Matematično-fizikalni seminar

June 12, 2025

Contents

1 Numerical Precision and the Gaussian Integral

Frame 1

In many numerical algorithms, a crucial distinction is made between **accuracy** and **precision**. Accuracy refers to how close a measurement or calculation is to the true value. Precision refers to how close repeated measurements or calculations are to each other, regardless of their accuracy.

Imagine you are shooting at a target. Which of the following best describes a result that is **precise, but not accurate**?

[a] All shots are clustered tightly in the bullseye. [b] All shots are spread out randomly all over the target. [c] All shots are clustered tightly together, but in the top-left corner, far from the bullseye.

Choose an answer and go to Frame 2.

Frame 2

Your answer was [a — b — c].

The correct answer is [c]. A tight cluster of shots indicates high precision, but because the cluster is far from the bullseye (the true value), it is not accurate.

a High accuracy and high precision.

b Low accuracy and low precision.

Understanding this distinction is vital in programming, as the way computers store numbers can introduce errors that affect the accuracy of our results, even if the calculations seem precise.

Go to Frame 3.

Frame 3

Let's start with how computers store simple numbers. The representation of natural numbers (positive integers) in binary is likely familiar to you. Each position represents a power of 2.

For example, the binary number 11001101_2 is converted to a decimal number as follows:

$$\begin{aligned}(1 \cdot 2^7) + (1 \cdot 2^6) + (0 \cdot 2^5) + (0 \cdot 2^4) + (1 \cdot 2^3) + (1 \cdot 2^2) + (0 \cdot 2^1) + (1 \cdot 2^0) \\ = 128 + 64 + 0 + 0 + 8 + 4 + 0 + 1 = 205_{10}\end{aligned}$$

This is straightforward. Storing integers is also relatively simple, typically by using the most significant bit as a sign bit (0 for positive, 1 for negative).

But what about numbers with decimal points, like 0.1? This is where the real challenge begins.

Go to Frame 4.

Frame 4

The fundamental problem is that the binary system is base-2. It can perfectly represent any fraction whose denominator is a power of 2. For example:

- $0.5 = 1/2 = 1 \cdot 2^{-1} = 0.1_2$
- $0.25 = 1/4 = 1 \cdot 2^{-2} = 0.01_2$
- $0.75 = 3/4 = 1/2 + 1/4 = 0.11_2$

However, many common decimal fractions, like $0.1 = 1/10$, do not have a denominator that is a power of 2. This means they cannot be represented with a finite number of binary digits. They become repeating fractions in binary, just like 1/3 is a repeating fraction (0.333...) in decimal.

To handle this and to represent a huge range of numbers (from very small to very large), computers use a standardized system for "floating-point" numbers. The most common standard is **IEEE 754**.

Go to Frame 5.

Frame 5

Let's look at the IEEE 754 standard for a 32-bit (single precision) floating-point number, often called a 'float'. The 32 bits are divided into three parts:

- **Sign (S):** 1 bit. 0 for positive, 1 for negative.
- **Exponent (E):** 8 bits. Stores the power of 2.
- **Fraction (F) or Mantissa:** 23 bits. Stores the significant digits of the number.

```
 S | EEEEEEEE | FFFFFFFFFFFFFFFFFFFFFFF
[1]|[  8 bits  ]|[        23 bits        ]
```

How are these three parts combined to represent a number?

Go to Frame 6 to see the formula.

Frame 6

For normal numbers, the value V is given by the formula:

$$V = (-1)^S \times 2^{(E-127)} \times (1.F)_2$$

Let's break this down:

- $(-1)^S$: This just sets the sign.
- $2^{(E-127)}$: The exponent is stored with a **bias** of 127. To get the true exponent, you must subtract 127 from the value stored in the E bits. This allows the exponent to represent both positive and negative powers.
- $(1.F)_2$: This is the mantissa. The '1.' is an *implicit leading bit*. It is not stored but is assumed to be there for all normal numbers, effectively giving us 24 bits of precision for the price of 23. F represents the fractional part in binary.

If you see the formula '(1 + B)' from the slides, 'B' is the value of the mantissa interpreted as a binary fraction. For example, if the mantissa bits 'F' are '1010...', then the full term is $(1.101)_2 = 1 + 1/2 + 0/4 + 1/8$.

Go to Frame 7 to see what happens in special cases.

Frame 7

The IEEE 754 standard reserves certain exponent values for special numbers.

- If Exponent = 255 (all 1s) and Fraction = 0, the number represents ±**Infinity**.

- If Exponent = 255 (all 1s) and Fraction $\neq 0$, the value is **NaN** (Not a Number), used for results of invalid operations like 0/0.
- If Exponent = 0 and Fraction = 0, the number is **0**.
- If Exponent = 0 and Fraction $\neq 0$, the numbers are **sub-normal** (or denormal). These are special numbers that don't have the implicit leading 1, allowing representation of values closer to zero than normal numbers would allow.

This careful design allows for a robust system to handle many computational situations.

Go to Frame 8.

Frame 8

Because of this finite representation, there are "gaps" between representable numbers. The size of this gap changes with the magnitude of the number.

Functions like 'std::nextafter' in C++ or 'numpy.nextafter' in Python let you find the very next representable floating-point number after a given one.

Consider the numbers 1.0 and 1000.0. Is the gap (the difference) between 1.0 and the next representable float *smaller than, larger than, or the same as* the gap for 1000.0?

[a] Smaller than [b] Larger than [c] The same as

Choose an answer and go to Frame 9.

Frame 9

Your answer was [a — b — c].

The correct answer is [a] Smaller than. The gap between representable numbers grows as the magnitude of the numbers grows. This is because the exponent part of the float effectively scales the step size defined by the last bit of the mantissa. For 1.0, the exponent is small, so the steps are small. For 1000.0, the exponent is larger, so the steps are larger.

This leads to some important lessons for programming.

Go to Frame 10.

Frame 10

Here are the key takeaways for any programmer, especially in physics and engineering:

1. **Precision is Finite:** 'float' (single precision) has about 7 decimal digits of precision. 'double' (64-bit) has about 16. Choose the one appropriate for your problem.
2. **Never test floats for exact equality.** Because of representation errors (like with 0.1), a calculation you expect to result in 'a == b' might fail.

Instead of 'if (a == b)', what is a much safer way to compare two floating-point numbers 'a' and 'b'? _____

Go to Frame 11 for the answer.

Frame 11

A safer way to compare floats 'a' and 'b' is to check if their difference is smaller than some tiny tolerance, called an epsilon (ϵ).

```
if (abs(a - b) < epsilon) {
    // Treat them as equal
}
```

where ‘epsilon‘ might be a small number like ‘1e-7‘.

Now that we are aware of these precision issues, let’s tackle the main task from the presentation: programming the Gaussian integral.

Go to Frame 12.

Frame 12

Task: The Gaussian Integral (Error Function)

The goal is to accurately program the error function, ‘erf(z)‘, which is defined as:

$$\text{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$$

This function is monotonically increasing, is 0 at $z = 0$, and approaches 1 as $z \to \infty$.

There is no simple, closed-form solution to this integral. We must approximate it. We’ll explore two common methods.

Go to Frame 13.

Frame 13

Method 1: Taylor Series Expansion

A well-known approach is to expand the exponential term, e^{-t^2}, as a Taylor series and then integrate term-by-term. This gives:

$$\text{erf}(z) = \frac{2}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{(-1)^n z^{2n+1}}{n!(2n+1)}$$

This is a power series in z. Based on how power series work, for what range of z values would you expect this approximation to be most useful and efficient?

[a] Very large values of z. [b] Values of z near 0 (small values). [c] It works equally well for all values of z.

Choose an answer and go to Frame 14.

Frame 14

Your answer was [a — b — c].

The correct answer is [b]. Taylor series are expansions around a point (in this case, $z = 0$) and are most accurate near that point. For small z, the terms get very small very quickly, and you only need a few terms for a good approximation. For large z, you would need a huge number of terms, making it very inefficient.

So, how do we handle large z?

Go to Frame 15.

Frame 15

Method 2: Asymptotic Series

For large values of z, a different kind of series, called an asymptotic series, is much better. We often use it for the complementary error function, $\text{erfc}(z) = 1 - \text{erf}(z)$. Through a process of repeated integration by parts, one can arrive at the following approximation:

$$\text{erfc}(z) \approx \frac{e^{-z^2}}{z\sqrt{\pi}} \left[1 - \frac{1}{2z^2} + \frac{1 \cdot 3}{4z^4} - \frac{1 \cdot 3 \cdot 5}{8z^6} + \dots\right]$$

Asymptotic series have a peculiar property: they do not converge. After a certain point, the terms start getting larger again. The trick is to sum the terms only up to the smallest term for a given z. The larger z is, the more terms you can use before they start growing.

This method is an excellent approximation only for **large values of z**.

Go to Frame 16.

Frame 16

Method 3: Rational Approximations (Padé)

You have one method for small z (Taylor) and another for large z (Asymptotic). How can you create a single function that works well for all values? One answer is to "glue" them together: use the Taylor series if z is small, and the asymptotic series if z is large.

A more advanced technique is to use a **rational approximation**, such as a Padé approximant. This uses a ratio of two polynomials to approximate the function. The presentation gives a well-known rational approximation:

$$\text{erf}(z) = 1 - (a_1 t + a_2 t^2 + \dots + a_5 t^5) e^{-z^2} + \epsilon(z) \quad \text{where} \quad t = \frac{1}{1 + pz}$$

The coefficients $p, a_1, \dots, a_5$ are carefully chosen constants to minimize the error $\epsilon(z)$ over a wide range.

Go to Frame 17.

Frame 17

Conclusion and Your Task

The goal of the original seminar task was to put these ideas into practice. The challenge is to:

1. Implement the Taylor series approximation for 'erf(z)'.
2. Implement the Asymptotic series approximation for 'erfc(z)'.
3. Compare their results to a trusted, built-in library function for 'erf(z)' (like 'math.erf' in Python).
4. Plot the absolute error, $log|f_{approx} - f_{true}|$, to see where each method is most accurate.
5. Think about how you could combine these methods, or a rational approximation, to create a single, robust function that is accurate for all values of z.

This practical exercise highlights the real-world trade-offs between different numerical methods and the fundamental limitations imposed by floating-point arithmetic.

End of Section.

2 Root Finding and Simple Numerical Integration in 1D

Frame 1

When we perform numerical tasks like finding the roots of functions (where a function equals zero) or integrating them, it is crucial to understand the **scale** of the problem.

In other words, we must ask: for our specific problem, what is "small" and what is "large"?

