Physics 584 Computational Methods

Introduction to Matlab and Numerical Solutions to Ordinary Differential Equations

Ryan Ogliore

February 15th, 2018

Lecture Outline

Introduction to Matlab

Numerical Solutions to Ordinary Differential Equations

Euler's Method

Taylor Methods

Runge-Kutta Methods

Homework Assignment

Lecture Outline

Introduction to Matlab

Numerical Solutions to Ordinary Differential Equations

Euler's Method

Taylor Methods

Runge-Kutta Methods

Homework Assignment

Matlab

- **Matlab** stands for **mat**rix **lab**oratory.

Matlab

- **Matlab** stands for **mat**rix **lab**oratory.
- Proprietary, commercial programming language developed by MathWorks.

Matlab

- **Matlab** stands for **mat**rix **lab**oratory.
- Proprietary, commercial programming language developed by MathWorks.
- Primarily for numerical calculations, but can also perform symbolic manipulations.

Matlab

- **Matlab** stands for **mat**rix **lab**oratory.
- Proprietary, commercial programming language developed by MathWorks.
- Primarily for numerical calculations, but can also perform symbolic manipulations.
- Capabilities can be greatly expanded via toolboxes and packages ($$$) so that one can build, e.g. graphical-user interfaces.

Matlab

- **Matlab** stands for **mat**rix **lab**oratory.
- Proprietary, commercial programming language developed by MathWorks.
- Primarily for numerical calculations, but can also perform symbolic manipulations.
- Capabilities can be greatly expanded via toolboxes and packages ($$$) so that one can build, e.g. graphical-user interfaces.
- A high-level interpreted language (not compiled) but can run compiled C or Fortran code.

Matlab

- **Matlab** stands for **mat**rix **lab**oratory.
- Proprietary, commercial programming language developed by MathWorks.
- Primarily for numerical calculations, but can also perform symbolic manipulations.
- Capabilities can be greatly expanded via toolboxes and packages ($$$) so that one can build, e.g. graphical-user interfaces.
- A high-level interpreted language (not compiled) but can run compiled C or Fortran code.
- Used broadly in science and engineering, including industry.

Matlab

- **Matlab** stands for **mat**rix **lab**oratory.
- Proprietary, commercial programming language developed by MathWorks.
- Primarily for numerical calculations, but can also perform symbolic manipulations.
- Capabilities can be greatly expanded via toolboxes and packages ($$$) so that one can build, e.g. graphical-user interfaces.
- A high-level interpreted language (not compiled) but can run compiled C or Fortran code.
- Used broadly in science and engineering, including industry.
- **Matlab's power comes from its ease of use, easy debugging, pre-built set of toolboxes, interactive development environment, and visualization**.

Matlab Coder generates readable and portable C and C++ code from Matlab code.

Matlab Coder generates readable and portable C and C++ code from Matlab code.

- It supports most of the Matlab language and a wide range of toolboxes.
- You can integrate the generated code into your projects as source code, static libraries, or dynamic libraries.
- You can also use the generated code within the Matlab environment to accelerate computationally intensive portions of your Matlab code.
- Matlab Coder lets you incorporate legacy C code into your Matlab algorithm and into the generated code.

Please install MATLAB on your laptop if you have one, or have easy access to it if you don't. It works on Linux/Mac/Windows.

Please contact Sai Iyer (sai@physics.wustl.edu) about obtaining and installing Matlab.

Getting Started

Matlab Academy: Matlab Onramp

Matlab offers a nice introduction to the language in the Matlab Academy: https://matlabacademy.mathworks.com/

You'll have to create a login for MathWorks (apologies). But you *do not* need Matlab installed on your computer to use the Matlab Academy.

Please familiarize yourself with Matlab, *before class on Thursday February 22*, by completing the Matlab Onramp in the Matlab Academy. This should take less than two hours to complete.

Which Language Should I Use?

Which Language Should I Use?

- There is probably a perfect language for almost every problem

Which Language Should I Use?

- There is probably a perfect language for almost every problem
- But it is confusing to switch syntax, coding style, etc.

Which Language Should I Use?

- There is probably a perfect language for almost every problem
- But it is confusing to switch syntax, coding style, etc.
- **My advice: find one general language you like, and get good at that!**

Which Language Should I Use?

- There is probably a perfect language for almost every problem
- But it is confusing to switch syntax, coding style, etc.
- **My advice: find one general language you like, and get good at that!**
  - If I were young, it would probably be Python (lots of the benefits of Matlab, plus it's free)

Which Language Should I Use?

- There is probably a perfect language for almost every problem
- But it is confusing to switch syntax, coding style, etc.
- **My advice: find one general language you like, and get good at that!**
  - If I were young, it would probably be Python (lots of the benefits of Matlab, plus it's free)
- Become *efficient* and *comfortable*, and *have fun* (while you have the time)

Which Language Should I Use?

- There is probably a perfect language for almost every problem
- But it is confusing to switch syntax, coding style, etc.
- **My advice: find one general language you like, and get good at that!**
  - If I were young, it would probably be Python (lots of the benefits of Matlab, plus it's free)
- Become *efficient* and *comfortable*, and *have fun* (while you have the time)
- Learn how to develop algorithms, comment your code, and make it readable to others

Which Language Should I Use?

- There is probably a perfect language for almost every problem
- But it is confusing to switch syntax, coding style, etc.
- **My advice: find one general language you like, and get good at that!**
  - If I were young, it would probably be Python (lots of the benefits of Matlab, plus it's free)
- Become *efficient* and *comfortable*, and *have fun* (while you have the time)
- Learn how to develop algorithms, comment your code, and make it readable to others
- *Don't reinvent the wheel*, but also understand how canned algorithms work

Which Language Should I Use?

- There is probably a perfect language for almost every problem
- But it is confusing to switch syntax, coding style, etc.
- **My advice: find one general language you like, and get good at that!**
  - If I were young, it would probably be Python (lots of the benefits of Matlab, plus it's free)
- Become *efficient* and *comfortable*, and *have fun* (while you have the time)
- Learn how to develop algorithms, comment your code, and make it readable to others
- *Don't reinvent the wheel*, but also understand how canned algorithms work
- If a situation arises where you *need* to use another language (e.g. LabView for controlling hardware) then actually *learn* that language (don't just copy code from Stack Overflow)

Lecture Outline

First Order Differential Equations

A first-order ordinary differential equation (ODE) is an initial value problem with the form:

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

First Order Differential Equations

A first-order ordinary differential equation (ODE) is an initial value problem with the form:

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

Differential equations are used to model problems in science and engineering that involve a change of some variable with respect to another (e.g. solve for a time-dependent temperature of a radiating body).

