dormand Prince
    5th order accurate solver with a 4th order embedded error estimate.

- FSAL
  When the last evaluation of the previous step is the same as the first evaluation of the next step. Makes this solver 4/3 times more efficient compared to the case when it didn't have the FSAL property.

  - Bogacki-Shampine
    Third order method with embedded second order solution.

- Performance

  The higher order the better, but each order requires some new function evals. So there's a theoretical limit on what order you can reach based on the number evals per step.

Also important is the amount of memory high order solvers use, because GPUs often have limited VRAM.

- Extrapolation methods
  Same idea as Richardson interpolation

  - Capable of high accuracy (to machine precision)
  - Less efficient and les flexible
  - Only used when cost/time isn't a factor

- Multistep methods
  Use information of more than one previous point to determine next point
  In general:
  - Need guesses for the first few points (single-step method can be used)
  - Are not ideal for adaptive step size (because most methods are based on equally spaced intervals)
  - Implicit is generally more stable than explicit and can be used to solve stiff ODEs

  - Methods
    - Adams-Bashforth
      Use the 3 previous steps (next to the current one)
      - Explicit
    - Adams-Moulton
      Use the 3 previous steps (next to the current one)
      - Implicit
      - More stable than explicit method
      - Needs to guess the first few points, so often Adams-Bashforth and Adams-Moulton is used as a predictor-corrector pair.

- Multivalue methods
  Allow for adaptive step sizes in multistep methods

- Shooting method
  Used to solve BVPs, we guess an intial slope.

  - Conceptually easy to implement

  - it inherits the stability issues of an IVP (even though the BVP might be stable).

  - It is hard to get convergence, because the IVP might be ill-conditioned, or the IVP might not exist for stome starting guess of the initial value (become unbounded).

  - Multiple shooting
    subdiving the interval

    - A bit more stable
    - Requires a larger and more complicated nonlinear system

9 PDEs

9.1 Introduction

We seek to determine a 2d function u, subjugated to some BCs or IVs.

Such a solution function u can be visualized as a surface over the relevant two-dimensional domain in the (t, x) or (x, y) plane.

We denote the unknown solution function by u, and we denote its partial derivatives with respect to the independent variables by appropriate subscripts:

- $$u_t = \partial u / \partial t$$
- $$u_{xy} = \partial^2 u / \partial x \partial y$$
- $$\dots$$

9.2 Classification and examples

- **Heat equation**

$$u_t = u_{xx}$$

- **Wave equation**

$$u_{tt} = u_{xx}$$

- **Laplace equation**

$$u_{xx} + u_{yy} = 0$$

Any second-order linear PDE of the form

$$au_{xx} + bu_{xy} + cu_{yy} + du_x + eu_y + fu + g = 0$$

can be transformed into these three equations.

- $b^2 - 4ac > 0$: **hyperbolic**, typified by the wave equation
- $b^2 - 4ac = 0$: **parabolic**, typified by the heat equation
- $b^2 - 4ac < 0$: **elliptic**, typified by the Laplace equation
- *Hyperbolic* PDEs describe time-dependent, conservative physical processes, such as convection, that are not evolving toward a steady state.
- *Parabolic* PDEs describe time-dependent, dissipative physical processes, such as diffusion, that are evolving toward a steady state.
- *Elliptic* PDEs describe systems that have already reached a steady state, or equilibrium, and hence are time-independent.

This is important, because o fparabolic PDEs have a smoothing effect that over time damps out any lack smoothness in the initial conditions, whereas hyperbolic PDEs propagate steep fronts or shocks undiminished, and discontinuities can develop in the solution even with smooth initial data.

Systems governed by hyperbolic PDEs are in principle reversible in time, whereas parabolic systems are not.

9.3 Solving time-dependent problems

- Semidiscrete
  Discretize in space but let the time variable be continuous. You then get a system of ODEs to solve.

> **Example: solving the heat equation with a semidiscrete method**
>
> Consider the equation $u_t = cu_{xx}$, $0 \leq x \leq 1$, $t \geq 0$.
> If we introduce spatial mesh points $x_i = i\Delta x$, $i = 0, \ldots, n+1$, where $\Delta x = 1/(n+1)$, and replace the second derivative u_{xx} with the finite difference approximation
>
> $$u_{xx}(t, x_i) \approx \frac{u(t, x_{i+1}) - 2u(t, x_i) + u(t, x_{i-1})}{(\Delta x)^2} \quad , \quad i = 1, \ldots, n$$
>
> but leave the time variable continuous, then we obtain a system of ODEs
>
> $$y_i'(t) = \frac{c}{(\Delta x)^2}(y_{i+1}(t) - 2y_i(t) + y_{i-1}(t)) \quad , \quad i = 1, \ldots, n$$
>
> where $y_i(t) \approx u(t, x_i)$.
> From the boundary conditions we know that, $y_0(t) = 0$ and $y_{n+1}(t) = 0$, and from the initial conditions, that $y_i(0) = f(x_i)$, $i = 1, \ldots, n$
>
> We can therefore use an ODE method to solve the initial value problem for this system.
> The foregoing semidiscrete system of ODEs can be written in matrix form as
>
> $$\mathbf{y}' = \frac{c}{(\Delta x)^2}\begin{bmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \cdots & 0 \\ 0 & 1 & -2 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & -2 \end{bmatrix}\mathbf{y} = \mathbf{Au}$$
>
> The Jacobian matrix $\mathbf{A}$ of this system has eigenvalues between $-4c/(\Delta x)^2$ and 0, which makes the ODE very stiff as the spatial mesh size Δx becomes small.

> This stiffness, which is typical of ODEs derived from PDEs in this manner, must be taken into account in choosing an appropriate ODE method for solving the semidiscrete system.

- Fully discrete
  Make a meshgrid in time and space. Use finite difference approximations for all the derivatives. You get a system of equations, which might be linear or nonlinear depending on the underlying PDE. IVPs are solved by beginning with the initial values and integraitng forward step by step in time. Such a procedure can be explicit or implicit.

  > **Example: solving the heat equation with a fully discrete method**
  >
  > We define spatial mesh points $x_i = i\Delta x$, $i = 0, 1, \ldots, n+1$, where $\Delta x = L/(n+1)$,
  > and temporal mesh points $t_k = k\Delta t$, $k = 0, 1, \ldots \$, where \$\Delta t$ is chosen appropriately.
  >
  > We denote the approximate solution at mesh point (t_k, x_i) by u_i^k , where we have used both a subscript and a superscript (the k is not an exponent) to distinguish clearly between increments in space and time, respectively.
  >
  > Using centered difference approximations for both u_{tt} and u_{xx} yields a system of algebraic equations
  >
  > $$\frac{u_i^{k+1} - 2u_i^k + u_i^{k+1}}{(\Delta t)^2} = c\frac{u_{i+1}^k - 2u_i^k + u_{i-1}^k}{(\Delta x)^2} \quad , \quad i = 1, \ldots, n$$
  >
  > which can be rearranged to give the explicit recurrence
  >
  > $$u_i^{k+1} = 2u_i^k - u_i^{k-1} + c\left(\frac{\Delta t}{\Delta x}\right)^2 (u_{i+1}^k + 2u_i^k + u_{i-1}^k)$$

9.3.1 Implicit methods

As with ODEs, a larger stability region that permits larger time steps can be obtained by using implicit methods. For the heat equation, for example, applying the backward Euler method to the semidiscrete system shown in the Heat equation example yields the implicit finite difference scheme

$$u_i^{k+1} = u_i^k + c\frac{\Delta t}{(\Delta x)^2}(u_{i+1}^{k+1} - 2u_i^{k+1} + u_{i-1}^{k+1}) \quad , \quad i = 1, \ldots, n$$

This scheme inherits the unconditional stability of the backward Euler method, which means that there is no stability restriction on the relative sizes of Δt and Δx.