- In an astronomy problem, a time interval of 1 day might be infinitesimally small.
- In particle physics, 1 picosecond (10^{-12} s) might be a relatively long time.

A useful technique is to write our equations in a **dimensionless form**, for instance by defining a new variable $z = t/\tau$, where τ is a characteristic time scale of the system. In such a system, a value of $z = 1$ is a natural, objective scale.

In this section, we will look at some of the simplest algorithms to understand their basic logic. Many professional software libraries contain far superior algorithms, but often, the simple methods are good enough.

Go to Frame 2.

Finding Roots of Functions in 1D

Frame 2

Our goal is to find all possible solutions (roots) to the equation $f(x) = 0$.

Let's start with a hard truth: there is no universal numerical algorithm that can find *all* the roots of an arbitrary function. However, there are excellent algorithms that can find a root *if we already know it exists within a certain interval* $[a, b]$.

What condition must be true for the function $f(x)$ on the interval $[a, b]$ to guarantee that at least one root lies within it? (Hint: think about the sign of the function at the endpoints.) _____

Go to Frame 3 for the answer.

Frame 3

To guarantee at least one root in the interval $[a, b]$, the function must have opposite signs at the endpoints. That is:

$$f(a) \cdot f(b) < 0$$

If $f(a)$ is positive and $f(b)$ is negative (or vice-versa), and assuming the function is continuous, it must cross the x-axis somewhere between a and b.

So, the first step in root-finding is often to find such an interval. The most robust way to do this is to search the domain of interest with a grid of points, $\{x_i\}$, separated by a small step h, where $x_{i+1} = x_i + h$. We then look for any sub-interval $[x_i, x_{i+1}]$ where the sign of the function changes, i.e., where $f(x_i) \cdot f(x_{i+1}) < 0$.

Once we have found such an interval, how do we "zoom in" on the root?

Go to Frame 4.

Frame 4

The Bisection Method

The simplest and most robust method is the **bisection method**. It works by repeatedly dividing the interval in half.

1. Choose an initial interval $[a_1, b_1]$ where $f(a_1) \cdot f(b_1) < 0$.

2. Calculate the midpoint of the interval: $x = (a_1 + b_1)/2$.

3. Check the sign of the function at the midpoint.

If $f(a_1) \cdot f(x) < 0$, which point becomes the new endpoint for the next, smaller interval? [a] $a_2 = a_1$ [b] $b_2 = b_1$ [c] $b_2 = x$ [d] $a_2 = x$

Choose an answer and go to Frame 5.

Frame 5

Your answer was [a — b — c — d].

The correct answer is [c]. If $f(a_1) \cdot f(x) < 0$, it means the root lies in the first half of the interval, $[a_1, x]$. So we set our new interval to be $[a_2, b_2] = [a_1, x]$.

Conversely, if $f(a_1) \cdot f(x) > 0$, it implies the root is in the second half, and our new interval becomes $[a_2, b_2] = [x, b_1]$.

We repeat these steps (2 and 3) until the size of our interval, $|b_n - a_n|$, is smaller than our desired precision, ϵ. Since the interval size is halved at each step, this method is guaranteed to converge, although it can be slow.

Go to Frame 6.

Frame 6

The Regula Falsi (False Position) Method

The bisection method is robust but doesn't use any information about the function's values, only their signs. The **Regula Falsi** method tries to be smarter.

The basic idea is to draw a straight line (a secant) between the points $(a_1, f(a_1))$ and $(b_1, f(b_1))$. Our next guess for the root, x, is the point where this line crosses the x-axis. The formula for this intersection is:

$$x = \frac{f(b_1)a_1 - f(a_1)b_1}{f(b_1) - f(a_1)}$$

After finding x, the update rule is the same as for bisection: we check the signs to determine the new, smaller interval.

A potential problem with this method is that if the function is very curved, one endpoint may remain "stuck" for many iterations, slowing down convergence.

Go to Frame 7.

Frame 7

The Secant Method

The Secant method is very similar to Regula Falsi, but with a key difference. It also uses a line between two points to find the next guess, but it doesn't require those two points to bracket the root (i.e., we don't need $f(x_n) \cdot f(x_{n-1}) < 0$).

1. Start with two initial guesses, x_0 and x_1.

2. Use the same linear interpolation formula as Regula Falsi to find the next point, x_2:

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

3. For the next iteration, we discard the oldest point, using x_1 and x_2 to find x_3, and so on.

This method is often faster than Regula Falsi, but because it doesn't keep the root bracketed, it is not guaranteed to converge. If the initial guesses are poor or the function is ill-behaved, it can fail.

Go to Frame 8.

Frame 8

The Newton-Raphson Method

What if, in addition to the function value $f(x)$, we also know its derivative, $f'(x)$? The **Newton-Raphson** (or Newton's) method uses this extra information to converge very quickly.

Instead of drawing a secant line between two points, it draws a **tangent line** to the curve at our current guess, x_n. The next guess, x_{n+1}, is where this tangent line intersects the x-axis.

Given that the equation of the tangent line at $(x_n, f(x_n))$ is $t(x) = f(x_n) + f'(x_n)(x - x_n)$, what is the formula for x_{n+1}? (Set $t(x_{n+1}) = 0$ and solve for x_{n+1}). _____

Go to Frame 9 for the answer.

Frame 9

Setting the tangent line equation to zero gives:

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

Solving for x_{n+1}, we get the Newton-Raphson iteration formula:

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

This method can be extremely fast (often converging quadratically). However, like the secant method, it can fail to converge if the initial guess is poor or if the derivative is close to zero near the root. Many robust, professional algorithms (like 'scipy.optimize.brentq') combine the safety of the bisection method with the speed of these faster methods.

Go to Frame 10.

Simple Numerical Integration in 1D

Frame 10

We can formally interpret the definite integral of a function in 1D, $\int_a^b f(x)dx$, as the area under the curve $y = f(x)$ over the interval $[a, b]$. The methods for numerically calculating this are often called **quadrature rules**.

The basic idea is to divide the interval $[a, b]$ into N narrow strips of equal width $h = (b-a)/N$. We then approximate the area of each strip and sum them up.

The simplest method is the **midpoint rule**. How do we approximate the area of the strip from x_n to x_{n+1} using this rule?

$$\int_{x_n}^{x_{n+1}} f(x)dx \approx \ ?$$

Go to Frame 11 for the answer.

Frame 11

The midpoint rule approximates the area of the strip by the area of a rectangle whose height is the function's value at the middle of the interval:

$$\int_{x_n}^{x_{n+1}} f(x)dx \approx f\left(\frac{x_n + x_{n+1}}{2}\right) \cdot h$$

Summing these rectangles is equivalent to calculating the area of a histogram of the function.

A slightly better approximation is the **trapezoidal rule**. Instead of a rectangle, we approximate the area of each strip with a trapezoid formed by connecting the points $(x_n, f(x_n))$ and $(x_{n+1}, f(x_{n+1}))$ with a straight line.

What is the formula for the area of one such trapezoid? _____

Go to Frame 12 for the answer.

Frame 12

The area of a trapezoid is the average of the parallel side lengths times the width. For our strip, this is:

$$\int_{x_n}^{x_{n+1}} f(x)dx \approx \frac{f(x_n) + f(x_{n+1})}{2} \cdot h$$

When we sum these trapezoids over the entire interval $[a, b]$, we get the composite trapezoidal rule:

$$\int_a^b f(x)dx \approx h\left[\frac{f_0}{2} + f_1 + f_2 + \cdots + f_{N-1} + \frac{f_N}{2}\right]$$

where $f_n = f(x_n)$. This rule is exact for linear functions. The error on a single interval is on the order of $O(h^3)$, while the total (global) error is $O(h^2)$.

Go to Frame 13.

Frame 13

Higher-Order Formulas

We can achieve much better results with the same number of function evaluations by using higher-order interpolation.

Simpson's Rule fits a parabola (a quadratic polynomial) through three adjacent points. This requires evaluating the integral over a "double-strip" of width $2h$. The formula for one such segment is:

$$\int_{x_0}^{x_2} f(x)dx \approx \frac{h}{3}[f_0 + 4f_1 + f_2]$$

By a fortunate coincidence, this formula is exact not just for quadratic polynomials, but for cubic polynomials as well. This makes it very accurate. The error for the entire integral scales as $O(h^4)$.

Even higher-order rules exist, like Bode's rule (using 5 points), which improve accuracy further. The general class of these methods is called Newton-Cotes formulas. Advanced methods like Gauss-Jacobi quadrature use non-uniformly spaced points to achieve optimal accuracy for a given number of function evaluations.

This concludes our quick overview of root-finding and integration. The task is now to apply these methods to solve practical problems, like those suggested in the slides (finding polynomial roots or integrating functions like x^3e^{-x}).

End of Section.

3 Monte Carlo Integration

Frame 1

This section introduces **Monte Carlo (MC) methods**. The primary goal is to learn how to perform integration and simulate processes using randomness.

The core idea of these methods predates modern computers. Originally, a "computer" was a person, often working in a large room with others, performing calculations by hand. The electronic "digital computer" later took over these tasks, with early applications in critical areas like codebreaking during World War II at Bletchley Park.

The foundation of all Monte Carlo methods is the ability to generate random numbers.

Go to Frame 2.

Random Numbers

Frame 2

The first lesson of computational science is a crucial one: **truly random numbers do not exist in programming**.

Computers are deterministic machines. They cannot create genuine randomness. Instead, we use algorithms that generate sequences of numbers that *appear* to be random. These are called **pseudo-random numbers**.

What properties do these pseudo-random number generator (PRNG) algorithms have?

- They produce numbers that are typically uniformly distributed in the interval $[0, 1]$. This means any number between 0 and 1 has an equal chance of being generated.
- They are completely **deterministic**.
- They are inherently **periodic**.

Go to Frame 3.

Frame 3

Because PRNGs are deterministic algorithms, if you start one with the same initial value, you get the exact same sequence of numbers. This initial value is called the **random seed**.

Why is using a specific seed useful in scientific computing? [a] It ensures the results are truly random. [b] It makes the program run faster. [c] It allows for reproducible results and easier debugging.

Choose an answer and go to Frame 4.

Frame 4

Your answer was [a — b — c].

The correct answer is [c]. Seeding the random number generator allows anyone to reproduce your exact simulation or calculation, which is essential for verifying scientific results. It also helps immensely when debugging, as the "random" behavior is predictable.

A good PRNG should have two key features:

- It should be **fast**.

- It should have a very **long period** before the sequence of numbers repeats.

One of the most famous and widely used PRNGs is the **Mersenne Twister**, which has an enormous period of $2^{19937} - 1$.

Go to Frame 5.