First Order Differential Equations

A first-order ordinary differential equation (ODE) is an initial value problem with the form:

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

Differential equations are used to model problems in science and engineering that involve a change of some variable with respect to another (e.g. solve for a time-dependent temperature of a radiating body).

Most problems are constrained to satisfy an initial condition (e.g. you know the starting temperature of the body).

Solutions to Real-World Differential Equations

Only a small subset of real-world problems can be solved via the analytical techniques students learn in a differential equations math class.

Solutions to Real-World Differential Equations

Only a small subset of real-world problems can be solved via the analytical techniques students learn in a differential equations math class.

One has two choices for the more complicated, real-world problems:

1. Simplify the differential equation to one that can be solved analytically and use that solution to approximate the actual solution
2. Use numerical methods to approximate the solution to the more complicated problem.

Solutions to Real-World Differential Equations

Only a small subset of real-world problems can be solved via the analytical techniques students learn in a differential equations math class.

One has two choices for the more complicated, real-world problems:

1. Simplify the differential equation to one that can be solved analytically and use that solution to approximate the actual solution
2. Use numerical methods to approximate the solution to the more complicated problem.

We will be explore Option #2, using a Matlab implementation of numerical algorithms. This is preferred to Option #1 because it tends to be **more accurate** and can yield **error estimates**.

Solutions to Real-World Differential Equations

Only a small subset of real-world problems can be solved via the analytical techniques students learn in a differential equations math class.

One has two choices for the more complicated, real-world problems:

1. Simplify the differential equation to one that can be solved analytically and use that solution to approximate the actual solution
2. Use numerical methods to approximate the solution to the more complicated problem.

We will be explore Option #2, using a Matlab implementation of numerical algorithms. This is preferred to Option #1 because it tends to be **more accurate** and can yield **error estimates**.

Note: Numerical solutions do not provide a continuous solution to the equation. Rather, the approximate solution is calculated on a grid of values.

Systems of Equations and Higher Order Differential Equations

Numerical methods to solve ODEs can be extended to *systems* of ODEs.

Systems of Equations and Higher Order Differential Equations

Numerical methods to solve ODEs can be extended to *systems* of ODEs.

A higher order ordinary differential equation (ODE) can be converted into a system of first order ODEs by introducing new variables.

The second order ODE:

$$y'' = -y$$

can be written as two first order ODEs:

$$\begin{aligned} z &= y' \\ z' &= -y \end{aligned}$$

Systems of Equations and Higher Order Differential Equations

Numerical methods to solve ODEs can be extended to *systems* of ODEs.

A higher order ordinary differential equation (ODE) can be converted into a system of first order ODEs by introducing new variables.

The second order ODE:

$$y'' = -y$$

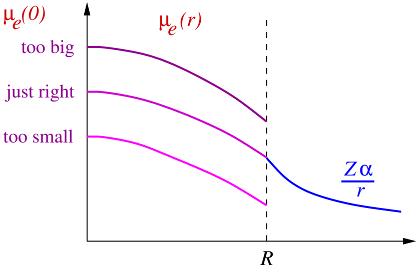
can be written as two first order ODEs:

$$z = y'$$
$$z' = -y$$

$$(\text{solution :} \quad y(x) = c_1 \sin x + c_2 \cos x)$$

Boundary Value Problem vs. Initial Value Problem

Thomas-Fermi: solving Poisson eqn



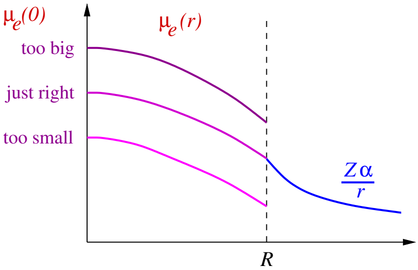
"Shooting" method: vary $\mu_e(0)$ until we get a solution of

$$\frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{d\mu_e}{dr}\right) = -4\pi\alpha Q_\beta\big(\mu_e(r)\big) \qquad \left.\frac{d\mu_e}{dr}\right|_{r=0} = 0$$

that matches to $\mu_e(R) = \dfrac{Z\alpha}{R}$.

Boundary Value Problem vs. Initial Value Problem

Thomas-Fermi: solving Poisson eqn



"Shooting" method: vary $\mu_e(0)$ until we get a solution of

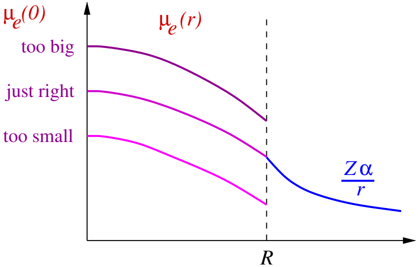
$$\frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{d\mu_e}{dr}\right) = -4\pi\alpha Q_\beta\big(\mu_e(r)\big) \qquad \left.\frac{d\mu_e}{dr}\right|_{r=0} = 0$$

that matches to $\mu_e(R) = \dfrac{Z\alpha}{R}$.

The "shooting method" turned this *boundary-value* problem (constrained at $\mu_e(R)$) into an *initial-value* problem (constrained by $\mu_e(0)$ and $\mu_e'(0)$)

Boundary Value Problem vs. Initial Value Problem

Thomas-Fermi: solving Poisson eqn



"Shooting" method: vary $\mu_e(0)$ until we get a solution of

$$\frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{d\mu_e}{dr}\right) = -4\pi\alpha Q_\beta\left(\mu_e(r)\right) \qquad \left.\frac{d\mu_e}{dr}\right|_{r=0} = 0$$

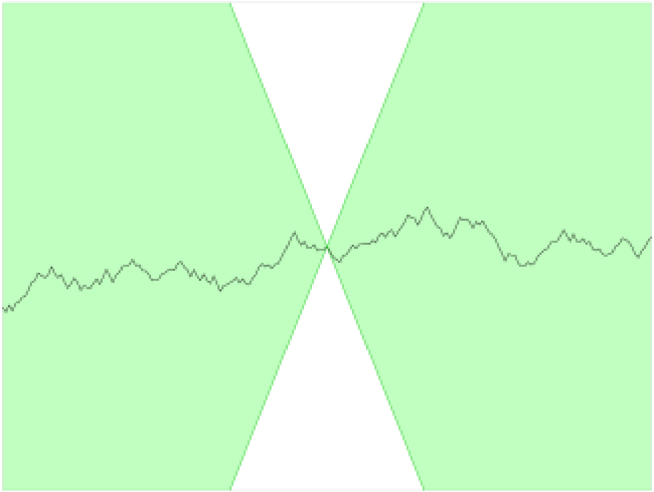
that matches to $\mu_e(R) = \dfrac{Z\alpha}{R}$.

