present

```
1 html"<button onclick='present()'>present</button>"
```

Runge Kutta Methods

- Lambert Theisen (www.thsn.dev)
- Mathematische Grundlagen III (WS24/25), RWTH Aachen


```
1 # import necessary pkgs
2 using Plots
```

Introduction

In this notebook, we will explore how to implement Runge-Kutta (RK) methods for solving ordinary differential equations (ODEs) using Julia. We will focus on generalizing RK methods using the Butcher tableau, and we'll apply these methods to solve example problems, comparing the numerical solutions with exact solutions.

The Runge-Kutta Method

The Runge-Kutta methods are a family of iterative techniques for approximating solutions to ODEs. They are particularly useful for solving initial value problems of the form:

$$\frac{dy}{dt} = f(y,t), \quad y(t_0) = y_0$$

RK methods estimate the solution at the next time step by combining multiple evaluations of the derivative function (f) at strategically chosen points within the time step. This approach increases the accuracy without requiring higher-order derivatives.

The Butcher Tableau

The Butcher tableau provides a concise representation of the coefficients used in a Runge-Kutta method. It consists of three components: the A matrix, the b vector, and the c vector. The general form is:

$$\begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s \end{array}$$

- A: An $s \times s$ matrix of coefficients for the internal stages.
- b: A vector of weights used to combine the stages for the final update.
- c: A vector representing the nodes where the function f is evaluated.

Implementing the General Runge-Kutta Solver

We will implement a general Runge-Kutta solver function `runge_kutta` that takes as inputs the coefficients A, b, and c, along with the derivative function f, initial condition x_0, time span, and step size h.

runge_kutta (generic function with 1 method)

```julia
# Define the general Runge-Kutta solver function
function runge_kutta(f, x0, tspan, h, A, b, c)
    # Number of steps
    N = Int((tspan[2] - tspan[1]) / h)
    # Initialize arrays to store the solution
    t = collect(range(tspan[1], length=N+1, step=h))
    x = zeros(length(t))
    x[1] = x0

    s = length(b)  # Number of stages

    # Main RK loop
    for n in 1:length(t)-1
        k = zeros(s)
        for i in 1:s
            ti = t[n] + c[i] * h
            xi = x[n] + h * sum([A[i, j] * k[j] for j in 1:i-1])
            k[i] = f(xi, ti)
        end
        x[n+1] = x[n] + h * sum([b[i] * k[i] for i in 1:s])
    end
    return t, x
end
```

Classical RK4 Method

As an example, we will use the classical fourth-order Runge-Kutta (RK4) method, which is widely used due to its balance between computational efficiency and accuracy. The Butcher tableau for the RK4 method is:

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

- **Coefficients**:
  - $A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
  - $b = \left[\frac{1}{6}, \frac{1}{3}, \frac{1}{3}, \frac{1}{6}\right]$
  - $c = \left[0, \frac{1}{2}, \frac{1}{2}, 1\right]$

▸ [0.0, 0.5, 0.5, 1.0]

```julia
begin
    # Define RK4 method coefficients
    A = [0.0  0.0  0.0  0.0;
         0.5  0.0  0.0  0.0;
         0.0  0.5  0.0  0.0;
         0.0  0.0  1.0  0.0]

    b = [1/6, 1/3, 1/3, 1/6]
    c = [0.0, 0.5, 0.5, 1.0]
end
```