Accuracy is still a consideration, however, and the fact that this particular method is only first-order accurate in time still strongly limits the time step.

If instead we apply the trapezoid method we obtain the implicit finite difference scheme

$$u_i^{k+1} = u_i^k + c\frac{\Delta t}{2(\Delta x)^2}(u_{i+1}^{k+1} - 2u_i^{k+1} + u_{i-1}^{k+1} + u_{i+1}^k - 2u_i^k + u_{i-1}^k) \quad , \quad i = 1, \ldots, n$$

This is called the *Crank-Nicolson method* and is unconditionally stable and is second-order accurate in time as well as in space.

9.4 Solving time-independent problems

Just as *time-dependent* parabolic and hyperbolic PDEs are analogous to *initial value problems* for ODEs, time-*independent* elliptic PDEs are analogous to *boundary-value problems* for ODEs, and most of the solution methods for ODE BVPs carry over to elliptic PDEs as well.

For an elliptic boundary value problem, the solution at every point in the problem domain depends on all of the boundary data (in contrast to the limited domain of dependence for time-dependent problems), and consequently an approximate solution must be computed everywhere simultaneously, rather than being generated step by step using a recurrence, as in the previous examples.

9.4.1 Laplace equation

The Laplace equation is a special case of the **Poisson equation**, which in two space dimensions has the form

$$u_{xx} + u_{yy} = f(x, y)$$

where f is a given function defined on a domain whose boundary is typically a closed curve in $\mathbb{R}^2$, such as a square or circle.

If $f \equiv 0$, then we have the **Laplace equation**.

There are numerous possibilities for the boundary conditions that must be specified on the boundary of the domain or portions thereof:

- **Dirichlet boundary conditions**, sometimes called essential boundary conditions, in which the solution u is specified.
- **Neumann boundary conditions**, sometimes called natural boundary conditions, in which one of the derivatives u_x or u_y is specified.
- **Robin boundary conditions**, or **mixed boundary conditions**, in which a combination of solution values and derivative values is specified.

> **Example: solving the Laplace equation with a finite difference method**
> Consider the Laplace equation on the unit square

$$u_{xx} + u_{yy} = 0, \quad 0 \le x \le 1, \quad 0 \le y \le 1,$$

We define a discrete mesh in the domain, including boundaries.
The interior grid points where we will compute the approximate solution are given by

$$(x_i, y_j) = (ih, jh), \quad i, j = 1, \ldots, n$$

where in our example $n = 2$ and $h = 1/(n+1) = 1/3$.

Next we replace the second derivatives in the equation with the standard second-order centered difference
approximation at each interior mesh point to obtain the finite difference equations

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} = 0 \quad , \quad i, j = 0, \ldots, n$$

where $u_{i,j}$ is an approximation to the true solution $u(x_i, y_j)$ and represents one of the given boundary values if i or j is 0 or $n+1$.

Simplifying and writing out the resulting four equations explicitly, we obtain

$$\begin{aligned}
4u_{1,1} - u_{0,1} - u_{2,1} - u_{1,0} - u_{1,2} &= 0 \\
4u_{2,1} - u_{1,1} - u_{3,1} - u_{2,0} - u_{2,2} &= 0 \\
4u_{1,2} - u_{0,2} - u_{2,2} - u_{1,1} - u_{1,3} &= 0 \\
4u_{2,2} - u_{1,2} - u_{3,1} - u_{2,1} - u_{2,3} &= 0
\end{aligned}$$

Writing these four equations in matrix form, we obtain

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{bmatrix} \begin{bmatrix} u_{1,1} \\ u_{2,1} \\ u_{1,2} \\ u_{2,2} \end{bmatrix} = \begin{bmatrix} u_{0,1} + u_{1,0} \\ u_{3,1} + u_{2,0} \\ u_{0,2} + u_{1,3} \\ u_{3,2} + u_{2,3} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \mathbf{b}$$

This symmetric positive definite system of linear equations can be solved either by Cholesky factorization or by an iterative method, yielding the solution

$$\mathbf{x} = \begin{bmatrix} u_{1,1} \\ u_{2,1} \\ u_{1,2} \\ u_{2,2} \end{bmatrix} = \begin{bmatrix} 0.125 \\ 0.125 \\ 0.375 \\ 0.375 \end{bmatrix}$$

Note the symmetry in the solution, which reflects the symmetry in the problem, which we could have taken advantage of and solved a problem only half as large.

In a practical problem, the mesh size h would need to be much smaller to achieve acceptable accuracy in the approximate solution of the PDE, and the resulting linear system would be much larger than in the preceding example.

The matrix would be very sparse, however, since each equation would still involve at most only five of the variables, thereby saving substantially on work and storage.

9.5 Summary

Any second-order linear PDE of the form

$$au_{xx} + bu_{xy} + cu_{yy} + du_x + eu_y + fu + g = 0$$

can be transformed into these three equations.

- $b^2 - 4ac > 0$: **hyperbolic**
    - typified by the wave equation
    - describe time-dependent, conservative physical processes, such as convection, that are not evolving toward a steady state.
- $b^2 - 4ac = 0$: **parabolic**,
    - typified by the heat equation
    - describe time-dependent, dissipative physical processes, such as diffusion, that are evolving toward a steady state.
- $b^2 - 4ac < 0$: **elliptic**,
    - typified by the Laplace equation
    - describe systems that have already reached a steady state, or equilibrium, and hence are time-independent.
- Solving time-dependent problems (hyperbolic & parabolic)
    - Semidiscrete
      Discretize space but leave time continuous
        - Typically very stiff system of equations
    - Fully discrete
      Discretize time and space

    can be explicit or implicit

    - implicit methods have the usual benifits:
      -when using Euler backward:
      - unconditional stability
      - not that accurate (only first order)
        - when using Crank-Nicolson (trapezoid method for finite difference schemes)

    - unconditional stability
    - *second* order accurate

- Solving time-independent problems (elliptic)
  We must calculate the entire solution at once because all data depends on all other data.
  We can use fully discrete methods similarly to time-dependent problems

  Types of boundary conditions:

  - **Dirichlet boundary conditions**, sometimes called essential boundary conditions, in which the solution u is specified.
  - **Neumann boundary conditions**, sometimes called natural boundary conditions, in which one of the derivatives u_x or u_y is specified.
  - **Robin boundary conditions**, or **mixed boundary conditions**, in which a combination of solution values and derivative values is specified.