Frame 5

While the numbers from a PRNG are not truly random, they are designed to be **statistically independent**. This is a critical property, as it allows us to treat the generated numbers as if they were outcomes of a random process and apply the tools of statistics to them.

There is another category of numbers called **quasi-random numbers**. Unlike pseudo-random numbers, which aim to mimic randomness, quasi-random sequences are designed to cover a space as uniformly as possible. They are intentionally *not* independent.

Which type of number is essential for simulations that rely on statistical analysis? [a] Pseudo-random [b] Quasi-random

Go to Frame 6 for the answer.

Frame 6

The correct answer is [a] Pseudo-random. Because they are statistically independent, they correctly model random processes. Quasi-random numbers are useful in other areas, like computer graphics (CGI) for generating evenly spaced patterns, but not for statistical simulation.

Now that we understand the tool (pseudo-random numbers), let's use it for integration.

Go to Frame 7.

Monte Carlo Integration

Frame 7

The simplest way to understand Monte Carlo integration is by calculating the area of a shape. This is often called the "hit-or-miss" method.

Let's find the area of a unit circle (radius $r = 1$). We can do this by:

1. Enclosing the circle in a simple shape we know the area of, like a square. Let's use a square that goes from -1 to 1 in both x and y. The area of this square is $S_0 = (2r)^2 = 4$.
2. Generating a large number, N, of random points (x, y) uniformly inside this square.
3. Counting how many points land inside the circle (N_{in}) versus outside (N_{out}).

What is the probability, P_{in}, that a randomly thrown point will land inside the circle? (Hint: It's the ratio of the areas). _____

Go to Frame 8 for the answer.

Frame 8

The probability is the ratio of the area of the circle ($S_k = \pi r^2 = \pi$) to the area of the square ($S_0 = 4$):

$$P_{in} = \frac{S_k}{S_0} = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4}$$

We can estimate this probability from our simulation by the fraction of points that landed inside:

$$\hat{P}_{in} = \frac{N_{in}}{N_{in} + N_{out}} = \frac{N_{in}}{N_{total}}$$

By equating the theoretical probability with our experimental estimate, we can solve for the unknown area, $S_k = S_0 \cdot \hat{P}_{in}$. We can even use this to estimate $\pi \approx 4 \cdot \hat{P}_{in}$.

Go to Frame 9.

Frame 9

The error in this Monte Carlo estimation decreases with the number of trials, N. The error is statistical (binomial) in nature and scales as $1/\sqrt{N}$. This slow convergence is a "painful fact" of MC methods, but the great advantage is its simplicity and generality.

The "hit-or-miss" method can be used for a shape of *any* complexity, as long as we can define a boundary (a test for inside/outside) and enclose it in a simple volume.

This idea can be easily extended to higher dimensions to calculate volumes. What two things would you need to calculate the volume of a complex 3D object using this method? 1. ___2. ___

Go to Frame 10 for the answer.

Frame 10

To calculate a 3D volume with the Monte Carlo hit-or-miss method, you need: 1. A simple, enclosing volume whose volume you know (e.g., a cube). 2. A criterion to test if a random point (x, y, z) is inside or outside the object.

Go to Frame 11.

Frame 11

Mean Value Method for 1D Integrals

Now let's apply this to a 1D integral, $\int_a^b f(x)dx$. Instead of "hit-or-miss," we can use the concept of the **average value** of a function. The integral can be expressed as the average value of $f(x)$ over the interval, multiplied by the length of the interval.

$$\int_a^b f(x)dx = \langle f(x) \rangle \cdot (b - a)$$

We can estimate the average value, $\langle f(x) \rangle$, by sampling the function at N random points, x_i, chosen uniformly from $[a, b]$ and taking their mean:

$$\langle f(x) \rangle \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i)$$

Therefore, our Monte Carlo estimate of the integral is:

$$\int_a^b f(x)dx \approx \frac{b - a}{N} \sum_{i=1}^{N} f(x_i)$$

The error in this method also scales as $1/\sqrt{N}$. This "mean value" method is generally more efficient than the "hit-or-miss" method.

Go to Frame 12.

Frame 12

Importance Sampling

The mean value method samples points uniformly. But what if our function $f(x)$ has a large peak in a narrow region? A uniform sampling might miss this peak, leading to a poor estimate.

We can improve our estimate by sampling more points where the function is large. This is the idea behind **importance sampling**.

We rewrite the integral by introducing a probability distribution function (PDF), $w(x)$, that we will sample from:

$$\int_a^b f(x)dx = \int_a^b \frac{f(x)}{w(x)} w(x)dx$$

This looks more complicated, but it is powerful. The integral is now the expected value of the function $g(x) = f(x)/w(x)$ for random variables x_i drawn from the distribution $w(x)$.

$$\int_a^b f(x)dx \approx \frac{1}{N}\sum_{i=1}^{N} g(x_i) = \frac{1}{N}\sum_{i=1}^{N} \frac{f(x_i)}{w(x_i)}$$

What would be the *ideal* choice for the sampling distribution $w(x)$ to minimize the error (variance) of the estimate? _____

Go to Frame 13 for the answer.

Frame 13

The ideal choice for $w(x)$ would be a distribution that is shaped just like the function we are trying to integrate, but normalized. That is, $w(x) = f(x)/\int f(x)dx$.

If we could do this, then $g(x) = f(x)/w(x)$ would be a constant! The variance of a constant is zero, and our integral estimate would be perfect with just a single sample.

Of course, if we knew $\int f(x)dx$ to create the perfect $w(x)$, we wouldn't need to do the integration in the first place! The practical idea is to choose a simple PDF $w(x)$ that we *can* sample from and that *approximates* the shape of $f(x)$. This concentrates the sample points in the important regions and dramatically reduces the error for the same number of samples N.

Go to Frame 14.

Monte Carlo Simulation

Frame 14

Besides integration, MC methods are used to simulate events that are distributed according to a specific probability distribution, $w(x)$. For example, simulating the energy of particles from a radioactive decay.

The simplest method is again **hit-or-miss (or rejection sampling)**.

1. Find a maximum value, h, of the distribution $w(x)$ in the domain $[a, b]$.
2. Generate a uniform random x-coordinate, x_i, in $[a, b]$.
3. Generate a uniform random y-coordinate, $\rho \cdot h$, in $[0, h]$.
4. If the point $(x_i, \rho \cdot h)$ is *below* the curve $w(x)$, we "accept" x_i as our event. If it is above, we "reject" it and try again.

This is simple but can be wasteful if the function has sharp peaks.

Go to Frame 15.

Frame 15

Inverse Transform Sampling

A more powerful, unitary method (which gives a valid event for every random number) is **inverse transform sampling**. This method relies on the cumulative distribution function (CDF).

The CDF, $F(x)$, is the integral of the PDF: $F(x) = \int_a^x w(z)dz$. The CDF runs from 0 to some maximum value (1, if $w(x)$ is normalized).

The method works by:

1. Generating a uniform random number, ρ, between 0 and 1.
2. Setting this equal to the normalized CDF.
3. Solving for x by inverting the function: $x = F^{-1}(\rho)$.

This method is extremely efficient if the inverse CDF, F^{-1}, is known analytically. For example, to generate events from an exponential distribution $w(t) = \frac{1}{\tau}e^{-t/\tau}$, the corresponding random variable is $t = -\tau \ln(1 - \rho)$.

This concludes our quick overview of Monte Carlo methods.

End of Section.

4 Finding Function Extrema and Fitting Functions to Measurements

Frame 1

Today we will learn procedures for finding the **extrema** (maxima or minima) of functions in one dimension. As with all numerical methods, the complexity of these procedures increases with the dimensionality of the space in which the function is defined. Because of this, the most robust and reliable methods are those developed for 1D.

It also turns out that the process of **fitting a function to data** (also known as modeling or curve fitting) is directly related to finding extrema. We will learn about this connection as well.

Go to Frame 2.

Finding Extrema in 1D

Frame 2

Finding the extrema of a function $f(x)$ in one dimension can be divided into two sub-problems, based on what we know about the function's derivative, $f'(x)$.

Case 1: The derivative $f'(x)$ is known. If we have an analytical expression for the derivative, the search for an extremum is reduced to a problem we have already solved. What problem is that? (Hint: What is true about the derivative at a maximum or minimum?) _____

Go to Frame 3 for the answer.

Frame 3

If the derivative $f'(x)$ is known, finding an extremum reduces to **finding the roots** of the derivative, i.e., solving the equation $f'(x) = 0$. We already know how to do this using methods like bisection or Newton-Raphson.

Case 2: The derivative $f'(x)$ is unknown. There are many reasons why we might not know the derivative. For example, $f(x)$ itself might be the result of a complex numerical procedure (like an integration), or the derivative might simply be too difficult or computationally expensive to calculate.

In our exercises, we will focus on this second, more challenging case.

Go to Frame 4.

Frame 4

Before we begin, we must acknowledge a crucial fact: there is **no general method that can automatically find the global extremum** (the absolute lowest or highest point) of an arbitrary function. Any iterative method can get "stuck" in a *local* extremum.

Therefore, finding the global extremum often requires further investigation, such as trying different starting points or comparing several local extrema.

Also, almost all standard numerical procedures are designed to find the **minimum** of a function. How can we use a minimization algorithm to find the *maximum* of a function $f(x)$? _____

Go to Frame 5 for the answer.

Frame 5

To find the maximum of a function $f(x)$, we can simply find the minimum of the function $-f(x)$. The x value that minimizes $-f(x)$ is the same x that maximizes $f(x)$.

One final point: always remember to check the function's values at the boundaries of your interval of interest! The global extremum might be at an endpoint, not where the derivative is zero.

Go to Frame 6.

Frame 6

To numerically identify the existence of a minimum in a certain region, we need at least three points, let's call them A, B, and C, with known function values $f(A)$, $f(B)$, and $f(C)$.

What conditions must these three points satisfy to "bracket" a minimum? (Hint: Think about the ordering of the points and their corresponding function values.) _____

Go to Frame 7 for the answer.

Frame 7

To bracket a minimum, we need a triplet of points (A, B, C) such that:

1. The points are ordered: $A < B < C$.
2. The middle point is the lowest: $f(B) < f(A)$ and $f(B) < f(C)$.

If these conditions are met, we know that at least one local minimum must exist within the interval $[A, C]$. The point B serves as our first approximation of the minimum.

Our goal is now to refine this initial guess and shrink the interval $[A, C]$ until its width is smaller than some desired precision, ϵ.

Go to Frame 8.