Defining the ODE and Exact Solution

We will solve the simple exponential decay ODE:

$$\frac{dx}{dt} = -x\sin(t)^2 x, \quad x(0) = 1$$

The exact solution to this ODE is:

$$x(t) = x(0)\exp(0.5(t - \sin(t)\cos(t)))$$

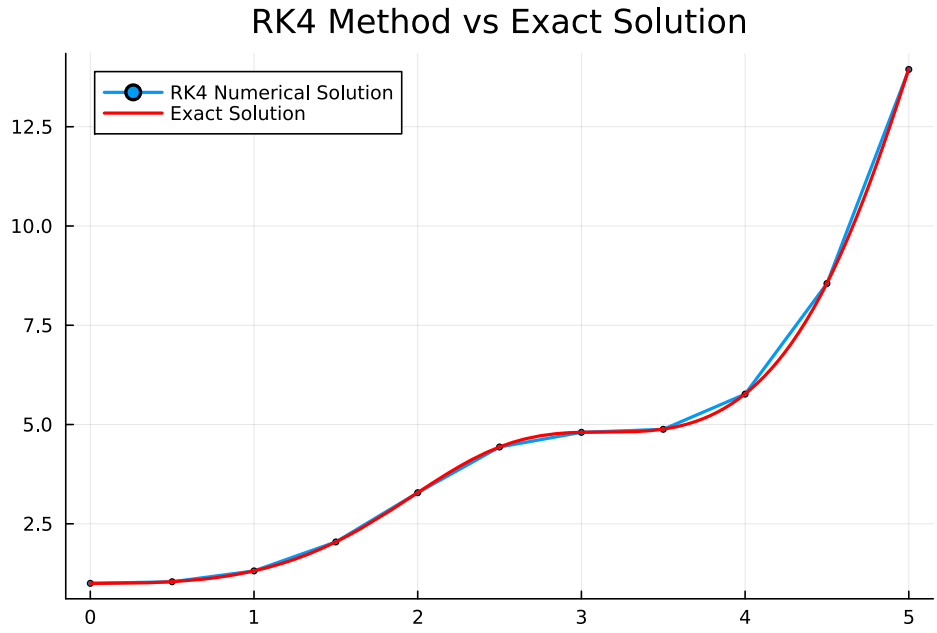
This problem is ideal for demonstrating the accuracy of the RK methods since we have a known analytical solution.

```
0.5
```

```
begin
    # Define the ODE function
    function f(x, t)
        return sin(t)^2 * x  # Example ODE: dx/dt = sin(t)^2 * x
    end

    # Exact solution for comparison
    function exact_solution(t, x0)
        return x0 * exp(0.5 * (t - sin(t) * cos(t)))  # Exact solution
    end

    # Initial condition
    x0 = exact_solution(0.0, 1.0)  # Ensure consistency with the exact solution

    # Time interval and step size
    tspan = (0.0, 5.0)
    h = 0.5  # Base step size
end
```

Running the Solver and Plotting the Results

Now we'll run the `runge_kutta` solver using the RK4 coefficients and plot the numerical solution alongside the exact solution.




```
begin
    # Call the RK solver with RK4 coefficients
    t, x = runge_kutta(f, x0, tspan, h, A, b, c)

    # Compute the exact solution
    x_exact = exact_solution.(t, x0[1])

    # Plot the numerical and exact solutions
    plot(t, x, label="RK4 Numerical Solution", lw=2, markers=:circle, markersize=2)
    plot!(t->exact_solution(t,x0[1]), label="Exact Solution", lw=2, color=:red)
    title!("RK4 Method vs Exact Solution")
end
```

Understanding the Results

The plot shows that the numerical solution obtained using the RK4 method closely matches the exact solution. This demonstrates the high accuracy of the RK4 method for solving ODEs with smooth solutions over relatively large step sizes.

Implementing Explicit Euler and Heun's Methods

In this section, we'll implement both the **Explicit Euler method** and **Heun's method**.