10 FFT

10.1 DFT

10.1.1 Definition

Given a sequence $\mathbf{x} = [x_0, \dots, x_{n-1}]^T$, its **discrete Fourier transform** (or DFT), is the sequence $\mathbf{y} = [y_0, \dots, y_{n-1}]^T$ given by

$$y_m = \sum_{k=0}^{n-1} \omega_n^{mk} x_k, \quad \forall\, m \in \{0, 1, \dots, n-1\}$$

where

$$\omega_n = \cos(2\pi/n) - i \sin(2\pi/n) = e^{-2\pi i/n}$$

for the (or rather *one particular*) n^{th} root of unity, meaning that $\omega_n^n = 1$.

Also note the effect of complex conjugation: $\omega_n^* = \omega_n^{-1} = \omega_n^{n-1}$.

This can be written in matrix notation as $\mathbf{y} = \mathbf{F_{nx}}$ where the entries of the symmetric Fourier matrix $\mathbf{F}_n$ are given by

$$\{\mathbf{F}_n\}_{mk} = \omega_n^{mk}$$

The inverse of $\mathbf{F}_n$ is given by $\mathbf{F}_n^{-1} = (1/n)\mathbf{F}_n^H$.

> **Short Proof**
>
> $$\{\mathbf{F}_n \mathbf{F}_n^{-1}\}_{mm} = \frac{1}{n} \sum_{k=0}^{n-1} \omega_n^{mk} (\omega_n^*)^{km} = 1$$
>
> $$\{\mathbf{F}_n \mathbf{F}_n^{-1}\}_{m\ell} = \frac{1}{n} \sum_{k=0}^{n-1} \omega_n^{mk} (\omega_n^*)^{k\ell} = \frac{1}{n} \sum_{k=0}^{n-1} \omega_n^{(m-\ell)k} = \frac{1}{n} \frac{1 - \omega_n^{n(m-\ell)}}{1 - \omega_n^{m-\ell}} = 0$$
>
> In the last step, we made use of $\omega_n^n = 1$.

The **inverse** DFT thus reads

$$\begin{aligned} x_k &= \frac{1}{n} \sum_{m=0}^{n-1} \omega_n^{-km} y_m \\ &= \frac{1}{n} \sum_{m=0}^{n-1} y_m \big[\cos(2\pi mk/n) + i \sin(2\pi mk/n)\big] \qquad \forall\, k \in \{0, 1, \dots, n-1\} \end{aligned}$$

10.1.2 Frequencies

In the definitions of the (inverse) DFT, the frequency is represented by an dimensionless integer index m, which seems to have the wrong unit.
It is implicitly assumed that the samples x_k are taken at regular (time) steps

$$t_k = t_0 + k/f_s.$$

where f_s is called the sampling rate.
Put differently,

$$k = (t_k - t_0) f_s.$$

With that choice, the sine and cosine basis functions, can be written as a function of time:

$$\begin{aligned}\omega_n^{-mk} &= \cos(2\pi mk/n) + i\sin(2\pi mk/n) \\ &= \cos\left(\frac{2\pi m}{n}(t_k - t_0) f_s\right) + i\sin\left(\frac{2\pi m}{n}(t_k - t_0) f_s\right)\end{aligned}$$

As soon as a sampling rate f_s is assumed, the index m corresponds to a genuine frequency $(f_s m)/n$.

Note that one may also sample in a spatial domain instead of the time domain, in which case the index m corresponds to a wavenumber instead of a frequency.

Because $\omega_n^{-mk} = \omega_n^{(\ell n - m)k}$, where ℓ is an arbitrary integer, and $\omega_n^n = 1$, you can also discuss negative frequencies. Most often, the frequencies are given in lowest absolute value, or

$$\left[0, \frac{f_s}{n}, \frac{2f_s}{n}, \ldots, \frac{\lceil n/2 \rceil f_s}{n}, \frac{(\lceil n/2 \rceil - n) f_s}{n}, \ldots, \frac{-2f_s}{n}, \frac{-f_s}{n}, \right]$$

In the sequence of frequencies shown above, the highest one is $\lceil n/2 \rceil f_s/n$.
In the limit of many samples, or for any even number of samples, the highest frequency always becomes $f_s/2$, which is known is the Nyquist frequency.

If the underlying time-dependent function, of which x_k are samples, contains relevant fluctuations at frequencies above $f_s/2$, it will be impossible to discern them from lower-frequency fluctuations.
The reason is that, on the sampling grid, the basis function ω_n^{mk} is indistinguishable from $\omega_n^{(\ell n + m)k}$, where ℓ is an arbitrary integer.

In practice, this means that the sampling frequency should at least twice the highest relevant frequency of the underlying function. This is known as the Nyquist-Shannon sampling theorem.

The lowest of all frequencies is always zero (in Hz or m^{-1}).
The corresponding coefficient of the DFT is simply the sum of all values x_k and is called the DC (direct current) component.

Note that a constant shift of all x_k will only affect the DC component.

10.1.3 DFT of a real sequence

The DFT of a sequence (even a real sequence) is in general complex. This is not something to worry about, as the inverse DFT will take us back to the real domain.

The DFT of a real sequence of length n has $2n$ real and imaginary parts, but still contains only n independent pieces of information.
This can be understood as follows: for each basis function ω_n^{-km}, there is another basis function $\omega_n^{k(n+m)} = \omega_n^{km} = (\omega^*)_n^{km}$, which is just its complex conjugate.
To represent a real sequence $\mathbf{x}$, the DFT must adhere to $y_m = y_{n-m}^*$.
This has a few implications for real sequences:

- y_0 is real.
- When n is even, $y_{n/2}$ is also real.
- When n is odd, $y_{(n+1)}/2$ may still be complex.

In summary, the second half of a DFT of a real squence is redundant.

10.2 FFT algorithm

We can exploit certain symmetries and redundancies in the definition of the DFT to calculate it efficiently, with a cost scaling like $\mathcal{O}(n\log(n))$ instead of $\mathcal{O}(n^2)$.

> Consider the first few Fourier matrices:
>
> $$\mathbf{F_1} = 1$$
>
> $$\mathbf{F_2} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$
>
> $$\mathbf{F_4} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}$$
>
> Additionally let $\mathbf{P_4}$ be the permutation matrix, which separates the odd and even subsequences:
>
> $$\mathbf{P_4} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
>
> and $\mathbf{D_2}$ the diagonal matrix
>
> $$\mathbf{D_2} = \mathrm{diag}(1, \omega_4^1) = \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix}$$
>
> With these matrices, we can rearrange $\mathbf{F_4}$ such that each block is a diagonally scaled version of $\mathbf{F_2}$.

$$\mathbf{F_4P_4} = \left[\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & -1 & -i & i \\ \hline 1 & 1 & -1 & -1 \\ 1 & -1 & i & -i \end{array}\right] = \begin{bmatrix} \mathbf{F_2} & \mathbf{D_2F_2} \\ \mathbf{F_2} & \mathbf{-D_2F_2} \end{bmatrix}$$

In general, $\mathbf{P_n}$ is the permutation that groups the even-numbered columns of $\mathbf{F_n}$ before the odd numbered columns and

$$\mathbf{D}_{n/2} = \operatorname{diag}\left(1, \omega_n, \ldots, \omega_n^{(n/2)-1}\right)$$

Thus, to apply $\mathbf{F}_n$ to a sequence of length n, we merely need to apply $\mathbf{F}_{n/2}$ to its even and odd subsequences and scale the results by $\pm\mathbf{D}_{n/2}$.
This recursive *divide-and-conquer* approach is called the **fast Fourier transform** and can be used to recursively calculate the DFT of sequences of any length (that is a power of 2).