Frame 8

Golden Section Search

We have an initial bracketing interval $[A, C]$ with a point B inside. We need to choose a new point, D, to shrink the interval. A moment's thought reveals we should place the new point inside the *larger* of the two sub-intervals, $[A, B]$ or $[B, C]$.

Suppose we place a new point D in the interval $[B, C]$. We now have four points A, B, D, C. We evaluate $f(D)$.

- If $f(D) < f(B)$, our new bracketing triplet becomes (B, D, C).
- If $f(D) > f(B)$, our new bracketing triplet becomes (A, B, D).

In either case, we have a new, smaller interval that brackets the minimum. We can repeat this process until the interval is small enough.

But where exactly should we place point D to be most efficient?

Go to Frame 9.

Frame 9

The most elegant and efficient way to choose the new point uses the **Golden Ratio**, φ. The golden ratio has a unique self-similarity property that simplifies the search.

The golden ratio φ is defined by the relation $\frac{a+b}{a} = \frac{a}{b} = \varphi$. This leads to the quadratic equation $\varphi^2 - \varphi - 1 = 0$.

What is the value of φ?

$$\varphi = \frac{1 \pm \sqrt{1 - 4(1)(-1)}}{2} = \frac{1 \pm \sqrt{5}}{2}$$

Since it's a ratio of lengths, we take the positive root. So, $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.61803...$

Go to Frame 10.

Frame 10

The power of the golden section search comes from placing the interior points at a specific fraction of the interval width. Let the initial interval be $[A, C]$ with width $h = C - A$. We choose two interior points, B and D, such that they divide the interval according to the golden ratio.

$$B = A + (1 - 1/\varphi)h = A + h/\varphi^2$$

$$D = A + (h/\varphi)$$

Notice that $1/\varphi = \varphi - 1 \approx 0.618$ and $1/\varphi^2 = (\varphi - 1)^2 \approx 0.382$.

The magic is this: after one step, when we have our new, smaller bracketing triplet (e.g., (A, B, D)), the *old* interior point (B) is now perfectly positioned to be one of the *new* interior points for the next iteration. We only need to compute one new point per step.

At each step, the interval width is reduced by a factor of $1/\varphi$.

Go to Frame 11.

Frame 11

Parabolic Interpolation

An alternative method, which can be much faster, is **parabolic interpolation**. The idea is to fit a parabola through our three bracketing points $(A, f(A))$, $(B, f(B))$, and $(C, f(C))$. Since a parabola has a unique minimum, we can analytically calculate the location of that minimum and use it as our next guess, D.

This method is very fast if the function is well-approximated by a parabola near its minimum. What is a potential downside of this method? (Hint: Think about functions that are not "parabola-like".)

Go to Frame 12 for the answer.

Frame 12

A major downside of pure parabolic interpolation is that it can be unreliable. If the function is not well-behaved (e.g., very flat, or the points are collinear), the calculation for the minimum can be unstable or even send the next guess outside the bracketing interval.

A robust algorithm needs to check if the new point is reasonable and fall back to a safer method if it isn't.

Go to Frame 13.

Frame 13

Brent's Method

Brent's method is a hybrid algorithm that combines the best of both worlds. It attempts to use fast parabolic interpolation whenever possible. However, at each step, it checks if the step taken was productive.

If the parabolic step is too small or fails to improve the situation, the algorithm automatically switches to the slower, but guaranteed-to-converge, golden section search for one or more steps.

This combination of speed and robustness makes Brent's method a standard choice for 1D minimization in many scientific libraries (e.g., 'scipy.optimize.brent').

Go to Frame 14.

Fitting Functions to Data

Frame 14

One of the most common applications of minimization is fitting a model function to a set of measured data points. Suppose we have a set of data points (x_i, y_i) with uncertainties σ_i, and a model function $y = f(x, \vec{\alpha})$ that depends on a set of unknown parameters $\vec{\alpha}$.

Our goal is to find the values of the parameters $\vec{\alpha}$ that make the function "best fit" the data. We need a way to quantify what "best" means. The most common method is **least squares fitting**, which uses the chi-squared (χ^2) statistic as an estimator.

What do you think we do with the χ^2 value to find the best-fit parameters?

$$\chi^2(\vec{\alpha}) = \sum_i \frac{(y_i - f(x_i, \vec{\alpha}))^2}{\sigma_i^2}$$

[a] Maximize it [b] Minimize it [c] Find where it equals zero

Go to Frame 15.

Frame 15

Your answer was [a — b — c].

The correct answer is [b] Minimize it. The χ^2 value represents the sum of squared, uncertainty-weighted differences between the data and the model. A smaller χ^2 means a better agreement. Therefore, finding the best-fit parameters $\vec{\alpha}$ is a minimization problem.

If our model function is a straight line, $f(x, \alpha_1, \alpha_2) = \alpha_1 x + \alpha_2$, this process is called **linear regression**. But the principle applies to any model function, linear or not. We can use multi-dimensional versions of algorithms like Brent's method (e.g., the Downhill Simplex method) to find the minimum of the χ^2 function and thus the best-fit parameters. Professional libraries like 'scipy.optimize.curve_fit' provide powerful tools for this task.

This concludes our overview of function minimization and fitting.

End of Section.

5 Matrix Diagonalization, Eigenvalues, and Eigenvectors

Frame 1

This section tackles a very common problem in mathematics and physics: finding the eigenvalues and eigenvectors of a matrix. For a given square matrix $\mathbf{A}$, we are looking for special vectors $\mathbf{x}$ and scalars λ that satisfy the eigenvalue equation:

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

Here, λ is an **eigenvalue** and $\mathbf{x}$ is the corresponding **eigenvector**. This means that when the matrix $\mathbf{A}$ acts on its eigenvector $\mathbf{x}$, the result is simply the same vector scaled by the eigenvalue λ.

We will focus on finding eigenvalues AND their corresponding eigenvectors.

Go to Frame 2.

Frame 2

The eigenvalue equation can be rewritten as $(\mathbf{A} - \lambda \mathbf{I})\mathbf{x} = 0$, where $\mathbf{I}$ is the identity matrix. For this equation to have a non-trivial solution (i.e., $\mathbf{x} \neq 0$), the matrix $(\mathbf{A} - \lambda \mathbf{I})$ must be singular. What does this imply about its determinant?

[a] $\det(\mathbf{A} - \lambda \mathbf{I}) > 0$ [b] $\det(\mathbf{A} - \lambda \mathbf{I}) < 0$ [c] $\det(\mathbf{A} - \lambda \mathbf{I}) = 0$

Choose an answer and go to Frame 3.

Frame 3

Your answer was [a — b — c].

The correct answer is [c]. A non-trivial solution exists only if $\det(\mathbf{A} - \lambda \mathbf{I}) = 0$. This equation is called the **characteristic polynomial**. Finding its roots gives the eigenvalues λ. While this analytical approach works for small matrices, it becomes impractical for dimensions larger than about 4x4. We need numerical methods.

Our focus will be on **symmetric matrices** ($\mathbf{A} = \mathbf{A}^T$). Symmetric matrices have two very nice properties:

- Their eigenvalues are always **real**.
- They can be diagonalized by an **orthogonal matrix Z** (where $\mathbf{Z}^{-1} = \mathbf{Z}^T$).

The columns of this matrix $\mathbf{Z}$ are the eigenvectors of $\mathbf{A}$.

Go to Frame 4.

Iterative Methods

Frame 4

The Power Iteration Method

The simplest iterative approach for finding the largest eigenvalue is the **power method**.

1. Start with a random initial vector, $\mathbf{x}^{(0)}$.
2. Normalize this vector.

3. Repeatedly apply the matrix $\mathbf{A}$ to the vector in an iterative procedure:

$$\mathbf{y}^{(i+1)} = \mathbf{A}\mathbf{x}^{(i)} \qquad \text{and} \qquad \mathbf{x}^{(i+1)} = \frac{\mathbf{y}^{(i+1)}}{|\mathbf{y}^{(i+1)}|}$$

The second step is re-normalization to prevent the vector's magnitude from growing uncontrollably.

As the number of iterations i increases, what does the vector $\mathbf{x}^{(i)}$ converge to? _____

Go to Frame 5 for the answer.

Frame 5

The vector $\mathbf{x}^{(i)}$ converges to the **eigenvector corresponding to the largest eigenvalue**.

The largest eigenvalue itself, λ_{max}, can then be found using the Rayleigh quotient:

$$\lambda_k = \frac{(\mathbf{x}^{(k)})^T(\mathbf{A}\mathbf{x}^{(k)})}{(\mathbf{x}^{(k)})^T\mathbf{x}^{(k)}} = \frac{\mathbf{x}^{(k)} \cdot (\mathbf{A}\mathbf{x}^{(k)})}{|\mathbf{x}^{(k)}|^2}$$

This method can be slow and may fail if the initial guess $\mathbf{x}^{(0)}$ is unlucky (e.g., orthogonal to the desired eigenvector). It can also be adapted to find the smallest eigenvalue by applying power iteration to the inverse matrix, $\mathbf{A}^{-1}$.

How can we find the *other* eigenvalues and eigenvectors?

Go to Frame 6.

Frame 6

Once we have found the largest eigenvalue λ_1 and its corresponding eigenvector $\mathbf{x}_1$, we can find the next one by removing the first one from the matrix. This process is called **deflation**.

Wielandt's deflation constructs a new matrix $\mathbf{A}'$ that has the same eigenvalues as $\mathbf{A}$, except that λ_1 is replaced by 0:

$$\mathbf{A}' = \mathbf{A} - \lambda_1 \mathbf{x}_1 \mathbf{x}_1^T$$

(Here, $\mathbf{x}_1\mathbf{x}_1^T$ is the outer product). We can then apply the power method to $\mathbf{A}'$ to find the next largest eigenvalue. This process can be repeated, but errors accumulate.

Go to Frame 7.

Transformation Methods

Frame 7

A more robust and advanced approach is to find a transformation matrix $\mathbf{Z}$ that diagonalizes $\mathbf{A}$ all at once. The goal is to find an orthogonal matrix $\mathbf{Z}$ such that:

$$\mathbf{Z}^{-1}\mathbf{A}\mathbf{Z} = \mathbf{Z}^T\mathbf{A}\mathbf{Z} = \mathbf{D}$$

where $\mathbf{D}$ is a diagonal matrix whose entries are the eigenvalues of $\mathbf{A}$. The columns of $\mathbf{Z}$ will be the corresponding normalized eigenvectors.

The strategy is to construct $\mathbf{Z}$ as a sequence of simpler orthogonal transformations:

$$\mathbf{Z} = \mathbf{P}_1 \cdot \mathbf{P}_2 \cdot \mathbf{P}_3 \cdots$$

Each transformation $\mathbf{P}_k$ is chosen to make the matrix "more diagonal" until it converges.

