LU Decomposition

LU Decomposition Method

Introduction

LU decomposition is used in numerical methods to solve linear equations. To solve a problem like Ax = B, LU decomposition breaks the matrix (A) into two simpler matrices: a lower-triangular matrix (L) and an upper-triangular matrix (U).

Another big advantage is efficiency: if we need to solve the same matrix (A) with many different **b** vectors, we only need to do LU factorization once.

Theory

1) The main idea

Given a square matrix (A). It will be rewritten as follows :

A = LU, where:

- **(L)** is a lower triangular matrix
- **(U)** is an upper triangular matrix

2) Using LU to solve AX = B

Once we have (A = LU), we can write:

LUx = b

To make this easy, we solve it in two smaller steps. We introduce a helper vector Y :

Ly=b

Ux=y

- First, solve **Ly=b** using **forward substitution**
- Then, solve **Ux=y** using **backward substitution**

3) Why pivoting is often needed

Sometimes LU decomposition runs into trouble if a diagonal element (called a **pivot**) becomes zero. That can cause division problems and also makes result less reliable.

To fix this, we often swap rows to bring a better pivot into position. With row swapping, the factorization becomes:

$$\mathbf{PA = LU}$$

where (P) is a matrix that represents the row swaps. This is called **partial pivoting**. It makes the method more stable in real calculations.

Algorithm

The diagonal of (L) is set to 1. The idea is:

For each column/step (k = 1) to (n):

1. Compute the (k)-th row of (U)
2. Compute the (k)-th column of (L)

Mathematically:

$$U_{k,j} = A_{k,j} - \sum_{s=1}^{k-1} L_{k,s}U_{s,j} \ , \ j = k,\ldots,n$$

$$L_{i,k} = \left(A_{i,k} - \sum_{s=1}^{k-1} L_{i,s}U_{s,k}\right) \ / \ (U_{k,k}) \ , \ i = k+1,\ldots,n$$

And set $L_{k,k} = 1$

If pivoting is used row swaps are done before continuing whenever needed.

Procedure

Step 1: Check the matrix

- Make sure (A) is a square matrix.

Step 2: Decompose (A)

- Find (L) and (U) such that:
    - (A = LU)
    - (PA = LU)

Step 3: Solve the first triangular matrix

- Solve: Ly = b

Step 4: Solve the second triangular system

- Solve: Ux = b

The vector X is the solution of the problem

Runge-Kutta

Runge–Kutta (RK) Method

Introduction

When we work with ordinary differential equations (ODEs), we often try to understand how one value changes when another value changes. For example, how (y) changes when (x) increases. An ODE has one independent variable and one dependent variable with one or more derivatives of the dependent variable with respect to the independent variable.

Some ODEs can be solved exactly (analytical solution) but many real problems become too complicated for that. In those cases we use numerical methods to get a good approximation. The Runge–Kutta method is one of the most popular method to solve this kind of numerical problems.

Theory

The Runge–Kutta (RK) method is a family of step-by-step methods used to approximate solutions of ODEs. In practice, the most common one is the 4th-order Runge–Kutta method (RK4) because it gives a strong balance between accuracy and efficiency.

Most RK4 problems are written like this (a first-order ODE with an initial value):

- Differential equation: $\frac{dy}{dx} = f(x, y)$
- Initial condition: $y(x0) = y0$

The main idea is simple: instead of using one slope to move from $\mathbf{x_n}$ to $\mathbf{x_{n+1}}$, RK4 samples the slope multiple times (start, midpoints, end) and then takes a weighted average. This makes it more accurate than basic methods.

A key practical choice is the step size (h) (how far you move in (x) each time). Smaller (h) usually improves accuracy, but it also increases the number of calculations.

Algorithm

Given (x_n, y_n) and step size (h):

$k1 = h * f(x_n, y_n)$

$k2 = h * f(x_n + \frac{h}{2}, y_n + \frac{k1}{2})$

$k3 = h * f(x_n + \frac{h}{2}, y_n + \frac{k2}{2})$

$k4 = h * f(x_n + \frac{h}{2}, y_n + \frac{k3}{2})$

Then update:

$y_{n+1} = y_n + (\frac{1}{6}) * (k_1 + 2k_2 + 2k_3 + k_4)$

Also update:

$x_{n+1} = x_n + h$

Procedure

1. **Write the ODE in the right form**
   Make sure your equation is written as (dy/dx = f(x,y)), and note the initial condition (y(x_0)=y_0).
2. **Choose the step size (h)**
   Pick how much (x) will increase each step. Smaller (h) → usually better accuracy.
3. **Start from the initial point**
   Begin at ((x_0, y_0)).
4. **Compute the four slopes (k1, k2, k3, k4)**
   Use the RK4 formulas to calculate the trial slopes based on your function (f(x,y)).
5. **Find the next value**
   Combine the slopes using the RK4 weighted average to get (y_{n+1}), and move to (x_{n+1}=x_n+h).
6. **Repeat until you reach your final (x)**
   Keep stepping forward until you cover the full interval you need (like the examples done over small intervals with a chosen (h)).