FFT has some limitations:

- The input sequence needs to be **equally spaced** (although this sometimes can be mitigated using interpolation).
- The input sequence is assumed to be **periodic** (resulting from the definition of the DFT, which tries to transform the data in a linear combination of sines and cosines).

  > In signal processing, the discrete cosine transform (DCT) is often used instead of DFT.
  > It does not assume the input is periodic, making it more broadly applicable.
  > Furthermore, the DCT of a real signal is also real, making the output more intuitive.

- The sequence is assumed to be a **power-of-2 in length** (specific to our algorithm).

  > The FFT can be generalized to handle sequences of arbitrary length, not only powers of 2.
  > The general FFT algorithm does not only split a sequence into two.
  > Instead, it partitions a sequence in $M \geq 2$ subsets, where M is the smallest prime factor of the length of a sequence, at the current recursion level.
  > So in general, M may vary across different levels of recursion.
  > At each level, an M-fold DFT must be computed with conventional matrix-vector multiplication.
  >
  > This shows that the FFT, as discussed here, will be slow for long sequences whose length is a prime number.
  > For such cases, more advanced implementations exist to efficiently handle

> sequences whose lengths are prime numbers.
> These advanced algorithms still result in an $\mathcal{O}(n \log(n))$ scaling, but with a much larger prefactor, compared to the algorithm discussed here.

10.3 Applications

10.3.1 Signal processing

Analyse which frequencies are in a certain discrete set of data, remove certain frequencies (active noise cancellation), ...

10.3.2 Convolutions

The discrete circular convolution of two **periodic** sequences $\mathbf{u}$ and $\mathbf{v}$ of length n is defined as

$$\{\mathbf{u} * \mathbf{v}\}_m = \sum_{k=0}^{n-1} v_k u_{m-k}, \quad \forall\, m \in \{0, 1, \dots, n-1\}.$$

The periodicity means that $v_k = v_{k+n}$ and $u_k = u_{k+n}$. Still, the arrays in computations only contain values for only one period.

This operation is equivalent to multiplication by a **circulant matrix**

$$\begin{bmatrix} z_0 \\ z_1 \\ \vdots \\ z_{n-2} \\ z_{n-1} \end{bmatrix} = \begin{bmatrix} u_0 & u_{n-1} & u_{n-2} & \cdots & u_1 \\ u_1 & u_0 & u_{n-1} & \cdots & u_2 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ u_{n-2} & \cdots & u_1 & u_0 & u_{n-1} \\ u_{n-1} & \cdots & u_2 & u_1 & u_0 \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-2} \\ v_{n-1} \end{bmatrix}$$

Such a matrix is diagonalized by the DFT, thus:

$$\begin{bmatrix} \hat{z}_0 \\ \hat{z}_1 \\ \vdots \\ \hat{z}_{n-2} \\ \hat{z}_{n-1} \end{bmatrix} = \begin{bmatrix} \hat{u}_0 & 0 & \cdots & \cdots & 0 \\ 0 & \hat{u}_1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & \hat{u}_{n-2} & 0 \\ 0 & \cdots & \cdots & 0 & \hat{u}_{n-1} \end{bmatrix} \begin{bmatrix} \hat{v}_0 \\ \hat{v}_1 \\ \vdots \\ \hat{v}_{n-2} \\ \hat{v}_{n-1} \end{bmatrix}$$

For this reason, it is more efficient to use the FFT algorithm to transform the inputs to the frequency domain, compute one pointwise multiplication, and transform the result back to the time domain.

10.3.3 Autocorrelation

The **autocorrelation** of a real sequence $\mathbf{y}$ expresses the similarity between a sequence and a delayed copy of itself.
It is defined as

$$\mathbf{R}_\ell = \sum_{k=0}^{n-1} \mathbf{y}_k \mathbf{y}_{k-\ell} \quad \forall\, l \in \{0, 1, \dots, n-1\},$$

We recognize that this is a convolution of the sequence with a **reversed copy** of itself, i.e. $\mathbf{y}_{k-\ell}$ instead of $\mathbf{y}_{\ell-k}$, and thus can be calculated efficiently using FFTs as

$$\mathbf{R} = \mathcal{F}^{-1}\left(\mathcal{F}(\mathbf{y})(\mathcal{F}(\mathbf{y})^*)\right) = \mathcal{F}^{-1}\left(\|\mathcal{F}(\mathbf{y})\|^2\right)$$

where * denotes a complex conjugate. The complex conjugation is due to the reversal of the second convolution argument in the time domain.

If we want to get the **autocovariance** instead of the **autocorrelation**, we need to *demean* (i.e. subtracting the mean of the sequence) the sequence:

$$\mathbf{K} = \mathcal{F}^{-1}\left(\|\mathcal{F}(\mathbf{y} - \overline{\mathbf{y}})\|^2\right)$$

It is also common to further divide by the *variance* of the sequence, which is conveniently calculated from the first entry in the Fourier transformed sequence.

$$\rho = \frac{\mathbf{K}}{\sigma_y^2}$$

(There is not proper name for the latter quantity. It is often called autocorrelation or autocovariance, while it is actually neither.)

10.3.4 Fast polynomial multiplication

To find the coefficients of the product of two polynomials $f(x) = f_1(x) \cdot f_2(x)$, we need to:

- calculate all the pair-wise products of their respective terms,
- group these products by the order of the resulting mononomial,
- and sum the products within each group.

When we write the coefficients as vectors $\mathbf{f}_1$ and $\mathbf{f}_2$ (ordered from lowest order to largest; i.e. the constant term first), and append zeros such that the vectors have a dimension larger than the degree of f_1 + the degree of f_2, we can write the polynomial product as a convolution, where the coefficients of their product $\mathbf{f}$ are given by:

$$\mathbf{f} = \mathbf{f}_1 * \mathbf{f}_2$$

We saw earlier that we can calculate this in an efficient way using Fast Fourier Transforms:

$$\mathbf{f} = \mathcal{F}^{-1}\big(\mathcal{F}(\mathbf{f}_1)\mathcal{F}(\mathbf{f}_2)\big)$$

10.4 Summary

- DFT
  Used to get frequency data from time/spatial data
  - The highest frequency which you can detect is the Nyquist frequency $\lceil n/2 \rceil f_s/n$

    - The frequency-0 component is always the sum of the input data (the DC component)
- FFT
  Uses a divide and conquer strategy to split the Fourier transform in multiple factor-of-2 smalller DFTs, such that the computational complexity is $O(n \log n)$
    - Input needs to be equally spaced
      Can be solved by interpolation
    - Input needs to be periodic
      One can also use DCT (discrete cosine transform), which doesn't have such a requirement.
    - Input needs to be power-of-2 in length
      You can generalize it to abritrary lengths, but cost will increase
- Applications
    - Signal processing
    - Convolution
    - Auto-correlation
    - Efficient polynomial multiplication

11 Monte Carlo

11.1 Pseudo-random-number-generators (PRNGS)

Computer hardware is designed to carry out purely deterministic arithmetic operations.
Any computational result is therefore not truly random.
Nevertheless, one may design algorithms that produce a sequence of seemingly uncorrelated numbers.