Go to Frame 8.

Frame 8

The Jacobi Method

The Jacobi method is an elegant approach that uses a sequence of **plane rotations** to zero out the off-diagonal elements of the matrix.

An arbitrary rotation in N-dimensions that affects only the p-th and q-th coordinates is represented by a matrix $\mathbf{P}_{pq}$ which looks like the identity matrix, except for four elements:

$$(\mathbf{P}_{pq})_{pp} = c, \quad (\mathbf{P}_{pq})_{qq} = c, \quad (\mathbf{P}_{pq})_{pq} = s, \quad (\mathbf{P}_{pq})_{qp} = -s$$

where $c = \cos(\theta)$ and $s = \sin(\theta)$ for some rotation angle θ.

The transformation $\mathbf{A}' = \mathbf{P}_{pq}^T \mathbf{A} \mathbf{P}_{pq}$ only changes the rows and columns p and q of the matrix $\mathbf{A}$. How do we choose the angle θ?

Go to Frame 9 for the answer.

Frame 9

We choose the angle θ specifically to make the off-diagonal element a'_{pq} in the new matrix equal to zero. This leads to the condition:

$$\cot(2\theta) = \frac{a_{qq} - a_{pp}}{2a_{pq}}$$

Solving this for $\cos(\theta)$ and $\sin(\theta)$ gives us the required rotation matrix.

The problem is that the next rotation (to zero out a different element) will generally "mess up" the zero we just created. However, the Jacobi method is guaranteed to converge because the sum of the squares of the off-diagonal elements decreases with every step.

The procedure is to repeatedly "sweep" through all off-diagonal elements, zeroing them out, until the matrix is diagonal to within a desired tolerance.

Go to Frame 10.

Frame 10

The QR Method and Tridiagonalization

A more modern and generally faster approach for finding all eigenvalues is the **QR method**. It is an iterative method based on the **QR decomposition**, where any matrix $\mathbf{A}$ can be decomposed into $\mathbf{A} = \mathbf{QR}$, with $\mathbf{Q}$ being an orthogonal matrix and $\mathbf{R}$ being an upper-triangular matrix.

The iterative process is:

1. Start with $\mathbf{A}_0 = \mathbf{A}$.
2. In step k, decompose $\mathbf{A}_k = \mathbf{Q}_k \mathbf{R}_k$.
3. Construct the next matrix as $\mathbf{A}_{k+1} = \mathbf{R}_k \mathbf{Q}_k$.

What is the relationship between $\mathbf{A}_{k+1}$ and $\mathbf{A}_k$? (Hint: Substitute $\mathbf{R}_k = \mathbf{Q}_k^T \mathbf{A}_k$ into the third step).

Go to Frame 11.

Frame 11

The relationship is a **similarity transformation**:

$$\mathbf{A}_{k+1} = \mathbf{R}_k\mathbf{Q}_k = (\mathbf{Q}_k^T\mathbf{A}_k)\mathbf{Q}_k = \mathbf{Q}_k^T\mathbf{A}_k\mathbf{Q}_k$$

Because it's a similarity transformation, $\mathbf{A}_{k+1}$ has the same eigenvalues as $\mathbf{A}_k$. This iterative process converges to an upper triangular matrix (or a diagonal matrix, if $\mathbf{A}$ is symmetric). The eigenvalues appear on the diagonal.

For general matrices, this process is computationally expensive. The crucial insight is that the method is *much faster* for matrices that are already almost diagonal. The standard procedure is therefore a two-step process:

1. First, use a finite number of transformations (like Householder reflections) to reduce the symmetric matrix to a **tridiagonal form**.
2. Then, apply the QR algorithm to the much simpler tridiagonal matrix to find the eigenvalues.

This two-step process is at the heart of most modern, robust eigenvalue solvers, like those found in NumPy and SciPy.

Go to Frame 12.

General Diagonalization: SVD

Frame 12

What if the matrix is not square, or not symmetric? The concepts of eigenvalues and eigenvectors don't apply in the same way. The generalization of diagonalization for *any* $m \times n$ matrix is the **Singular Value Decomposition (SVD)**.

SVD states that any matrix $\mathbf{A}$ can be decomposed as:

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

- $\mathbf{U}$ is an $m \times m$ orthogonal matrix.
- $\mathbf{V}$ is an $n \times n$ orthogonal matrix.
- $\mathbf{\Sigma}$ is an $m \times n$ diagonal matrix, containing the non-negative **singular values** on its diagonal.

SVD is an extremely powerful and stable numerical tool. It reveals the fundamental action of a matrix as a combination of rotation ($\mathbf{V}^T$), scaling ($\mathbf{\Sigma}$), and another rotation ($\mathbf{U}$). It is used everywhere from data science and machine learning to image compression.

This concludes our quick overview of matrix diagonalization.

End of Section.

6 Equations of Motion

Frame 1

The title of this section, "Equations of Motion," is just another name for a very common and important type of problem: **first-order ordinary differential equations (ODEs)** with a given starting condition. This is known as an **Initial Value Problem (IVP)**.

The general form of an IVP is:

$$y'(x) = f(x, y), \quad \text{with a known starting point} \quad y(x_0) = y_0$$

We are given the derivative of a function, $y'(x)$, which can depend on both the position x and the function's value y itself. We are also given a single point (x_0, y_0) where the solution must pass through. Our goal is to find the function $y(x)$ for other values of x.

The methods we discuss for this scalar (1D) problem can be extended to systems of first-order equations, allowing us to solve problems with multi-dimensional vectors as well.

Go to Frame 2.

Frame 2

While analytical solutions are sometimes possible, we run into trouble as soon as the right-hand side, $f(x, y)$, depends on the unknown function y in any non-trivial way.

Numerically, however, we can solve these problems with a variety of approximation methods. The core idea of all these methods is to start at (x_0, y_0) and take small steps to estimate the value of y at subsequent points.

The simplest of all methods is **Euler's method**. Let's see how it works.

Go to Frame 3.

Frame 3

Euler's Method

To solve the IVP, we first set up a uniform grid of points starting at x_0, with a step size h:

$$x_i = x_0 + i \cdot h, \quad \text{for } i = 0, 1, \ldots, N$$

We will denote our numerical approximation to the true solution $y(x_i)$ as Y_i. We already know $Y_0 = y_0$. How do we find Y_1?

Euler's method makes two key approximations:

1. On the right side of $y' = f(x, y)$, we evaluate the function f using our current known point: $f(x_i, Y_i)$.
2. On the left side, we approximate the derivative $y'(x)$ with a simple **finite difference**:

$$y'(x_i) \approx \frac{y(x_{i+1}) - y(x_i)}{h} \approx \frac{Y_{i+1} - Y_i}{h}$$

By setting these two approximations equal, what is the resulting formula for finding the next point, Y_{i+1}?

Go to Frame 4 for the answer.

Frame 4

Setting the two sides equal gives:

$$\frac{Y_{i+1} - Y_i}{h} = f(x_i, Y_i)$$

Solving for Y_{i+1} gives us the **Euler method** iteration formula:

$$Y_{i+1} = Y_i + h \cdot f(x_i, Y_i), \quad \text{with} \quad Y_0 = y_0$$

This method is very simple: to find the next point, you take the current point and add the step size h multiplied by the slope calculated at the current point. It is equivalent to following a tangent line for a full step.

This method is exact for linear functions (straight lines).

Go to Frame 5.

Frame 5

Error in Euler's Method

If a method is exact for a straight line but not a parabola, we can use a Taylor expansion to see that the error it makes in a single step (the **local error**) is of order h^2. We write this as $O(h^2)$. The smaller the step size h, the much smaller the error per step.

However, to get from a to b, we must take $N = (b - a)/h$ steps. The errors from each step accumulate. The total **global error** is the sum of these local errors. For Euler's method, the global error is one order worse than the local error.

Global Error $\approx N \times$ (Local Error) $\propto \frac{1}{h} \times h^2 = O(h)$.

This means if you halve the step size, you halve the total error. This is not very efficient.

Is a smaller step size always better? What practical limitation prevents us from using infinitely small step sizes? _____

Go to Frame 6 for the answer.

Frame 6

A smaller step size is not always better due to the **finite precision of the computer**. As shown in the graph from the slides, the total error in a numerical method is a sum of two things:

- **Discretization Error:** The error from the method's approximation (e.g., approximating a curve with straight lines). This decreases as h gets smaller.
- **Round-off Error:** The error from the computer's inability to store numbers with infinite precision. This error accumulates with each calculation and becomes *worse* as you take more steps (i.e., as h gets smaller).

There is an **optimal step size,** h, that balances these two competing sources of error. Making the step size smaller than this optimum will actually make the final result less accurate.

Go to Frame 7.

Frame 7

Heun's Method (Modified Euler)

We can improve on Euler's method by getting a better estimate of the slope to use for the step. **Heun's method**, also called a predictor-corrector method, does this in two stages:

1. **Predictor:** First, take a temporary Euler step to predict the value at the next point: $\tilde{Y}_{i+1} = Y_i + h \cdot f(x_i, Y_i)$.

2. **Corrector:** Then, calculate the slope at this predicted point, $f(x_{i+1}, \tilde{Y}_{i+1})$. The final step is taken using the *average* of the slope at the beginning and the predicted slope at the end:

$$Y_{i+1} = Y_i + \frac{h}{2}\left[f(x_i, Y_i) + f(x_{i+1}, \tilde{Y}_{i+1})\right]$$

This method is exact for parabolas and has a much better global error of $O(h^2)$. The trade-off is that it requires two function evaluations per step.

Go to Frame 8.

Frame 8

Runge-Kutta Methods

The idea of using intermediate steps to get a better slope estimate can be generalized. This leads to the family of **Runge-Kutta (RK) methods**.

The **Midpoint Method** is another second-order method ($O(h^2)$ global error) that uses two evaluations per step. It takes a half-step using the Euler slope, evaluates the slope at that midpoint, and then uses that midpoint slope to take the full step from the original point.

The most widely used of all ODE solvers is the **classical 4th-order Runge-Kutta (RK4) method**. It uses a weighted average of four slope calculations per step to achieve a very accurate global error of $O(h^4)$. This means halving the step size reduces the error by a factor of 16! Its formulas are complex, but the result is a very powerful and general-purpose algorithm.

Go to Frame 9.

Frame 9

Adaptive Step Size Methods

So far, we have assumed a fixed step size h. This is not always optimal. If the solution curve is changing slowly, we can take large steps, but if it's changing rapidly, we need to take small steps to maintain accuracy.