Newton Interpoation

Newton's Interpolation Methods

(Forward, Backward, and Divided Difference)

Introduction

In numerical work, we often deal with tables of values instead of a clean formula. For example, we may know (y) at several values of (x) but the value we actually need is somewhere in between. **Interpolation** is the process of estimating that missing value by constructing a polynomial that fits the known data points as closely as possible.

Newton's interpolation methods are popular because they build this polynomial in a structured and efficient way. It uses difference tables instead of solving a full set of simultaneous equations.

Newton's methods are three types:

1. **Forward interpolation** (best near the start of the table, equal spacing)
2. **Backward interpolation** (best near the end of the table, equal spacing)
3. **Divided difference interpolation** (works even when spacing is not equal)

1) Newton's Forward Interpolation

When to use it

Newton's forward interpolation is used when the (x)-values in the table are **equally spaced** and the required value of (x) is closer to the **beginning** of the data set.

Main idea

This method uses **forward differences**. We start from the first value (y_0), then gradually correct it using first difference, second difference and so on. Each new term improves the estimate.

Key formula

First define:

$$h = x_1 - x_0, \quad u = \frac{x - x_0}{h}$$

Then the interpolation expression is:

$$f(x) \approx y_0 + u\Delta y_0 + \frac{u(u-1)}{2!}\Delta^2 y_0 + \frac{u(u-1)(u-2)}{3!}\Delta^3 y_0 + \cdots$$

Algorithm

1. Check that spacing (h) is constant.
2. Build the **forward difference table** ($(\triangle y)$, $(\triangle 2y)$, etc.).
3. Compute (u).
4. Substitute the needed difference values into the formula and calculate.

Procedure

1. Write the table in order starting from (x_0).
2. Create the forward difference table by subtracting consecutive (y)-values.
3. Find (u) using the required (x).
4. Plug values into the forward interpolation formula.
5. Add terms until the next difference terms become very small (or until you run out of table columns).

2) Newton's Backward Interpolation

When to use it

Newton's backward interpolation is also designed for **equally spaced** data, but it is more suitable when the required value of (x) lies near the **end** of the data set.

Main idea

Instead of starting from (y_0), this method starts from the last known value (y_n). It uses **backward differences**. It works in the reverse direction.

Key formula

Define:

$$h = x_1 - x_0, \quad v = \frac{x - x_n}{h}$$

Then :

$$f(x) \approx y_n + v\,\nabla y_n + \frac{v(v+1)}{2!}\nabla^2 y_n + \frac{v(v+1)(v+2)}{3!}\nabla^3 y_n + \cdots$$

Algorithm

1. Confirm the spacing (h) is constant.
2. Build the **backward difference table**.
3. Compute (v).
4. Substitute into the backward formula and evaluate.

Procedure

1. Arrange the table so the last point ((x_n, y_n)) is clearly identified.
2. Construct backward differences by subtracting from the bottom upward.
3. Compute (v) using the required (x).
4. Use the backward interpolation formula starting with (y_n).
5. Continue adding terms until the additional corrections become negligible.

3) Newton's Divided Difference Interpolation

When to use it

Divided difference interpolation is the method to use when the (x)-values are **not equally spaced**. In real-world datasets this is common. So this method is very important.

Main idea

Forward and backward interpolation depend heavily on constant spacing, because the difference table assumes a fixed step size (h). Divided differences remove that limitation by dividing the differences by the actual spacing between points.

Divided differences (how they are formed)

Start with:

$$f[x_0] = y_0,\ f[x_1] = y_1,\ \cdots$$

First divided difference:

$$f[x_0, x_1] = \frac{f[x_1] - f[x_0]}{x_1 - x_0}$$

Second divided difference:

$$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$$

In general:

$$f[x_i, \ldots, x_{i+k}] = \frac{f[x_{i+1}, \ldots, x_{i+k}] - f[x_i, \ldots, x_{i+k-1}]}{x_{i+k} - x_i}$$

Newton's divided difference polynomial

$$P_n(x) = f[x_0] + (x - x_0) f[x_0, x_1] + (x - x_0)(x - x_1) f[x_0, x_1, x_2] + \cdots$$
$$+ (x - x_0)(x - x_1) \cdots (x - x_{n-1}) f[x_0, x_1, \ldots, x_n]$$

Algorithm

1. List the data points clearly.
2. Build the divided difference table column by column.
3. Use the first row coefficients in the Newton polynomial.
4. Substitute the required (x) and compute (P_n(x)).

Procedure

1. Write the first column as the given (y)-values.
2. Compute divided differences using the formulas above.
3. Take the top value of each divided-difference column as the coefficient.
4. Form the Newton polynomial term-by-term.
5. Evaluate the polynomial at the required (x).

Choosing the correct method

1. If (x)-values are equally spaced and the unknown point is near the start → **Forward interpolation**
2. If (x)-values are equally spaced and the unknown point is near the end → **Backward interpolation**
3. If spacing is not equal → **Divided difference interpolation**