PRNGs generally contain:

1. A `seed`, which determines the entire random sequence.
2. The algorithm has an internal `state`, which is initialized with the seed and is updated after a new number is generated.
   Advanced PRNGs conceptually extend the state in several ways:
   - In general, the state can be an array of numbers, or a bit (0 or 1) array of some size.
   - When the seed contains too few bits of information, it can be padded with other values to fill up the initial state.
   - The random number generated at each sequence may be a function of the state, rather than being equal to the state in the simple case of LCG.
3. A **recurrence relation** is used to derive the next state from the current one.
4. Fixed **parameters** appearing in the algorithm.

PRNGs have the following properties

1. The sequence is **deterministic**: repeating the calculation with the same seed and parameters gives the same result.
2. There is only a **limited number of pseudo-random values**.
   The upper limit is given by 2^N, where N is the number of bits used to represent the state.
   In practice, algorithms have fewer different random numbers:
   - The algorithm may limit the interval of the pseudo-random numbers by construction.
   - Less obviously, the recurrence relation may not be able to visit all possible values.
3. Due to the previous points, **pseudo-random sequences must be periodic** (possibly after some initialization phase).
   When the same state is encountered as before, all subsequent states will be repeated as well.
   Because the number of states is limited, it is unavoidable that, after some time, the same state appears again.

PRNGs are in practice designed with well-defined periods and without any initialization phase.

What we *want* from PRNGs

1. The pseudo-random numbers should be **uniformly distributed**.
    - Ideally, all possible states in a sequence can be visited (from any seed).
    - Several isolated periodic pseudo-random subsequences may exist in the space of all states.
2. There should be **no apparent statistical correlations** between subsequent values.
    - A minimal requirement is that the sequence has a long periods, ideally much longer than the amount of random numbers needed in any application.
    - Also after differentiating the sequence, no obvious correlation should appear. For example, the spacing between two subsequent random numbers should also be pseudo-random.
    - Also when generating N-dimensional vectors of random numbers, these should be uniformly distributed over their space.

These can be tested by pen and paper or PRNG testing suites.

The most modern techinque is Xorshift

11.1.1 Xorshift

They use bit-wise operators:

A complete exposition of the history of PRNG development goes beyond the scope of this course.
It is however worth looking at one popular family of modern PRNGs, namely the Xorshift methods by Marsaglia.

Modern algorithms often make use of binary operators such as `xor` and `shift`, because these are computationally efficient:

- `or` is implemented in Python with the operator `|`. It applies the *bitwise or* to a pair of integers.
  For every corresponding pair of bits, `or` is computed as follows:

In1	In2	Out
0	0	0
0	1	1
1	0	1

In1	In2	Out
1	1	1

When applied to integer numbers, `|` has the following effect.

Python	Decimal	Binary
`a`	5	0101
`b`	12	1100
`a | b`	13	1101

- `xor` is implemented in Python with the operator `^`. It applies the *bitwise exclusive or* to a pair of integers.
  For every corresponding pair of bits, `xor` is computed as follows:

In1	In2	Out
0	0	0
0	1	1
1	0	1
1	1	0

When applied to integer numbers, `^` has the following effect.

Python	Decimal	Binary
`a`	5	0101
`b`	12	1100
`a ^ b`	9	1001

- `shift` shifts all the bits in the binary representation of an integer to the left or the right. Left and right shifts are implemented in Python with the `<<` and `>>` operators, respectively. For example:

Python	Decimal	Binary
	5	00101
`5 << 1`	10	01010
`5 << 2`	20	10100
`5 >> 1`	2	00010

Shifting to the left multiplies by 2, while shifting to the right is a division by 2.
Whenever bits are shifted out of the register, they are discarded.

- `bitroll` is not a low-level operation (and has no official name either), but is popular in modern PRNGs.
  It combines two shift operators to permute bits in an binary number.
  The following table contains some examples for 4-bit integers:

Input	Decimal	roll	Output	Decimal
0010	2	1	0100	4
0101	5	1	1010	10
1001	9	1	0011	3
0011	3	2	1100	12
0011	3	3	1001	9

For Monte Carlo methods, **Xorshift methods (and many others) are practically sufficient and have a low computational cost**.
An older popular method is the Mersenne-Twister algorithm, proposed in 1997, but both its computational performance and quality of randomness are outperformed by more recent algorithms.

Because the **recurrence relations are inherently serial**, one cannot simply generate random numbers in parallel.
Hence, for the sake of computational efficiency, PRNGs are typically implemented in low-level code (not Python).
Vectorization and parallelism is sometimes used to produce multiple streams of parallel random numbers of high-performance applications.

Besides Monte Carlo, another major application of random numbers is **cryptography**. This application comes with additional algorithm requirements, generally related to the predictability of pseudo-random sequences.

11.1.2 Transformations of univariate continuous distributions

Uniformly distributed numbers can be transformed, to sample other continuous univariate distributions. Two common methods are mentioned here for the sake of completeness:

- Inverse transform sampling. Given a random variable X uniformly distributed over $[0, 1]$, it can be transformed to $Y = F_Y^{-1}(X)$, where F_y is the cumulative distribution of the quantity Y.

- The Box-Muller transform is an efficient method for sampling a standard normal distribution.

(See statistics and VFR for more info)

11.2 Monte Carlo basics

The Monte Carlo method relies on the following identity from statistics:

$$\mathrm{E}[g(\mathbf{X})] = \int g(\mathbf{x}) p_{\mathbf{X}}(\mathbf{x}) \, \mathrm{d}\mathbf{x}$$

where:

- $\mathbf{X}$ is a stochastic vector in $\mathbb{R}^n$.
- $\mathbf{x}$ is an (ordinary) vector in $\mathbb{R}^n$.
- $p_{\mathbf{X}}(\mathbf{x})$ is the probability density.
- $p_{\mathbf{X}}(\mathbf{x}) \, \mathrm{d}\mathbf{x}$ is the probability of finding a sample point $\mathbf{X}$ in a region of size $\mathrm{d}\mathbf{x}$ around $\mathbf{x}$.
- $g(\cdot)$ can be any function $\mathbf{x} \mapsto g(\mathbf{x}) \in \mathbb{R}^m$.
- The integral is taken over the entire domain where $p_{\mathbf{X}}(\mathbf{x})$ is non-zero.
- $\mathrm{E}[\cdot]$ stands for "the expectation value, assuming $\mathbf{X}$ is distributed according to the probability density $p_{\mathbf{X}}(\mathbf{x})$.

For many applications, the function g is scalar.
For several examples below, also $\mathbf{X}$ and $\mathbf{x}$ are just scalar quantities.