Adaptive methods automatically adjust the step size. A common technique, used in the **Runge-Kutta-Fehlberg (RKF45)** method, is to calculate the next point using two different methods at once (e.g., a 4th-order and a 5th-order method).

The difference between the two results, $|Y_{i+1}^{(5)} - Y_{i+1}^{(4)}|$, gives an estimate of the local error. How can this error estimate be used to control the next step? _____

Go to Frame 10 for the answer.

Frame 10

The error estimate is used to control the step size as follows:

- If the estimated error is larger than a desired tolerance, the current step is rejected, the step size h is reduced, and the step is re-calculated.
- If the estimated error is much smaller than the tolerance, the step is accepted, and the step size h is increased for the next step to improve efficiency.

This allows the algorithm to automatically take small steps only when needed, making it both accurate and efficient.

Go to Frame 11.

Frame 11

Method Stability

When solving ODEs, we must also consider the **stability** of the method. For certain equations, particularly those describing decaying processes (like $y' = -ky$ where $k > 0$), a numerical method can "blow up" and give wildly oscillating or infinite results if the step size h is too large.

For Euler's method applied to $y' = -ky$, the stability condition is $|1 - kh| < 1$, which requires that the step size $h < 2/k$. If h is chosen larger than this, the numerical solution will be unstable and diverge from the true solution, even though the true solution is a simple decay to zero.

Different methods have different regions of stability, which is a crucial factor in choosing the right algorithm for a problem.

This concludes our overview of methods for solving initial value problems.

End of Section.

7 Newton's Law

Frame 1

Today we will extend our knowledge of solving differential equations from first-order to **second-order differential equations**.

Newton's Second Law of Motion is a prime physical example of a second-order ODE, making it a great case study for building intuition. This extension will allow us to solve differential equations of any order and in any number of dimensions.

Go to Frame 2.

Frame 2

In its most general vector form, Newton's Second Law is written as:

$$\vec{F} = m\frac{d^2\vec{r}}{dt^2}$$

where $\vec{F}$ is the force vector, m is mass, and $\vec{r}$ is the position vector.

This is a second-order ODE because it involves the second derivative of position. To find a specific trajectory, we need more than just the equation. What else is required to uniquely define a solution? (Hint: Think about what you need to know to predict the path of a thrown ball). ___

Go to Frame 3 for the answer.

Frame 3

To find a unique solution, we need the appropriate **initial conditions**. For a second-order ODE, we need to know the initial state of both the function and its first derivative. For Newton's law, this means we need the initial position and the initial velocity:

$$\vec{r}(t=0) = \vec{r}_0 \quad \text{and} \quad \dot{\vec{r}}(t=0) = \vec{v}_0$$

In three spatial dimensions, $\vec{r} = (x, y, z)$, this means we have a system of three second-order ODEs and require six initial conditions (initial position and velocity for each dimension).

The force $\vec{F}$ can be a function of time t, position $\vec{r}$, and velocity $\dot{\vec{r}}$. This makes the general problem very complex.

Go to Frame 4.

Frame 4

The crucial step for numerically solving a higher-order ODE is to transform it into a system of **first-order ODEs**.

How can we do this for Newton's law? We introduce an intermediate variable. The most physical choice is the **velocity** $\vec{v}$ or the **momentum** $\vec{p}$. Let's choose momentum, $\vec{p} = m\dot{\vec{r}}$.

Using this definition, how can we rewrite Newton's Second Law, $\ddot{\vec{r}} = \vec{F}/m$, as a system of two *first-order* differential equations for the variables $\vec{r}$ and $\vec{p}$? 1. $\dot{\vec{r}} =$? 2. $\dot{\vec{p}} =$?

Go to Frame 5 for the answer.

Frame 5

By introducing momentum, we transform the single second-order equation into a system of two coupled first-order equations:

$$\dot{\vec{r}}(t) = \vec{p}(t)/m$$
$$\dot{\vec{p}}(t) = \vec{F}(t, \vec{r}, \vec{p})$$

The first equation is simply the definition of momentum. The second equation comes from the fact that $\dot{\vec{p}} = m\ddot{\vec{r}} = \vec{F}$.

The initial conditions also transform: $\vec{r}(t=0) = \vec{r}_0$ and $\vec{p}(t=0) = \vec{p}_0 = m\vec{v}_0$.

We now have a system of six coupled first-order ODEs (three for the components of $\vec{r}$ and three for the components of $\vec{p}$) with six initial conditions.

Go to Frame 6.

Frame 6

We can make this look even more familiar by combining our variables into a single six-dimensional state vector, $\vec{y}$:

$$\vec{y} = (\vec{r}, \vec{p}) = (x, y, z, p_x, p_y, p_z)$$

Our system of two first-order vector equations can now be written as a single first-order IVP for the state vector $\vec{y}$:

$$\dot{\vec{y}}(t) = \vec{f}(t, \vec{y}), \quad \text{with} \quad \vec{y}(t=0) = \vec{y}_0$$

This looks exactly like the Initial Value Problem from the previous section! The only difference is that our variables are now vectors instead of scalars.

This means we can directly apply all the methods we've already learned (Euler, Midpoint, Runge-Kutta, etc.) to solve this problem. For example, the Midpoint method in N-dimensions is:

$$\vec{Y}_{i+1/2} = \vec{Y}_i + \frac{h}{2}\vec{f}(x_i, \vec{Y}_i)$$

$$\vec{Y}_{i+1} = \vec{Y}_i + h \cdot \vec{f}(x_i + h/2, \vec{Y}_{i+1/2})$$

The only difference from the 1D case is the proper use of vector arithmetic. Many programming libraries, like NumPy in Python, handle this automatically.

Go to Frame 7.

Symplectic Methods

Frame 7

In physics, we often deal with systems that have **conserved quantities**, such as energy, momentum, or angular momentum.

None of the standard numerical methods we've discussed (Euler, RK4, etc.) were specifically designed to enforce these conservation laws. Over many steps, their accumulated errors will typically cause the calculated energy of the system to drift, even if the step size is small.

What do we call methods that are specially designed to conserve the energy of a system? ____

Go to Frame 8 for the answer.

Frame 8

Methods that are designed to conserve energy (more precisely, to conserve the phase space volume in Hamiltonian systems) are called **symplectic integrators**.

The story is mathematically more complex, but for our purposes, this is a good working interpretation. While the derivations are often shown in 1D for clarity, they are easily extended to N-dimensions.

Go to Frame 9.

Frame 9

One of the simplest and most effective symplectic methods is the **Verlet (or Störmer-Verlet) method**. It is also known as the **leapfrog method**.

It has a global error of only $O(h^2)$, but it requires only *one* force evaluation per step, making it very efficient. Crucially, while it does not conserve energy perfectly, the energy error oscillates around a constant value instead of drifting away over time. This makes it excellent for long-term simulations of orbital mechanics.

The "leapfrog" name comes from how it updates position and velocity. In one common formulation, it updates velocity at half-steps and position at full-steps, with the two quantities "leaping" over each other.

Go to Frame 10.

Frame 10

The leapfrog method for the second-order equation $y'' = f(y)$ can be derived as follows. First, introduce velocity $v = y'$ as an intermediate variable.

$$y' = v, \quad v' = f(y)$$

The update step is broken into two halves:

1. First, update the velocity to a half-step point:

$$v_{n+1/2} = v_n + \frac{h}{2} f(y_n)$$

2. Then, use this half-step velocity to update the position for a full step:

$$y_{n+1} = y_n + h \cdot v_{n+1/2}$$

3. Finally, use the new position to update the velocity for the second half of the step:

$$v_{n+1} = v_{n+1/2} + \frac{h}{2} f(y_{n+1})$$

Note that the evaluation of the force $f(y_n)$ from the first part can be reused.

Go to Frame 11.

Frame 11

By eliminating the intermediate velocity v, we can write the Verlet method in its most common form, the **Central Difference Method**.

Starting with the update rules and doing some algebra, we arrive at a remarkably simple formula that directly relates the positions at three consecutive time steps:

$$y_{n+1} - 2y_n + y_{n-1} = h^2 f(y_n)$$

This can be derived by approximating the second derivative y'' with a central difference formula. This method is symplectic and is at the heart of many molecular dynamics simulations. A slight inconvenience is that it's not "self-starting"; to find y_1, you need to know both y_0 and a special value for y_{-1} (or get y_1 from a Taylor step).

This concludes our overview of methods for solving Newton's laws.

End of Section.

8 Eigenvalue Boundary Value Problems

Frame 1

In the last section, we saw how to solve second-order differential equations when we are given *initial conditions*—that is, the value of the function and its derivative are both known at the same starting point. This is called an Initial Value Problem (IVP).

Today, we will extend this to a different class of problems where the conditions are specified at different points, typically at the boundaries of an interval. This is called a **Boundary Value Problem (BVP)**.

Conceptually, this section has two parts:

- Learning how to solve general BVPs for differential equations.
- Applying this knowledge to a special type of BVP: finding the **eigenfunctions** and **eigenvalues** of a differential operator.

Go to Frame 2.

Frame 2

Today we are solving problems of the form:

$$y''(x) = f(x, y, y')$$

where we are seeking the function $y(x)$ on an interval $[a, b]$. Instead of initial conditions, we are given boundary conditions. Common types include:

- **Dirichlet conditions:** The value of the function is specified at both ends.

$$y(a) = \alpha, \quad y(b) = \beta$$

- **von Neumann conditions:** The derivative of the function is specified at both ends.

$$y'(a) = \alpha, \quad y'(b) = \beta$$

- **Mixed conditions:** A combination of the function and its derivative is specified at the ends.

How can we solve a problem where we don't have all the necessary information at a single starting point to begin our step-by-step integration?

Go to Frame 3.

The Shooting Method

Frame 3

A powerful and intuitive tool for solving BVPs is the **shooting method**. The core idea is to convert the BVP into an IVP that we already know how to solve. We do this by "guessing" the missing initial conditions.

Let's consider a linear BVP with Dirichlet conditions:

$$y'' + P(x)y' + Q(x)y = R(x), \quad \text{with} \quad y(a) = \alpha, y(b) = \beta$$

We have the starting position $y(a) = \alpha$, but we don't know the starting slope, $y'(a)$.

What is the strategy of the shooting method? [a] Guess the final value $y(b)$. [b] Guess the missing initial slope $y'(a)$. [c] Guess the entire solution $y(x)$.

Choose an answer and go to Frame 4.

Frame 4

Your answer was [a — b — c].

The correct answer is [b]. We guess the missing initial slope, let's call it ϑ. Now we have a complete set of initial conditions: $y(a) = \alpha$ and $y'(a) = \vartheta$. This is an IVP!