We will be concerned with solutions to ODEs as *initial-value* problems.

Well-Posed Problems and Unique Solutions

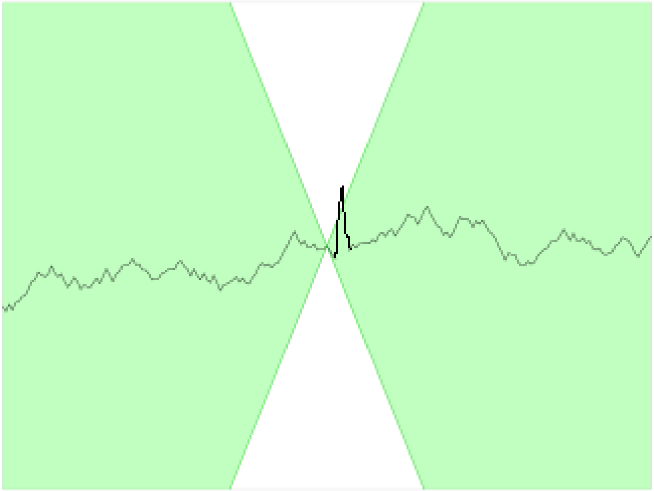
There are ways to determine if an initial value problem (e.g. ODE) has a unique solution within a given domain (**Lipschitz condition**) and if the problem is well-behaved regarding perturbations and round-off error (**well-posed**) but we will not discuss these in detail here.

Lipschitz condition



If we translate the vertex of the double cone (white, defined by the *Lipschitz constant*) along the function, the function always remains in the green area: **satisfies the Lipschitz condition**.

Lipschitz condition



…function crosses into the white area: **violates the Lipschitz condition for that Lipschitz constant**.

Lecture Outline

Introduction to Matlab

Numerical Solutions to Ordinary Differential Equations

Euler’s Method

Taylor Methods

Runge-Kutta Methods

Homework Assignment

Numerical Solutions for ODEs

- Euler's method is simple to understand but rarely used to solve real-world problems.

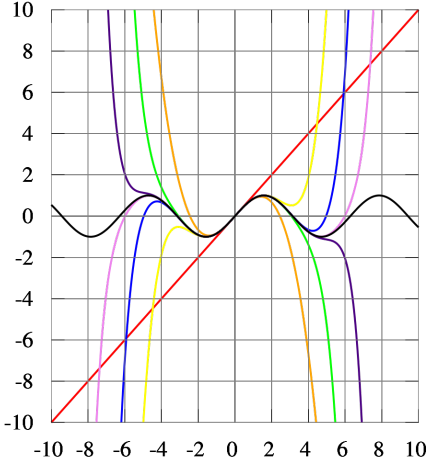
Numerical Solutions for ODEs

- Euler’s method is simple to understand but rarely used to solve real-world problems.
- However, understanding Euler’s method makes it easier to understand the more advanced techniques that we will use to solve ODEs.

Numerical Solutions for ODEs

- Euler's method is simple to understand but rarely used to solve real-world problems.
- However, understanding Euler's method makes it easier to understand the more advanced techniques that we will use to solve ODEs.
- First, we will need Taylor's Theorem…

Reminder: Taylor Series Expansion



$\sin(x)$ (black curve) and its Taylor approximations, polynomials of degree 0 (horizontal line at $y = 0$), 1, 3, 5, 7, 9, 11, and 13

Taylor's Theorem

Suppose f is a function that is $n+1$ times differentiable on the interval $[a,b]$ around x_0. For every x in the interval $[a,b]$ there is a number ξ between x_0 and x with:

$$f(x) = P_n(x) + R_n(x)$$

where:

$$\begin{aligned} P_n(x) &= f(x_0) + f'(x_0)\,(x - x_0) + \frac{f''(x_0)}{2!}\,(x - x_0)^2 + \cdots \\ &\quad + \frac{f^{(n)}(x_0)}{n!}\,(x - x_0)^n \\ &= \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}\,(x - x_0)^k \end{aligned}$$

and

$$R_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}\,(x - x_0)^{n+1}$$

Taylor's Theorem

Suppose f is a function that is $n+1$ times differentiable on the interval $[a, b]$ around x_0. For every x in the interval $[a, b]$ there is a number ξ between x_0 and x with:

$$f(x) = P_n(x) + R_n(x)$$

$$P_n(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!} (x - x_0)^k$$

$$R_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} (x - x_0)^{n+1}$$

$P_n(x)$ is the "nth Taylor polynomial" for f about x_0

$R_n(x)$ is the "remainder term" or "truncation error" of $P_n(x)$.

Taylor’s Theorem: Example

Find a polynomial approximation for $\sin x$ about $x_0 = 0$ accurate to ± 0.005.

Taylor’s Theorem: Example

Find a polynomial approximation for $\sin x$ about $x_0 = 0$ accurate to ± 0.005.

What can we say about the size of:

$$|R_n(x)| = \left| \frac{f^{(n+1)}(\xi)}{(n+1)!} (x - x_0)^{n+1} \right|$$

?

Taylor's Theorem: Example

Find a polynomial approximation for $\sin x$ about $x_0 = 0$ accurate to ± 0.005.

What can we say about the size of:

$$|R_n(x)| = \left| \frac{f^{(n+1)}(\xi)}{(n+1)!} (x - x_0)^{n+1} \right|$$

? Every derivative $f^{(n+1)}(x)$ of $\sin x$ is either $\pm \sin x$ or $\pm \cos x$, so $|f^{(n+1)}(\xi)| \leq 1$.

Taylor's Theorem: Example

Find a polynomial approximation for $\sin x$ about $x_0 = 0$ accurate to ± 0.005.