With the above identity, one may approximate the integral in the right-hand side, just by taking N sample points $\mathbf{X}_i$ from the distribution $p_{\mathbf{X}}$, computing all $g(\mathbf{X}_i)$ and averaging over all results.

$$\int g(\mathbf{x}) p_{\mathbf{X}}(\mathbf{x}) \, \mathrm{d}\mathbf{x} \approx \frac{1}{N} \sum_{i=1}^{N} g(\mathbf{X}_i) = \overline{g(\mathbf{X}_i)}$$

As the expectation value is just the average for discrete data.

11.2.1 Error estimation

The variance on the Monte Carlo estimate is

$$\mathrm{VAR}\left[\frac{1}{N} \sum_{i=1}^{N} g(\mathbf{X}_i) \right] = \frac{1}{N} \mathrm{VAR}[g(\mathbf{X}_i)]$$

where we made use of the variance of a linear combination of two stochastic quantities:

$$\mathrm{VAR}[a\mathbf{X} + b\mathbf{Y}] = a^2 \mathrm{VAR}[\mathbf{X}] + b^2 \mathrm{VAR}[\mathbf{Y}] + 2ab \, \mathrm{COV}[\mathbf{X}, \mathbf{Y}]$$

where $\mathrm{COV}[\mathbf{X}, \mathbf{Y}]$ stands for the covariance of two stochastic quantities.
We get a $1/N^2$ prefactor for bringing the varianc into the multiplication, and pick up an N to bring it in the sum, so the prefactor remains $1/N$. For the variance on the MC estimate, no covariance is taken into account, because we assume independent sample points.

The standard error on the MC estimate is simply the square root of the variance:

$$\text{Std. Err.} = \sqrt{\frac{\text{VAR}[g(\mathbf{X}_i)]}{N}}$$

The error decreases proportionally to $1/\sqrt{N}$ with increasing N.
Hence, by taking a sufficiently large sample, the error can be made arbitrarily small.

Put differently, the number of required points is proportional to $1/(\text{Std. Err.})^2$.
Note that this scaling is independent of the dimension of $\mathbf{X}$ and remains the same for high-dimensional integrals.
This is very different from conventional quadrature methods, where the number of required points scales exponentially with the dimension of $\mathbf{X}$.

Numerical estimates of the standard error can be derived from an unbiased estimate of the variance:

$$\text{VAR}\left[g(\mathbf{X}_i)\right] \approx \frac{1}{N-1}\sum_{i=1}^{N}\left(g(\mathbf{X}_i) - \overline{g(\mathbf{X}_i)}\right)^2$$

11.2.2 Pitfalls

For some choices of g, it becomes infeasible to converge the MC estimate.
In qualitative terms, this happens when:

- g is virtually zero in regions where $p_{\mathbf{X}}$ is significant, and
- g becomes very large where the probability density nearly vanishes.

In such cases, outliers of the distribution will have a large contribution to the average.
In the best case, this results in a large sampling error.
In the worst case, no such outliers are encountered in the sample and the error is not noticed.

A more formal way of describing the problem is that MC has convergence issues when there is a large variance on $g(\mathbf{X}_i)$.
The errors still scale proportionally with $1/\sqrt{N}$, but the prefactor, $\sqrt{\text{VAR}[g(\mathbf{X}_i)]}$, is huge.
In such cases, also the error on the sampling estimate of this prefactor is large (for the same reason), making it difficult to detect this pitfall empirically.

To solve this problem, one should construct another $g(\mathbf{X}_i)$, such that its distribution has no outliers, which is not always straightforward.

11.2.3 Monte Carlo integration

$$I = \int_{\Omega} g(\mathbf{x})\,\mathrm{d}\mathbf{x}$$

where the probability density is missing at first sight.
The integral runs over a finite domain Ω instead of over the entire space.

For this case, one may always insert a uniform distribution and divide out its normalization:

$$I = V_\Omega \int_\Omega g(\mathbf{x}) p_{\mathbf{X}\,U(a,b)}(\mathbf{x})\,\mathrm{d}\mathbf{x}$$

where V_Ω is the volume of the domain (and the inverse of the normalization constant of the uniform distribution).
For this case, the Monte Carlo method is essentially a quadrature method with equal weights and randomized grid points.

Monte Carlo integration is numerically well-behaved as long as g varies smoothly.
In line with the more general case, it may become problematic when g is negligible nearly everywhere in the domain.
For such ill-posed cases, adaptive methods such as the MISER algorithm have been developed to focus on subdomains where g is more informative.

11.3 Discrete Markov-chain Monte Carlo methods

For complex multi-variate distributions, drawing samples is not trivial. For this requirement, markov chains can be used.

11.3.1 Definition of Markov Chain

A **discrete Markov chain** is a stochastic process, i.e. a sequence of stochastic quantities $\{\mathbf{X}_k\}$, in which the probability of observing $\mathbf{x}_{i+1}$ is only determined by:

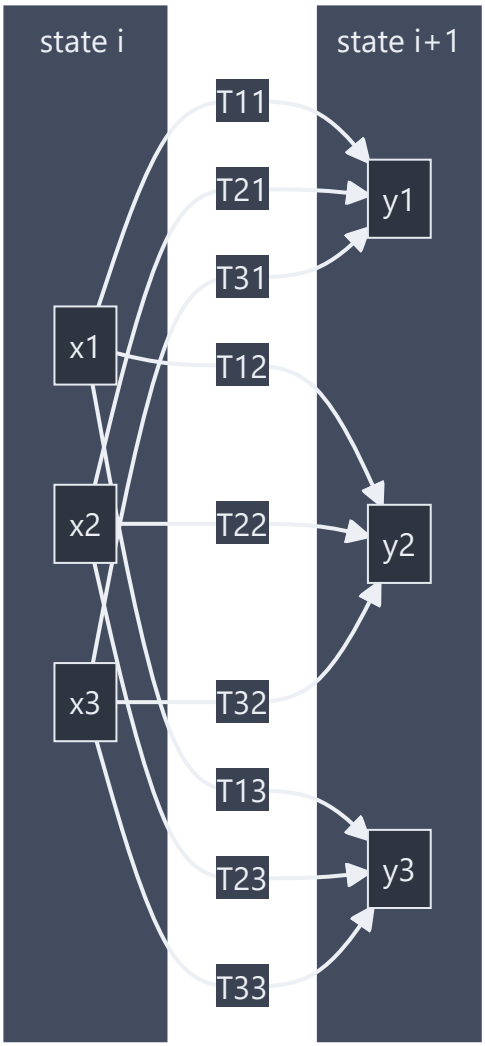
1. the previous state x_i, and
2. the index i.

Formally, one may write

$$p_{\mathbf{X}_{i+1}}(\mathbf{x}_{i+1}) = \int T_{i\to i+1}(\mathbf{x}_{i+1}|\mathbf{x}_i)\, p_{\mathbf{X}_i}(\mathbf{x}_i)\,\mathrm{d}\mathbf{x}_i$$

One can interpret $T_{i\to i+1}$ as a conditional probability density, and it is often called the *transition probability*. This notation is completely retarded though, as we only consider discrete markov chains here, thus the integral will be just a sum over the elements x_i with a weight $T_{i\to i+1}(x_j|x_i)$, which is the probability of going from x_i to x_j if we step from i to $i+1$.