We can now solve this IVP using any standard method (like Runge-Kutta) and integrate from $x = a$ to $x = b$. At the end, we check the resulting value, $y(b)$. It will almost certainly not be equal to the required boundary value β, because our initial guess for the slope, ϑ, was wrong.

The problem has now been transformed. We need to find the correct initial "shooting angle" ϑ that makes our solution hit the target value β at $x = b$.

Go to Frame 5.

Frame 5

For a **linear** ODE, we can be very clever. Because of the principle of superposition, we can solve for the general solution $Y(x)$ by combining the solutions of two simpler IVPs:

1. A particular solution, $u(x)$, that satisfies the inhomogeneous equation with the correct starting position but zero slope:

$$u'' + Pu' + Qu = R, \quad \text{with} \quad u(a) = \alpha, u'(a) = 0$$

2. A homogeneous solution, $v(x)$, that satisfies the homogeneous equation with zero starting position but unit slope:

$$v'' + Pv' + Qv = 0, \quad \text{with} \quad v(a) = 0, v'(a) = 1$$

The general solution is then a linear combination: $Y(x) = u(x) + \vartheta v(x)$. This automatically satisfies $Y(a) = \alpha$ and $Y'(a) = \vartheta$.

We can now solve for the correct ϑ_0 by forcing this solution to meet the final boundary condition, $Y(b) = \beta$. What is the resulting expression for ϑ_0? _____

Go to Frame 6 for the answer.

Frame 6

By setting $Y(b) = \beta$, we get:

$$u(b) + \vartheta_0 v(b) = \beta$$

Solving for the correct initial slope gives:

$$\vartheta_0 = \frac{\beta - u(b)}{v(b)}$$

So, for a linear BVP, we can find the exact solution with just two integrations (one for $u(x)$ and one for $v(x)$) and no iteration.

What if the ODE is **non-linear**? For example: $y'' = f(x, y, y')$. Can we still use this trick? [Yes — No]

Go to Frame 7.

Frame 7

The answer is [No]. For non-linear ODEs, the principle of superposition does not hold. We cannot construct the solution from a linear combination of simpler solutions.

In the non-linear case, we are back to a root-finding problem. We define a "discrepancy function" $F(\vartheta)$ which measures how much our solution, integrated with an initial slope ϑ, misses the target at $x = b$:

$$F(\vartheta) = y(b)|_{\vartheta} - \beta$$

We want to find the value of ϑ that makes $F(\vartheta) = 0$. This is a root-finding problem! We can use any of our known methods, like bisection or secant, to iteratively adjust our guess for ϑ until we hit the target.

Go to Frame 8.

Frame 8

Eigenvalue Problems

A special, but very important, type of BVP is an eigenvalue problem for a differential equation. Here, the equation contains a free parameter, λ, and we seek the specific values of λ (the eigenvalues) for which a non-trivial solution exists under homogeneous boundary conditions (e.g., $y(a) = y(b) = 0$). A classic example is the time-independent Schrödinger equation.

$$y''(x) = f(x, y, y', \lambda), \quad \text{with} \quad y(a) = 0, y(b) = 0$$

How can we solve this using the shooting method? (Hint: What do we guess, and what do we try to hit?) _____

Go to Frame 9 for the answer.

Frame 9

We can solve this using the shooting method, but the roles are now different.

- We are trying to find the special value of λ. So, λ is now what we "guess."
- The initial conditions are $y(a) = 0$ and, since the overall scale of an eigenfunction is arbitrary, we can choose a fixed initial slope, for instance $y'(a) = 1$.
- We then integrate the ODE with our guess for λ to $x = b$.
- The "target" we are trying to hit is the other boundary condition: $y(b) = 0$.

Again, this is a root-finding problem. We are looking for the roots λ of the function $F(\lambda) = y(b)|_{\lambda} = 0$. In general, there will be multiple solutions, corresponding to the different eigenvalues $\lambda_1, \lambda_2, \ldots$ of the system.

Go to Frame 10.

The Finite Difference Method

Frame 10

An entirely different approach to solving BVPs is the **finite difference method**. Instead of converting the problem to an IVP, we discretize the entire interval $[a, b]$ on a grid of points Y_j. The essential step is to replace all derivatives in the ODE with their finite difference approximations.

For a second-order derivative, what is the standard central difference approximation?

$$y_j'' \approx ?$$

Go to Frame 11 for the answer.

Frame 11

The central difference approximation for the second derivative is:

$$y_j'' \approx \frac{Y_{j+1} - 2Y_j + Y_{j-1}}{h^2}$$

By replacing the derivatives in our original ODE, $y'' = f(x, y, y')$, with these algebraic approximations, we convert the differential equation into a large system of coupled algebraic equations for the unknown values Y_j at each grid point.

For a linear ODE, this results in a large system of linear equations, which can be written in matrix form:

$$\mathbf{A} \cdot \vec{Y} = \vec{R}$$

where $\vec{Y}$ is the vector of unknown function values. The matrix $\mathbf{A}$ is typically **tridiagonal**, which allows for very fast and efficient solution using specialized linear algebra routines (like the Thomas algorithm).

Go to Frame 12.

Frame 12

How does the finite difference method handle an eigenvalue problem like the Sturm-Liouville equation?

$$y'' = -Q(x)y - \lambda V(x)y, \quad \text{with} \quad y(a) = y(b) = 0$$

When we discretize this equation, it transforms into a matrix equation of the form:

$$\mathbf{Z} \cdot \vec{Y} = \lambda h^2 \mathbf{V} \cdot \vec{Y}$$

where $\mathbf{Z}$ and $\mathbf{V}$ are tridiagonal matrices. This is a **generalized matrix eigenvalue problem**.

We have transformed the problem of finding the eigenvalues of a *differential operator* into the problem of finding the eigenvalues of a *matrix*. We can now use the powerful matrix methods from the previous section (like QR iteration) to find the eigenvalues λ and eigenvectors $\vec{Y}$ (which are the discretized eigenfunctions).

This concludes our overview of methods for solving boundary value problems.

End of Section.

9 Initial Value Problems for Partial Differential Equations

Frame 1

Today we will extend our knowledge from solving Ordinary Differential Equations (ODEs) to solving **Partial Differential Equations (PDEs)**. PDEs are differential equations that involve derivatives with respect to more than one independent variable.

We will focus on two foundational examples from physics:

- The Diffusion Equation
- The Wave Equation

We will essentially recycle the approaches we've already learned for ODEs. A key theme today will be the **stability** of our numerical methods, which is heavily influenced by our choice of step sizes in both space and time.

Go to Frame 2.

The Diffusion Equation

Frame 2

Our first example is the **diffusion equation**, which describes processes like heat conduction. We want to find the temperature field, $T(x,t)$, in a one-dimensional layer of thickness a. The governing equation is:

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2} + \frac{q}{\rho c}$$

where D is the diffusion coefficient and $q(x,t)$ is a heat source term.

To solve this, we need not only the equation itself but also the appropriate initial and boundary conditions. What are the three conditions needed to define a unique solution for $T(x,t)$ on a domain $x \in [0,a]$ and $t \geq 0$? 1. _____ 2. _____ 3. _____

Go to Frame 3 for the answer.

Frame 3

To specify the problem, we need: 1. An **initial condition**: The temperature profile at $t = 0$.

$$T(x, t = 0) = f(x)$$

2. A **boundary condition** at $x = 0$:

$$T(x = 0, t) = g_0(t)$$

3. A **boundary condition** at $x = a$:

$$T(x = a, t) = g_1(t)$$

The diffusion equation has a property called the **maximum principle**: for pure diffusion (no sources), the maximum and minimum values of the solution $T(x,t)$ must occur either at the initial time ($t = 0$) or on the spatial boundaries ($x = 0$ or $x = a$). This is physically intuitive: the hottest/coldest spot in a cooling/heating object can't spontaneously appear in the middle.

Go to Frame 4.

Frame 4

Discretization

The first step in solving a PDE numerically is to **discretize** the problem domain. We create a grid in both space and time.

- We divide the spatial domain $[0, a]$ into M steps of size $h = a/M$. Grid points are $x_m = m \cdot h$.
- We take time steps of size κ. Grid points are $t_n = n \cdot \kappa$.

Our continuous function $u(x, t)$ is now represented by its values on this grid: $u_m^n = u(x_m, t_n)$.

The next step is to replace the partial derivatives with finite difference approximations. What is a simple, first-order approximation for the time derivative $\partial u/\partial t$? (Hint: Think of the simplest possible way to estimate a rate of change.) _____

Go to Frame 5 for the answer.

Frame 5

The simplest approximation for the time derivative is a **forward difference**:

$$\frac{\partial u}{\partial t} \approx \frac{u_m^{n+1} - u_m^n}{\kappa}$$

This has a local error of $O(\kappa)$.

For the spatial derivative, $\partial^2 u/\partial x^2$, we should use a method that is symmetric and has better stability properties. The standard choice is the **central difference** approximation we've seen before:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{m+1}^n - 2u_m^n + u_{m-1}^n}{h^2}$$

This has a local error of $O(h^2)$.

Go to Frame 6.

Frame 6

By substituting these finite differences into our original PDE, $\partial_t u = D\partial_x^2 u + Q$, we convert the PDE into an algebraic **difference equation**.

$$\frac{u_m^{n+1} - u_m^n}{\kappa} = D\frac{u_{m+1}^n - 2u_m^n + u_{m-1}^n}{h^2} + Q$$

This is an **explicit method** known as **FTCS** (Forward Time, Centered Space). We can solve for the unknown future value u_m^{n+1} directly in terms of known present values:

$$u_m^{n+1} = u_m^n + r\left(u_{m+1}^n - 2u_m^n + u_{m-1}^n\right) + \kappa Q, \quad \text{where } r = \frac{D\kappa}{h^2}$$

This method is very easy to program. However, it has a major drawback. What is it? [a] It is computationally very expensive. [b] It is only conditionally stable. [c] It only works for linear equations.

Go to Frame 7.

Frame 7

Your answer was [a — b — c].

The correct answer is [b]. The FTCS method is only stable if the step sizes are chosen carefully. A von Neumann stability analysis shows that the method becomes unstable and "blows up" if the parameter r is too large. The stability condition is:

$$r = \frac{D\kappa}{h^2} \leq \frac{1}{2}$$

This is a very restrictive condition. It means that if you want to double your spatial resolution (halve h), you must reduce your time step κ by a factor of four.

Go to Frame 8.