What can we say about the size of:

$$|R_n(x)| = \left| \frac{f^{(n+1)}(\xi)}{(n+1)!} (x - x_0)^{n+1} \right|$$

? Every derivative $f^{(n+1)}(x)$ of $\sin x$ is either $\pm \sin x$ or $\pm \cos x$, so $|f^{(n+1)}(\xi)| \leq 1$. Let's restrict the range of x to be $[-\pi/2, \pi/2]$, so:

Taylor's Theorem: Example

Find a polynomial approximation for $\sin x$ about $x_0 = 0$ accurate to ± 0.005.

What can we say about the size of:

$$|R_n(x)| = \left| \frac{f^{(n+1)}(\xi)}{(n+1)!} (x - x_0)^{n+1} \right|$$

? Every derivative $f^{(n+1)}(x)$ of $\sin x$ is either $\pm \sin x$ or $\pm \cos x$, so $|f^{(n+1)}(\xi)| \leq 1$. Let's restrict the range of x to be $[-\pi/2, \pi/2]$, so:

$$|R_n(x)| \leq \left| \frac{x^{n+1}}{(n+1)!} \right| \leq \left| \frac{(\pi/2)^{n+1}}{(n+1)!} \right|$$

Taylor's Theorem: Example

Find a polynomial approximation for $\sin x$ about $x_0 = 0$ accurate to ± 0.005.

What can we say about the size of:

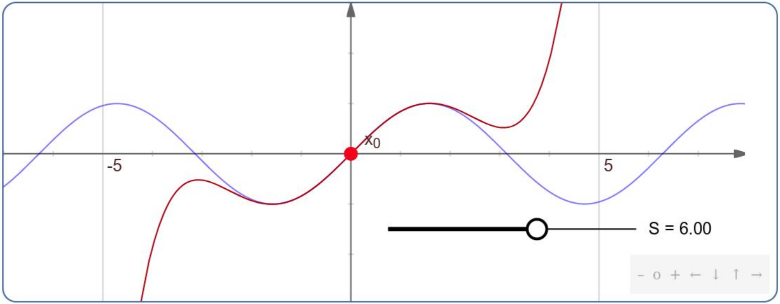
$$|R_n(x)| = \left| \frac{f^{(n+1)}(\xi)}{(n+1)!} (x - x_0)^{n+1} \right|$$

? Every derivative $f^{(n+1)}(x)$ of $\sin x$ is either $\pm \sin x$ or $\pm \cos x$, so $|f^{(n+1)}(\xi)| \leq 1$. Let's restrict the range of x to be $[-\pi/2, \pi/2]$, so:

$$|R_n(x)| \leq \left| \frac{x^{n+1}}{(n+1)!} \right| \leq \left| \frac{(\pi/2)^{n+1}}{(n+1)!} \right|$$

For $n = 6$, this quantity is 0.0047.

Taylor's Theorem: Example



$$0.00 + 1.00\frac{(x-0.00)^1}{1!} + 0.00\frac{(x-0.00)^2}{2!} + -1.00\frac{(x-0.00)^3}{3!} + 0.00\frac{(x-0.00)^4}{4!} + 1.00\frac{(x-0.00)^5}{5!} + 0.00\frac{(x-0.00)^6}{6!}$$

Euler’s Method

The goal of Euler’s Method is to solve the problem:

$$\frac{dy}{dt} = f(t, y) \qquad a \leq t \leq b \qquad y(a) = \alpha \tag{1}$$

Euler's Method

The goal of Euler's Method is to solve the problem:

$$\frac{dy}{dt} = f(t, y) \qquad a \leq t \leq b \qquad y(a) = \alpha \tag{1}$$

First choose the "mesh points" over which the solution will be calculated. Assume we want an equally spaced mesh over the time interval $[a, b]$, such that we construct $t_0, t_1, t_2, \ldots, t_N$:

$$t_i = a + ih \qquad \text{for each } i = 0, 1, 2, \ldots, N$$

The common distance between the points, $h = (b - a)/N$ is called the *step size*.

Euler's Method

Suppose that $y(t)$, the unique solution to Eq. 1, has two continuous derivatives (y' and y'') on $[a, b]$ so that for each $i = 0, 1, 2, \ldots, N-1$ (Taylor's Theorem):

$$y(t_{i+1}) = y(t_i) + (t_{i+1} - t_i)\, y'(t_i) + \frac{(t_{i+1} - t_i)^2}{2} y''(\xi_i) \qquad (2)$$

for some number ξ_i within (t_i, t_{i+1}).

Euler's Method

Suppose that $y(t)$, the unique solution to Eq. 1, has two continuous derivatives (y' and y'') on $[a, b]$ so that for each $i = 0, 1, 2, \ldots, N-1$ (Taylor's Theorem):

$$y(t_{i+1}) = y(t_i) + (t_{i+1} - t_i)\, y'(t_i) + \frac{(t_{i+1} - t_i)^2}{2} y''(\xi_i) \qquad (2)$$

for some number ξ_i within (t_i, t_{i+1}). Set $h = t_{i+1} - t_i$:

$$y(t_{i+1}) = y(t_i) + hy'(t_i) + \frac{h^2}{2} y''(\xi_i) \qquad (3)$$

Euler's Method

Suppose that $y(t)$, the unique solution to Eq. 1, has two continuous derivatives (y' and y'') on $[a, b]$ so that for each $i = 0, 1, 2, \ldots, N-1$ (Taylor's Theorem):

$$y(t_{i+1}) = y(t_i) + (t_{i+1} - t_i)\, y'(t_i) + \frac{(t_{i+1} - t_i)^2}{2} y''(\xi_i) \qquad (2)$$

for some number ξ_i within (t_i, t_{i+1}). Set $h = t_{i+1} - t_i$:

$$y(t_{i+1}) = y(t_i) + hy'(t_i) + \frac{h^2}{2} y''(\xi_i) \qquad (3)$$

Since $y(t)$ satisfies Eq. 1 ($y' = f(t, y)$)

$$y(t_{i+1}) = y(t_i) + hf(t_i, y(t_i)) + \frac{h^2}{2} y''(\xi_i) \qquad (4)$$

Euler's Method

$$y(t_{i+1}) = y(t_i) + hf(t_i, y(t_i)) + \frac{h^2}{2}y''(\xi_i)$$

Euler's method constructs $w_i \approx y(t_i)$ for each $i = 1, 2, \ldots, N$ by dropping the remainder term (i.e. only keeping the first-order term in Taylor's Theorem).