A more accurate representation of what's happening here is the diagram below in three dimensions



Some remarks:

- It is important that the function $T_{i\to i+1}$ is not explicitly dependent on older states such as $\mathbf{x}_{i-1}$ (You can't have arrows from x-s older than state i).
  Such a dependency would result in a non-Markov chain.
- When the function $T_{i\to i+1}$ is the same for all i, the chain is called time-homogeneous and one may just write T.
  This *doesn't* mean that the values of Tij are the same for every i and j! It just means that each Tij doesn't change with time.
- The adjective *discrete* means that the states are labeled by a discrete index i.
  In continuous Markov chains, the states are labeled by a continuous variable, e.g. a time t.

An **important property** of the transition probability is the following normalization:

$$\int T(\mathbf{x}_{i+1}|\mathbf{x}_i)\,\mathrm{d}\mathbf{x}_{i+1} = 1$$

namely, the probability of going *anywhere* from a node in state i is 1 (or the sum of outgoing arrows in x_i^k for each k should be 1).

> **Proof.** One should simply require that the probability density of state $i+1$ is properly normalized when state i is a Dirac delta distribution.

$$1 = \int p_{\mathbf{X}_{i+1}}(\mathbf{x}_{i+1})\,\mathrm{d}\mathbf{x}_{i+1} = \iint T(\mathbf{x}_{i+1}|\mathbf{x}_i)\,\delta(\mathbf{x}_i^0 - \mathbf{x}_i)\,\mathrm{d}\mathbf{x}_{i+1}\,\mathrm{d}\mathbf{x}_i = \int T(\mathbf{x}_{i+1}|\mathbf{x}_i^0)\,\mathrm{d}\mathbf{x}_{i+1}$$

How this proof works is:

- we start from the fact that the state i+1 should be normalized (the total probability of all the nodes in state i+1 should be one)
- we then assume that for each node in i+1 all the probability came from one node in state i, called x_i^0, and the proof follows.

Note that this also shows how to sample a Markov chain in practice.
At the point that we have a sample point $\mathbf{x}_i^0$, its position is fixed, as described by the Dirac delta function, and the next point is simple generated by sampling the transition probability in which $\mathbf{x}_i^0$ appears as a parameter.

11.3.2 Stationary distribution of a discrete Markov chain

As mentioned above, we assume that the Markov chain is *time-homogeneous*. All transition are described by the same transition probability T.

Consider the case of a random variable at a certain time of the markov chain, which is sampled by the distribution $p(\mathbf{x}_1)$. IF this distribution has the following property:

$$\int T(\mathbf{x}_2|\mathbf{x}_1)p(\mathbf{x}_1)\,\mathrm{d}\mathbf{x}_1 = p(\mathbf{x}_2)$$

In words, the transition to the next state leads to the same probability density, then p is called a stationary distribution of the chain.

One can therefore consider **the chain as a random number generator for its corresponding stationary distribution**:

1. Draw an initial sample point with a non-zero probability in the stationary distribution: $\mathbf{x}_1^0$.
   In this case, one can claim that the initial sample point is drawn from the stationary distribution, even if it is improbable.
2. Generate a next sample point, using the transition probability $T(\mathbf{x}_2|\mathbf{x}_1^0)$.
   Because the previous sample point could have been drawn from the stationary distribution, the current sample is also a valid sample point.
3. Repeat step 2, now with $T(\mathbf{x}_3|\mathbf{x}_2^0)$, $T(\mathbf{x}_4|\mathbf{x}_3^0)$, ... until you have enough samples.

- The stationary distribution **does not always exist**.
- When a stationary distribution exists, it is **not necessarily unique**.

11.3.3 Metropolis-Hastings algorithm

Finding the stationary distribution of a chain is generally challenging.
However, the reverse is relatively easy. For a given stationary distribution, there is an

uncountable infinite number of Markov chains.
Once the Markov chain is defined, sampling the stationary distribution is trivial, as explained in the previous section.

The most general approach for constructing a suitable Markov chain is the Metropolis-Hastings algorithm. Many other methods can be seen as special cases of this general framework.

11.3.3.1 Global balance vs detailed balance

- A stationary distribution satisfies the "global balance" condition, that is, after convolution with the transition probability, the same probability is recovered.

  $$\int T(\mathbf{x}_2|\mathbf{x}_1)p(\mathbf{x}_1)\,\mathrm{d}\mathbf{x}_1 = p(\mathbf{x}_2)$$

  This condition does not impose much structure on T and leaves plenty of freedom to decide from where to where the transition probability displaces sample points.

- A more restrictive requirement is called "detailed balance":

  $$T(\mathbf{x}_2|\mathbf{x}_1)p(\mathbf{x}_1) = T(\mathbf{x}_1|\mathbf{x}_2)p(\mathbf{x}_2)$$

  This can be interpreted as follows: for a stationary distribution, the following two probabilities must be equal:

  - The probability density of finding a sample point at $\mathbf{x}_1$ and moving it to $\mathbf{x}_2$.
  - The probability density of finding a sample point at $\mathbf{x}_2$ and moving it to $\mathbf{x}_1$.

  Or in our more intuitive understanding: The probability to go from a node x^k to a node x^l should be the same as the probability to go from x^l to a node x^k, (time is reversible, or T is symmetric, whatever you like).

Both flows of sample points will compensate each other, such that $p(\mathbf{x}_2)$ remains stationary.

One may also show that detailed balance is a sufficient condition for global balance.

> **Proof**:
>
> $$\int T(\mathbf{x}_2|\mathbf{x}_1)p(\mathbf{x}_1)\,\mathrm{d}\mathbf{x}_1 = \int T(\mathbf{x}_1|\mathbf{x}_2)p(\mathbf{x}_2)\,\mathrm{d}\mathbf{x}_1 = p(\mathbf{x}_2)\int T(\mathbf{x}_1|\mathbf{x}_2)\,\mathrm{d}\mathbf{x}_1 = p(\mathbf{x}_2)$$

11.3.3.2 Derivation

The Metropolis-Hastings algorithm defines a Markov chain that satisfies detailed balance for any given stationary distribution.

The transitions in Metropolis-Hastings Markov chain are constructed in two steps:

- **Generation step.** A displacement of $\mathbf{x}_1$ to $\mathbf{x}_2$ is generated with a *proposal* distribution $g(\mathbf{x}_2|\mathbf{x}_1)$. It is important to note that this proposal distribution is not the same as the probablity distribution we ultimately want to sample from. It's just the probability of proposing a state $\mathbf{x}_2$ given state $\mathbf{x}_1$.

- **Acceptance or rejection step.** After constructing $\mathbf{x}_2$ from $\mathbf{x}_1$, it is further assessed and accepted with a probability $A(\mathbf{x}_2, \mathbf{x}_1)$.
  If not accepted, the next state is identical to the current.

The total transition probability is the product of these two probabilities:
$T(\mathbf{x}_2|\mathbf{x}_1) = g(\mathbf{x}_2|\mathbf{x}_1)A(\mathbf{x}_2, \mathbf{x}_1)$.