Frame 8

The Crank-Nicolson Method

A much better method is the **Crank-Nicolson method**. It achieves better stability by being an **implicit method**. It approximates the spatial derivative not at the current time step n, but as an *average* between the current step n and the future step $n+1$:

$$\frac{u_m^{n+1} - u_m^n}{\kappa} = \frac{D}{2}\left[\left(\frac{\delta^2 u}{\delta x^2}\right)^{n+1} + \left(\frac{\delta^2 u}{\delta x^2}\right)^{n}\right] + Q$$

(where $\frac{\delta^2 u}{\delta x^2}$ is the central difference operator).

What is the main drawback of an implicit method like this? (Hint: Look at the terms. Can you solve for u_m^{n+1} directly?) _____

Go to Frame 9 for the answer.

Frame 9

The drawback of an implicit method is that the unknown future values (u^{n+1}) appear on both sides of the equation. We cannot solve for them one by one.

Instead, the difference equation becomes a **system of linear equations** for all the unknown points u_m^{n+1} at once. This can be written in a matrix form:

$$\left(\mathbf{I} - \frac{r}{2}\mathbf{A}\right)\vec{u}^{n+1} = \left(\mathbf{I} + \frac{r}{2}\mathbf{A}\right)\vec{u}^{n} + \vec{b}$$

where $\mathbf{A}$ is a tridiagonal matrix representing the spatial derivative operator.

The great advantage of the Crank-Nicolson method is that it is **unconditionally stable** for any choice of r. It also has a better accuracy of $O(\kappa^2 + h^2)$.

Go to Frame 10.

The Wave Equation

Frame 10

Our second example is the **wave equation**, which describes phenomena like a vibrating string. We seek the displacement of the string, $u(x,t)$. The equation is:

$$\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2}$$

This is a second-order PDE in time. To solve it, we need two initial conditions (e.g., initial position and initial velocity of the string) and two boundary conditions (e.g., the string is fixed at both ends).

We can discretize this equation in the same way as the diffusion equation, using central differences for both the time and space derivatives.

$$\frac{u_m^{n+1} - 2u_m^n + u_m^{n-1}}{\kappa^2} = c^2 \frac{u_{m+1}^n - 2u_m^n + u_{m-1}^n}{h^2}$$

This gives an explicit scheme for finding the future displacement u_m^{n+1} based on the displacement at the two previous time steps.

Go to Frame 11.

Frame 11

This explicit finite difference method for the wave equation is also only conditionally stable. The stability condition is known as the **Courant-Friedrichs-Lewy (CFL) condition**:

$$\frac{c\kappa}{h} \leq 1$$

What is the physical interpretation of this condition? (Hint: c is the wave speed, κ is the time step, and h is the spatial step). _____

Go to Frame 12 for the answer.

Frame 12

The Courant condition means that the numerical domain of dependence must contain the physical domain of dependence. In simpler terms, in one time step κ, a wave cannot travel further than one spatial grid step h. The information in the simulation cannot travel faster than the physical speed of the wave.

It turns out this simple finite difference method for the wave equation is also a symplectic method. This is why, when stable, it performs better than one might expect, conserving energy well over long simulations.

This concludes our quick overview of methods for solving partial differential equations.

End of Section.

10 Fourier Analysis

Frame 1

Today we will tackle the numerical approach to **Fourier Transforms (FT)**. The Fourier Transform is one of the most powerful tools in all of science and engineering. It is used for spectral analysis of all kinds, from the light of distant stars to stock market trends. It is also a crucial tool for solving certain types of differential equations.

The core idea of Fourier analysis is to decompose a function (or signal) into the frequencies that make it up.

Go to Frame 2.

Frame 2

The Fourier Transform Pair

A function of time, $h(t)$, can be transformed into a function of frequency, $H(f)$. This pair is defined by the Fourier Transform and its inverse. We will use the following convention:

$$\text{Forward FT:} \quad H(f) = \int_{-\infty}^{\infty} h(t) e^{-2\pi i f t} dt$$

$$\text{Inverse FT:} \quad h(t) = \int_{-\infty}^{\infty} H(f) e^{+2\pi i f t} df$$

Here, f is the frequency. Note that physicists often prefer to work with angular frequency $\omega = 2\pi f$. Different conventions exist for the placement of the 2π factor, so always be careful to check which one is being used.

What kind of function is $H(f)$ in general, even if $h(t)$ is a purely real-valued function? [a] Real and even [b] Purely imaginary [c] Complex

Go to Frame 3.

Frame 3

Your answer was [a — b — c].

The correct answer is [c] Complex. The term $e^{-2\pi i f t} = \cos(2\pi f t) - i \sin(2\pi f t)$ is complex, so the resulting transform $H(f)$ will generally have both a real and an imaginary part.

However, if the original function $h(t)$ is real, its transform $H(f)$ has a special symmetry:

$$H(-f) = H^*(f)$$

where H^* is the complex conjugate. This means that the real part of $H(f)$ is an even function, and the imaginary part is an odd function. This property implies that all the information is contained in the positive frequencies; the negative frequency part is redundant.

Another key property is **Parseval's Theorem**, which relates the total energy or power in the time domain to the energy in the frequency domain:

$$\int_{-\infty}^{\infty} |h(t)|^2 dt = \int_{-\infty}^{\infty} |H(f)|^2 df$$

Go to Frame 4.

The Discrete Fourier Transform (DFT)

Frame 4

To compute a Fourier Transform numerically, we must work with a discrete set of samples. This leads to the **Discrete Fourier Transform (DFT)**.

We start with a function $h(t)$ that we sample at N points over a total time interval T. The time step is $\Delta = T/N$, and our sample times are $t_n = n\Delta$ for $n = 0, 1, \ldots, N-1$. The sampling frequency is $\nu_s = 1/\Delta$.

The continuous integral for the FT is replaced by a finite sum. What is the fundamental assumption we are implicitly making about our function $h(t)$ when we do this? (Hint: what does a finite sample imply about the signal's behavior outside the window $[0, T]$?) _____

Go to Frame 5 for the answer.

Frame 5

By replacing the infinite integral with a finite sum over the interval $[0, T]$, we are implicitly assuming that our function $h(t)$ is **periodic** with period T. That is, $h(t + T) = h(t)$.

The discrete frequencies are then defined as multiples of the fundamental frequency $1/T$:

$$f_k = \frac{k}{T} = \frac{k}{N\Delta}, \quad \text{for } k = 0, 1, \ldots, N-1$$

The DFT and its inverse are then defined as:

$$\text{DFT:} \quad H_k = \sum_{n=0}^{N-1} h_n e^{-i2\pi kn/N}$$

$$\text{Inverse DFT:} \quad h_n = \frac{1}{N} \sum_{k=0}^{N-1} H_k e^{+i2\pi kn/N}$$

where $h_n = h(t_n)$ and $H_k \approx H(f_k) \cdot T$. (Note the normalization factor of $1/N$ appears in the inverse transform).

Go to Frame 6.

Frame 6

The Nyquist Frequency

Our discrete frequency index k runs from 0 to $N-1$. However, because of the periodicity of the complex exponential, these frequencies "wrap around." The frequency corresponding to the index $k = N/2$ is special. It is called the **Nyquist frequency**, ν_c:

$$\nu_c = \frac{N/2}{T} = \frac{N/2}{N\Delta} = \frac{1}{2\Delta} = \frac{\nu_s}{2}$$

The Nyquist frequency is exactly half the sampling rate.

What does the famous **Nyquist-Shannon Sampling Theorem** state about this frequency?

Go to Frame 7 for the answer.

Frame 7

The Sampling Theorem states that if a continuous signal $h(t)$ contains no frequencies higher than ν_c, then it can be *perfectly reconstructed* from its discrete samples taken at a rate of $\nu_s = 2\nu_c$.

In other words, if we sample fast enough (at least twice the highest frequency in our signal), we lose no information.

But what happens if our signal *does* contain frequencies higher than ν_c? [a] Those frequencies are simply lost. [b] Those frequencies get "folded" or "aliased" into the lower frequency range. [c] The transform gives an error.

Go to Frame 8.

Frame 8

Your answer was [a — b — c].

The correct answer is [b]. If the signal contains frequencies above the Nyquist frequency, those frequencies are not lost but are incorrectly represented as lower frequencies. This phenomenon is called **aliasing**.

For example, a high-frequency signal can alias to a low frequency, appearing as a "beat" pattern or a completely different tone. This is the reason for the strange patterns you sometimes see on video with spoked wheels or helicopter blades.

To prevent aliasing, one must either sample at a much higher rate or apply a **low-pass filter** to the analog signal to remove high frequencies *before* sampling.

Go to Frame 9.

Frame 9

Leakage

Another important numerical artifact arises from our assumption of periodicity. Suppose our true signal is a pure sine wave, but the sampling interval T does not contain an exact integer number of its cycles.

When the DFT assumes the signal is periodic with period T, it sees a sharp discontinuity at the boundary where the end of the signal "wraps around" to connect to the beginning.

What effect does a sharp discontinuity in the time domain have on the frequency spectrum? (Hint: sharp features require what kind of frequencies?) _____

Go to Frame 10 for the answer.

Frame 10

Sharp features, like discontinuities, require a very broad range of high frequencies to be represented.

Therefore, the power from our single-frequency sine wave "leaks" out into many other frequency bins in the DFT. Instead of a single sharp peak, we get a broadened peak with significant side-lobes. This is called **spectral leakage**.

To minimize leakage, one should choose a sampling window that matches the signal's periodicity if possible, or use "windowing functions" that smoothly taper the signal to zero at the boundaries.

Go to Frame 11.

Frame 11

The Fast Fourier Transform (FFT)

Calculating the DFT directly using the summation formula requires approximately N^2 complex multiplications. For large N, this is computationally very expensive.

In 1965, Cooley and Tukey rediscovered a highly efficient algorithm for calculating the DFT, now known as the **Fast Fourier Transform (FFT)**. The FFT is a clever recursive algorithm that breaks down a transform of size N into smaller transforms.

What is the computational complexity of the FFT algorithm? [a] $O(N^2)$ [b] $O(N \log N)$ [c] $O(N)$

Go to Frame 12.

Frame 12

Your answer was [a — b — c].

The correct answer is [b]. The FFT algorithm reduces the number of required operations from $O(N^2)$ to $O(N \log N)$. This is a colossal improvement. For $N = 1,000,000$, the difference is between a trillion (10^{12}) operations and about 20 million ($2 \cdot 10^7$) operations. The FFT is what made digital signal processing practical.

FFT routines are standard in almost all programming languages and libraries (e.g., 'numpy.fft' in Python). It's essential to use helper functions (like 'fftshift') to arrange the frequency components in the correct, physically intuitive order (from $-\nu_c$ to $+\nu_c$).

This concludes our overview of Fourier Analysis.

End of Section.