We construct the "difference equation" for Euler's method:

$$w_0 = \alpha \tag{5}$$

$$w_{i+1} = w_i + hf(t_i, w_i) \qquad \text{for each } i = 0, 1, \ldots, N-1 \tag{6}$$

Euler’s Method

Another way to think of Euler’s method is from the definition of the derivative. The approximate derivative over step size h is:

$$y'(t_0) \approx \frac{\Delta y}{\Delta t} = \frac{y(t_0 + h) - y(t_0)}{h}$$

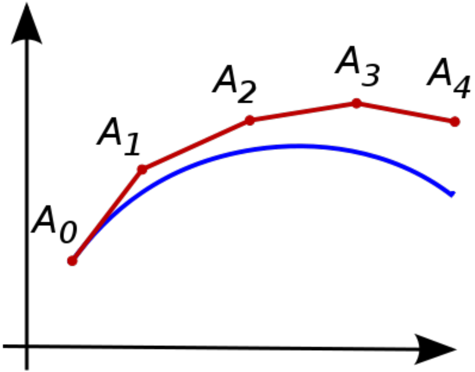
We can rewrite this as:

$$y(t_0 + h) \approx y(t_0) + hy'(t_0)$$

and y' is equal to $f(t, y)$.

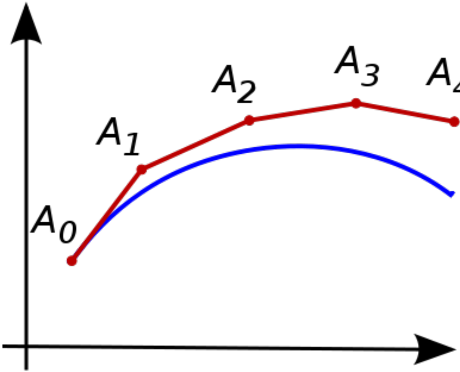
Euler's Method

A differential equation can be thought of as a formula by which the slope of the tangent line to the curve can be computed at any point on the curve, once the position of that point has been calculated.



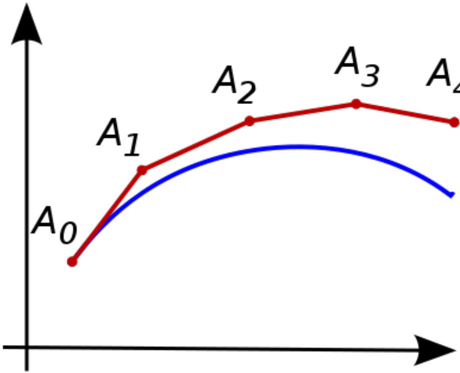
Euler’s Method

The idea is that while the curve is initially unknown, its starting point, which we denote by A_0, is known. Then, from the differential equation, the slope to the curve at A_0 can be computed, and so, the tangent line.



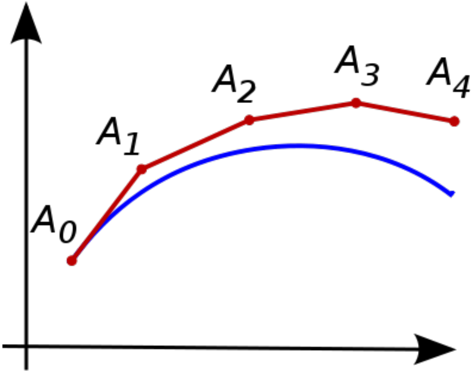
Euler's Method

Take a small step h along that tangent line up to a point A_1. Along this small step, the slope does not change too much, so A_1 will be close to the curve. If we pretend that A_1 is still on the curve, the same reasoning as for the point A_0 can be used. After several steps, a curve is sampled discretely.

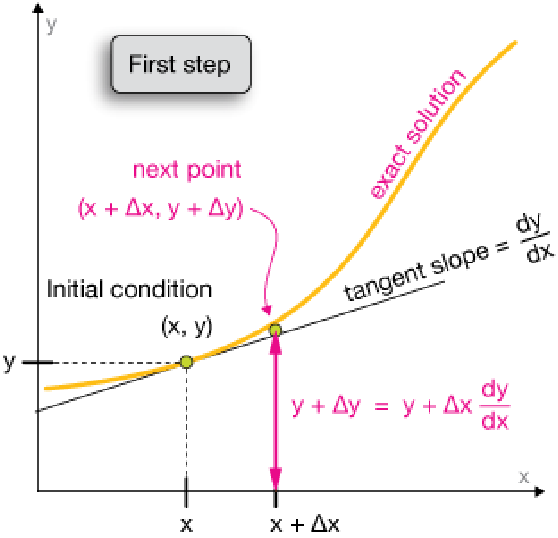


Euler's Method

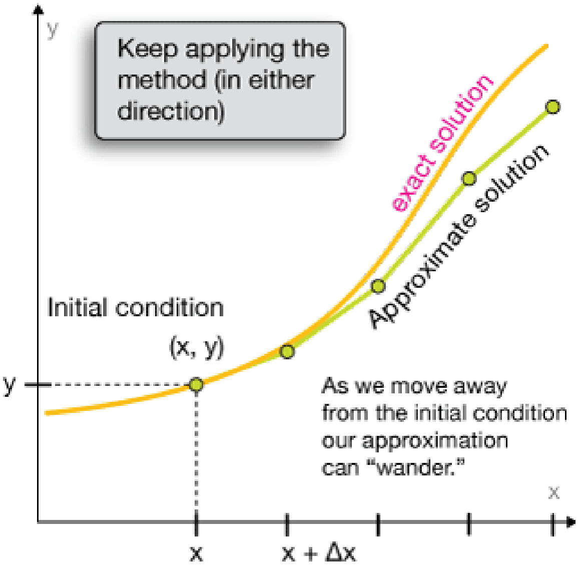
This curve doesn't usually diverge far from the original unknown curve, and the error between the two curves can be made small if the step size is small enough and the interval of computation is finite.