To find *a* correct expression for the acceptance probability, detailed balance is imposed:

$$g(\mathbf{x}_2|\mathbf{x}_1)A(\mathbf{x}_2, \mathbf{x}_1)p(\mathbf{x}_1) = g(\mathbf{x}_1|\mathbf{x}_2)A(\mathbf{x}_1, \mathbf{x}_2)p(\mathbf{x}_2)$$

and solved towards the ratio of acceptance probabilities:

$$\frac{A(\mathbf{x}_2, \mathbf{x}_1)}{A(\mathbf{x}_1, \mathbf{x}_2)} = \frac{p(\mathbf{x}_2)}{p(\mathbf{x}_1)} \frac{g(\mathbf{x}_1|\mathbf{x}_2)}{g(\mathbf{x}_2|\mathbf{x}_1)}$$

The highest possible acceptance probabilities satisfying this equation are

$$A(\mathbf{x}_2, \mathbf{x}_1) = \min\left(1, \frac{p(\mathbf{x}_2)}{p(\mathbf{x}_1)} \frac{g(\mathbf{x}_1|\mathbf{x}_2)}{g(\mathbf{x}_2|\mathbf{x}_1)}\right)$$

This is called the *Metropolis choice*. This assures that always $A(\mathbf{x}_2, \mathbf{x}_1) = 1$ or $A(\mathbf{x}_1, \mathbf{x}_2) = 1$. Either way the detailed balance condition is statisfied.

This expression becomes more intuitive when the *proposal* distributions become symmetric, i.e. $g(\mathbf{x}_1|\mathbf{x}_2) = g(\mathbf{x}_2|\mathbf{x}_1)$. In that case, one gets:

$$A(\mathbf{x}_2, \mathbf{x}_1) = \min\left(1, \frac{p(\mathbf{x}_2)}{p(\mathbf{x}_1)}\right)$$

This means the following:

- A transition to higher probability density is always accepted.
- A transition to lower probability density is accepted with a probability $p(\mathbf{x}_2)/p(\mathbf{x}_1)$.

To apply this method, one must merely be able to compute the probability density up to an unknown normalization factor. In practically all applications, this is possible.

11.3.3.3 Algorithm

Building on the above derivation, the Metropolis-Hastings algorithm works as follows:

1. Start with an initial state for which the (stationary) probability is non-zero.
   This more than likely will just be a PRN.

2. Generate a step according with the proposal distribution $g(x_2|x_1)$.
   This distribution can for example be a uniform distribution of the current step +- 0.5 or a normal distribution centered at the current step...

3. Compute the acceptance ratio

$$\mathrm{AR} = \frac{p(\mathbf{x}_2)}{p(\mathbf{x}_1)} \frac{g(\mathbf{x}_1|\mathbf{x}_2)}{g(\mathbf{x}_2|\mathbf{x}_1)}$$

   If this is larger than one, accept the step.
   If it is smaller than one, accept the step with probability AR.
   If the step is not accepted, the new state becomes equal to the old state.

4. Repeat steps 2 and 3 until the sample size is sufficient.

11.3.4 Closing remarks

- The Markov chain exhibits a significant amount of **burn-in**.
  In the limit of very long MC chains, burn-in becomes negligible, but usually that is too costly and one resorts to manual removal of the burn-in.
  This is often a subjective choice, for which automatic methods were proposed recently.

- **The Cauchy distribution is an interesting proposal distribution.**
  Due to the heavy tails of the distribution, both large steps (exploration) and small steps (refinement) are considered in one chain.

- Instead of using a Cauchy distribution, one may also work with **adaptive step sizes** in the proposal distribution.
  Naive approaches to control step size based on the acceptance rate of previous iterations, may bias the stationary distribution.
  Use these with care.
  Specialized algorithms have been developed to deal with this problem specifically, e.g. the Affine Invariant Markov Chain implemented in emcee is a good solution for when step sizes are hard to set manually.

One can solve many physical problems with MCs:

- *"For which distribution of physical parameters is is a certain process or result observed?"*

  > For such problems, the parameters can be found by coupling an MHMC algorithm to a computer-controlled experimental setup.
  > This approach is also used to discover new materials, crystallize proteins, discover new pharmaceuticals, etc.
  >
  > Besides MHMC, also genetic algorithms, particle swarm, and other methods are popular for this purpose. Just keep in mind that they don't

> have a stationary distribution, unless they are cast into an MHMC framework.

- *"What is the distribution of model parameters that can explain a distribution of measurements?"*

  > This type of modeling is called Bayesian inference and goes far beyond the scope of this section.

- One **important class of simulations is not considered in this section**, namely simulations of physical systems which are characterized by a probability density. These will be treated in the Statistical Physics course.

11.4 Summary

- Pseudo-random-number-generators
  - Contain:
    - a seed
    - an interal state
    - a reccurence relation on the state to create members
    - fixed parameters appearing in the algorithm
  - Have the following properties:
    - the sequence is deterministic (reproducible using the seed)
    - there's only a limited number of members (typically 2^N)
    - they must be periodic
  - We desire that they:
    - are uniformally distributed
    - dont contain statistical correlation
  - Examples
    - Linear congruential generators (LCG)
      Very old, don't use them
    - Mercenne-Twister algorithms
      Until recently the best way to generate PRNs
    - Xorshift methods
      more recent improvements that should be the industry standard
      E.g. `xoshiro256`

- Transforms
  Used to sample non-uniform distributions from a uniform generator:
  - Inverse transform sampling (for most distributions)
  - Box-Muller transform (for the normal distribution)

- Monte carlo integration

  - Essentially "quadrature" with normalised weights and a randomly chosen samples.
  - The error decreases proportionally to $1/\sqrt{N}$ with increasing N.
  - For some distributions, converge is difficult, e.g. when:
    - g is virtually zero in regions where $p_{\mathbf{X}}$ is significant, and
    - g becomes very large where the probability density nearly vanishes.
  - You can calculate any integral by substituting a uniform distribution and dividing out its normalisation.
  - Monte Carlo integration is numerically well-behaved as long as g varies smoothly. (For randgevallen, see MISER algorithm)

- DMCMC
  Calculate MC integrals using Markov-chains, which allows us to sample from multi-variate complex distributions

  - Metropolis-Hastings algorithm
    The Metropolis-Hastings algorithm defines a Markov chain that satisfies detailed balance for any given stationary distribution.

    1. Start with an initial state for which the (stationary) probability is non-zero.
       This more than likely will just be a PRN.

    2. Generate a step according with the proposal distribution $g(x_2|x_1)$.
       This distribution can for example be a uniform distribution of the current step +- 0.5 or a normal distribution centered at the current step...

    3. Compute the acceptance ratio

    $$\mathrm{AR} = \frac{p(\mathbf{x}_2)}{p(\mathbf{x}_1)} \frac{g(\mathbf{x}_1|\mathbf{x}_2)}{g(\mathbf{x}_2|\mathbf{x}_1)}$$

    If this is larger than one, accept the step.
    If it is smaller than one, accept the step with probability AR.
    If the step is not accepted, the new state becomes equal to the old state.

    4. Repeat steps 2 and 3 until the sample size is sufficient.

  - The Markov chain exhibits a significant amount of **burn-in**.

  - **The Cauchy distribution is an interesting proposal distribution.**