Explicit Euler Method

The Explicit Euler method is the simplest Runge-Kutta method and is a first-order method. Its Butcher tableau is:

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

- **Coefficients**:
  - $A = [0]$
  - $b = [1]$
  - $c = [0]$

Heun's Method

Next, we'll implement **Heun's method**, which is a second-order Runge-Kutta method (also known as the improved Euler method). Its Butcher tableau is:

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

- **Coefficients**:
  - $A = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$
  - $b = \left[\frac{1}{2}, \frac{1}{2}\right]$
  - $c = [0, 1]$

```
▸ (2×2 Matrix{Float64}:, [0.5, 0.5], [0.0, 1.0])
  0.0  0.0
```

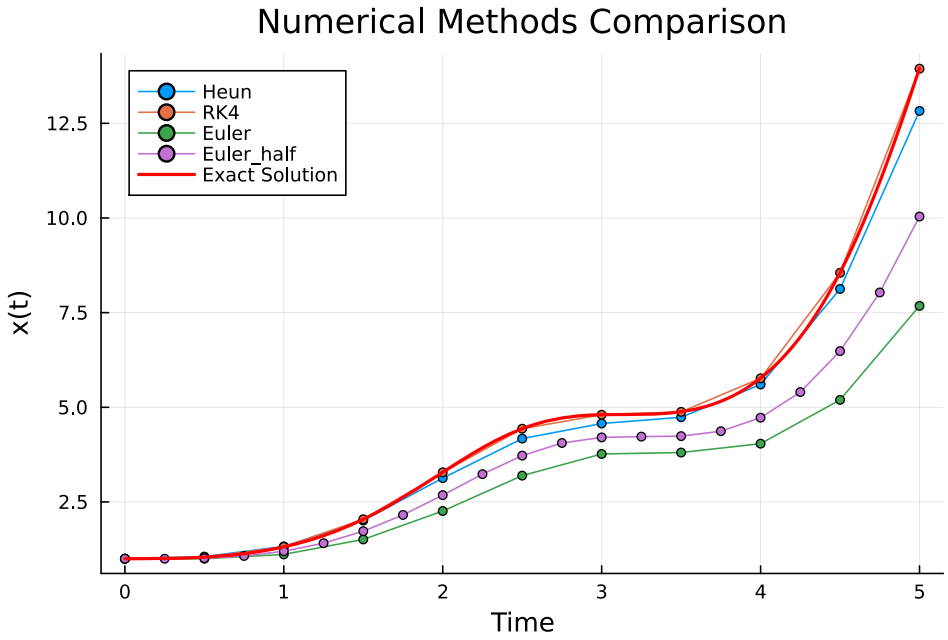
```
begin
    # Define Heun's method coefficients
    function heun_coefficients()
        A = [0.0 0.0;
             1.0 0.0]
        b = [0.5, 0.5]
        c = [0.0, 1.0]
        return A, b, c
    end

    # Get Heun's method coefficients
    A_heun, b_heun, c_heun = heun_coefficients()
end
```

```
▸ ([0.0], [1.0], [0.0])
```

```
begin
    # Define explicite Eulers's method coefficients
    function euler_coefficients()
        A = [0.0]
        b = [1.0]
        c = [0.0]
        return A, b, c
    end

    # Get Heun's method coefficients
    A_euler, b_euler, c_euler = euler_coefficients()
end
```

Comparing RK4 with Heun's and explicite Euler Method

We'll run the solver using Heun's method, explicit Euler and compare the numerical solution with the exact solution and the RK4 solution.




```julia
begin

    # Define methods dictionary
    methods = Dict(
        "RK4" => (A, b, c, h),
        "Heun" => (A_heun, b_heun, c_heun, h),
        "Euler" => (A_euler, b_euler, c_euler, h),
        "Euler_half" => (A_euler, b_euler, c_euler, h/2)
    )

    # Initialize plot
    p = plot(title="Numerical Methods Comparison", xlabel="Time", ylabel="x(t)")

    # For each method, compute the numerical solution and plot it
    for (method_name, (A, b, c, method_h)) in methods
        t_num, x_num = runge_kutta(f, x0, tspan, method_h, A, b, c)
        plot!(t_num, x_num, label=method_name, lw=1, marker=:circle, markersize=3)
    end

    # Plot the exact solution
    plot!(t->exact_solution(t,x0[1]), label="Exact Solution", lw=2, color=:red)

    # Display the plot
    p

end
```