Euler's Method



Euler's Method



To approximate the solution to the initial-value problem:

$$\frac{dy}{dt} = f(t, y) \qquad a \leq t \leq b \qquad y(a) = \alpha \tag{7}$$

at $N + 1$ equally spaced values in the interval $[a, b]$

To approximate the solution to the initial-value problem:

$$\frac{dy}{dt} = f(t, y) \qquad a \leq t \leq b \qquad y(a) = \alpha \tag{7}$$

at $N + 1$ equally spaced values in the interval $[a, b]$

`INPUT`: Endpoints a, b; integer N; initial value α

To approximate the solution to the initial-value problem:

$$\frac{dy}{dt} = f(t, y) \qquad a \leq t \leq b \qquad y(a) = \alpha \tag{7}$$

at $N + 1$ equally spaced values in the interval $[a, b]$

INPUT: Endpoints a, b; integer N; initial value α

OUTPUT: Approximation w of y at the $N + 1$ values of t

Step 1 Set $h = (b - a)/N$
Set $t_0 = a$, $w_0 = \alpha$
OUTPUT t_0 and w_0

Step 2 For $i = 1, 2, \ldots, N$ do Steps 3, 4

Step 3 Set $w_i = w_{i-1} + hf(t_{i-1}, w_{i-1})$
$t_i = a + ih$

Step 4 OUTPUT t_i and w_i

Step 5 STOP

Euler’s Method

Algorithm: Matlab Implementation (10 Steps)

Use Euler’s Method to obtain an approximation to the solution of:

$$y' = y - t^2 + 1, \qquad 0 \le t \le 2, \qquad y(0) = 0.5$$

Use Euler's Method to obtain an approximation to the solution of:

$$y' = y - t^2 + 1, \qquad 0 \leq t \leq 2, \qquad y(0) = 0.5$$

Analytical solution:

$$y(t) = c_1 e^t + t^2 + 2t + 1$$

$$y(0) = c_1 + 0 + 0 + 1 = 0.5$$

$$c_1 = -0.5$$

Euler’s Method

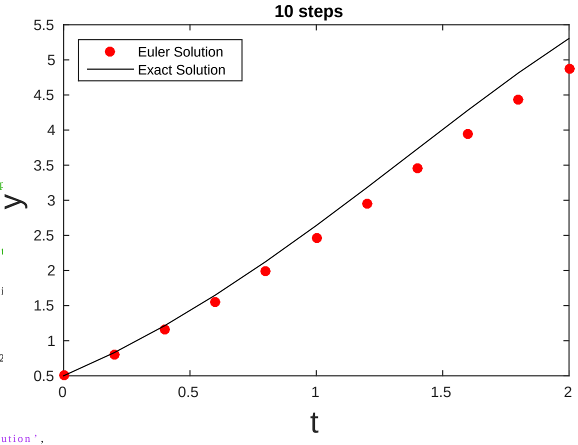
Algorithm: Matlab Implementation (10 Steps)

```
clear % clear variables
close all % close all plots

f_euler = @(t,y) y - t.^2 + 1; % defined f(t,y
    ) as anonymous function

N=10; % number of time steps

a=0; % lower bound of time domain

b=2; % upper bound of time domain

alpha=0.5; % initial value of y

h=(b-a)/N; % step size (time)

t=a:h:b; % calculate time array outside loop,
    for simplicity

w=zeros(size(t)); % preallocate w for speed,
    same size as time variable

w(1)=alpha; % The initial value of y is alpha

for ii=2:N+1 % Matlab begins indexing at 1,
    not 0!

    w(ii)=w(ii-1)+h.*f_euler(t(ii-1),w(ii-1));
        % Euler Difference Eqn

end

    pp=plot(t,w,'rO',t,-0.5.*exp(t)+t.^2+2.*t
        +1,'k-');
    set(pp(1),'MarkerFaceColor','r');
    xlabel('t','FontSize',20);
    ylabel('y','FontSize',20);
    title([num2str(N) ' steps']);
    legend('Euler Solution', 'Exact Solution',
        'Location','NorthWest')
    pdfname='euler_method_example_N10.pdf';
    print('-dpdf',pdfname);
    [~,~]=system(['pdfcrop ' pdfname ' '
        pdfname]);
```

Euler's Method

Algorithm: Matlab Implementation (10 Steps)

```
clear % clear variables
close all % close all plots

f_euler = @(t,y) y - t.^2 + 1; % defined f(t,y
    ) as anonymous function

N=10; % number of time steps

a=0; % lower bound of time domain

b=2; % upper bound of time domain

alpha=0.5; % initial value of y

h=(b-a)/N; % step size (time)

t=a:h:b; % calculate time array outside
    for simplicity

w=zeros(size(t)); % preallocate w for sp
    same size as time variable

w(1)=alpha; % The initial value of y is

for ii=2:N+1 % Matlab begins indexing at
    not 0!

    w(ii)=w(ii-1)+h.*f_euler(t(ii-1),w(i
        % Euler Difference Eqn

end

    pp=plot(t,w,'rO',t,-0.5.*exp(t)+t.^2
        +1,'k-');
    set(pp(1),'MarkerFaceColor','r');
    xlabel('t','FontSize',20);
    ylabel('y','FontSize',20);
    title([num2str(N) ' steps']);
    legend('Euler Solution', 'Exact Solution',
        'Location','NorthWest')
    pdfname='euler_method_example_N10.pdf';
    print('-dpdf',pdfname);
    [~,~]=system(['pdfcrop ' pdfname ' '
        pdfname]);
```



Euler's Method

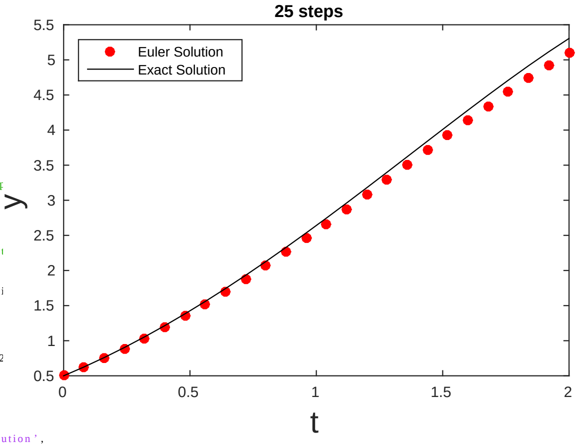
Algorithm: Matlab Implementation (25 Steps)

```matlab
clear % clear variables
close all % close all plots

f_euler = @(t,y) y - t.^2 + 1; % defined f(t,y) as anonymous function

N=25; % number of time steps

a=0; % lower bound of time domain

b=2; % upper bound of time domain

alpha=0.5; % initial value of y

h=(b-a)/N; % step size (time)

t=a:h:b; % calculate time array outside loop, for simplicity

w=zeros(size(t)); % preallocate w for speed, same size as time variable

w(1)=alpha; % The initial value of y is alpha

for ii=2:N+1 % Matlab begins indexing at 1, not 0!

    w(ii)=w(ii-1)+h.*f_euler(t(ii-1),w(ii-1)); % Euler Difference Eqn

end

    pp=plot(t,w,'rO',t,-0.5.*exp(t)+t.^2+2.*t+1,'k-');
    set(pp(1),'MarkerFaceColor','r');
    xlabel('t','FontSize',20);
    ylabel('y','FontSize',20);
    title([num2str(N) ' steps']);
    legend('Euler Solution', 'Exact Solution', 'Location','NorthWest')
    pdfname='euler_method_example_N10.pdf';
    print('-dpdf',pdfname);
    [~,~]=system(['pdfcrop ' pdfname ' ' pdfname]);
```

Euler's Method

Algorithm: Matlab Implementation (25 Steps)

```
clear % clear variables
close all % close all plots

f_euler = @(t,y) y - t.^2 + 1; % defined f(t,y
    ) as anonymous function

N=25; % number of time steps

a=0; % lower bound of time domain

b=2; % upper bound of time domain

alpha=0.5; % initial value of y

h=(b-a)/N; % step size (time)

t=a:h:b; % calculate time array outside
    for simplicity

w=zeros(size(t)); % preallocate w for sp
    same size as time variable

w(1)=alpha; % The initial value of y is

for ii=2:N+1 % Matlab begins indexing at
    not 0!

    w(ii)=w(ii-1)+h.*f_euler(t(ii-1),w(i
        % Euler Difference Eqn

end

    pp=plot(t,w,'rO',t,-0.5.*exp(t)+t.^2
        +1,'k-');
    set(pp(1),'MarkerFaceColor','r');
    xlabel('t','FontSize',20);
    ylabel('y','FontSize',20);
    title([num2str(N) ' steps']);
    legend('Euler Solution', 'Exact Solution',
        'Location','NorthWest')
    pdfname='euler_method_example_N10.pdf';
    print('-dpdf',pdfname);
    [~,~]=system(['pdfcrop ' pdfname ' '
        pdfname]);
```



Euler’s Method

Algorithm: Error Bound

You can see that our Euler estimate (red line) deviates more from the true solution (black line) as time increases.

You can see that our Euler estimate (red line) deviates more from the true solution (black line) as time increases.

We can derive a bound on the error for Euler's method mathematically, if we know an upper bound for the first and second derivatives of the solution (f has Lipschitz constant L and $|y''(t)| \leq M$). For each step i:

$$|y(t_i) - w_i| \leq \frac{hM}{2L}\left(e^{L(t_i - a)} - 1\right) \qquad (8)$$

Lecture Outline

- Introduction to Matlab
- Numerical Solutions to Ordinary Differential Equations
- Euler's Method
- **Taylor Methods**
- Runge-Kutta Methods
- Homework Assignment

Higher Order Euler's Method: Taylor's Method

Since Euler's method was derived using Taylor's Theorem with $n = 1$ to approximate the solution of the differential equation, we can improve the accuracy of our solution by keeping higher order terms.

$$\frac{dy}{dt} = f(t, y) \qquad a \le t \le b \qquad y(a) = \alpha \tag{9}$$

Higher Order Euler's Method: Taylor's Method

Since Euler's method was derived using Taylor's Theorem with $n = 1$ to approximate the solution of the differential equation, we can improve the accuracy of our solution by keeping higher order terms.

$$\frac{dy}{dt} = f(t, y) \qquad a \leq t \leq b \qquad y(a) = \alpha \tag{9}$$

Say the solution has $(n + 1)$ continuous derivatives. We expand the solution $y(t)$ in terms of its nth Taylor polynomial about t_i:

Higher Order Euler's Method: Taylor's Method

Since Euler's method was derived using Taylor's Theorem with $n = 1$ to approximate the solution of the differential equation, we can improve the accuracy of our solution by keeping higher order terms.

$$\frac{dy}{dt} = f(t, y) \qquad a \leq t \leq b \qquad y(a) = \alpha \tag{9}$$

Say the solution has $(n+1)$ continuous derivatives. We expand the solution $y(t)$ in terms of its nth Taylor polynomial about t_i:

$$y(t_{i+1}) = y(t_i) + hy'(t_i) + \frac{h^2}{2} y''(t_i) + \cdots + \frac{h^n}{n!} y^{(n)}(t_i) + \frac{h^{n+1}}{(n+1)!} y^{(n+1)}(\xi_i) \tag{10}$$

for some ξ_i in (t_i, t_{i+1})

Higher Order Euler's Method: Taylor's Method

Since Euler's method was derived using Taylor's Theorem with $n = 1$ to approximate the solution of the differential equation, we can improve the accuracy of our solution by keeping higher order terms.

$$\frac{dy}{dt} = f(t, y) \qquad a \leq t \leq b \qquad y(a) = \alpha \tag{9}$$

Say the solution has $(n + 1)$ continuous derivatives. We expand the solution $y(t)$ in terms of its nth Taylor polynomial about t_i:

$$y(t_{i+1}) = y(t_i) + hy'(t_i) + \frac{h^2}{2}y''(t_i) + \cdots + \frac{h^n}{n!}y^{(n)}(t_i) + \frac{h^{n+1}}{(n+1)!}y^{(n+1)}(\xi_i) \tag{10}$$

for some ξ_i in (t_i, t_{i+1})

This is Taylor's method.

Higher Order Euler's Method: Taylor's Method

Euler's Method:

$$w_0 = \alpha \tag{11}$$

$$w_{i+1} = w_i + hf(t_i, w_i) \qquad \text{for each } i = 0, 1, \ldots, N-1 \tag{12}$$

Taylor's Method of order n:

$$w_0 = \alpha \tag{13}$$

$$w_{i+1} = w_i + hT^{(n)}(t_i, w_i) \qquad \text{for each } i = 0, 1, \ldots, N-1 \tag{14}$$

where:

$$T^{(n)}(t_i, w_i) = f(t_i, w_i) + \frac{h}{2} f'(t_i, w_i) + \cdots + \frac{h^{n-1}}{n!} f^{(n-1)}(t_i, w_i) \tag{15}$$

Euler's method = Taylor's method of order one.

Use Taylor's Method of fourth order to obtain an approximation to the solution of:

$$y' = y - t^2 + 1, \qquad 0 \le t \le 2, \qquad y(0) = 0.5$$

Analytical solution:

$$y(t) = -0.5e^t + t^2 + 2t + 1$$

Use Taylor's Method of fourth order to obtain an approximation to the solution of:

$$y' = y - t^2 + 1, \qquad 0 \leq t \leq 2, \qquad y(0) = 0.5$$

We have to calculate analytical derivatives of f:

$$f' = \frac{df}{dt} = \frac{\partial f}{\partial t}\frac{dt}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$

$$f' = \frac{df}{dt} = (-2t)(1) + (1)(y')$$

$$f' = \frac{df}{dt} = y - t^2 + 1 - 2t$$

$$f'' = y - t^2 - 2t - 1$$

$$f''' = y - t^2 - 2t - 1$$

Taylor's Method

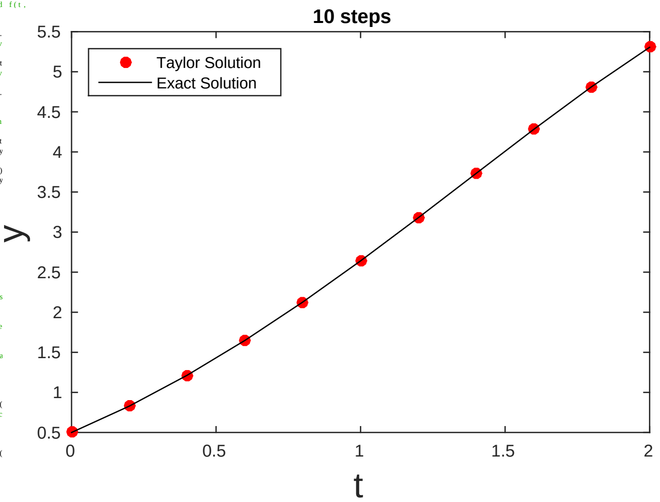
Algorithm: Matlab Implementation (fourth order)

```
clear % clear varaibles from workspace
close all % close all plots

f_taylor = @(t,y) y - t.^2 + 1; % defined f(t,y) as anonymous function

f_taylor_first_derivative = @(t,y) y - t.^2 + 1 - 2.*t; % Analytical first derivative

f_taylor_second_derivative = @(t,y) y - t.^2 - 2.*t-1; % Analytical second derivative

f_taylor_third_derivative = @(t,y) y - t.^2 - 2.*t-1; % Analytical third derivative

% Taylor's method factor for the difference equation:
taylor_fourth_order = @(t,y,h) f_taylor(t,y) + (h./2).*f_taylor_first_derivative(t,y) + ...
(h.^2/6).*f_taylor_second_derivative(t,y) + (h.^3/24).*f_taylor_third_derivative(t,y);

N=10; % number of time steps

a=0; % lower bound of time domain

b=2; % upper bound of time domain

alpha=0.5; % initial value of y

h=(b-a)/N; % step size (time)

t=a:h:b; % Calculate the time vector outside of the loop, for simplicity

w=zeros(size(t)); % preallocate w for speed, same size as time variable

w(1)=alpha; % The initial value of y is alpha

for ii=2:N+1 % Matlab begins indexing at 1, not 0!

    w(ii)=w(ii-1)+h.*taylor_fourth_order(t(ii-1),w(ii-1),h); % Taylor Difference Eqn

end

    pp=plot(t,w,'rO',t,(t+1).^2-0.5.*exp(t),'k-');
    set(pp(1),'MarkerFaceColor','r');
    xlabel('t','FontSize',20);
    ylabel('y','FontSize',20);
    title([num2str(N) ' steps']);
    legend('Taylor Solution', 'Exact Solution','Location','NorthWest')
    pdfname='taylor_method_example_N10.pdf';
    print('-dpdf',pdfname);
    [~,~]=system(['pdfcrop ' pdfname ' ' pdfname]);
```

Taylor's Method

Algorithm: Matlab Implementation (fourth order)

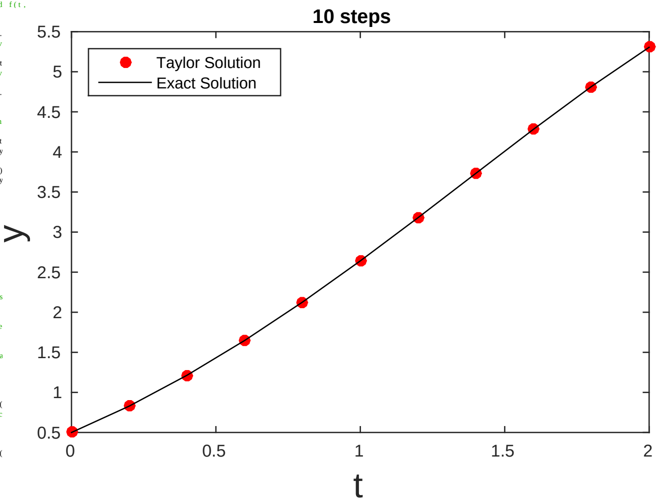
```
clear % clear varaibles from workspace
close all % close all plots

f_taylor = @(t,y) y - t.^2 + 1; % defined f(t,
    y) as anonymous function

f_taylor_first_derivative = @(t,y) y - t.
    1 - 2.*t; % Analytical first derivativ

f_taylor_second_derivative = @(t,y) y - t
    2.*t-1; % Analytical second derivativ

f_taylor_third_derivative = @(t,y) y - t.
    2.*t-1; % Analytical third derivative

% Taylor's method factor for the differen
    equation:
taylor_fourth_order = @(t,y,h) f_taylor(t
    (h./2).*f_taylor_first_derivative(t,y
    ...
(h.^2/6).*f_taylor_second_derivative(t,y)
    .^3/24).*f_taylor_third_derivative(t,y

N=10; % number of time steps

a=0; % lower bound of time domain

b=2; % upper bound of time domain

alpha=0.5; % initial value of y

h=(b-a)/N; % step size (time)

t=a:h:b; % Calculate the time vector outs
    of the loop, for simplicity

w=zeros(size(t)); % preallocate w for spe
    same size as time variable

w(1)=alpha; % The initial value of y is a

for ii=2:N+1 % Matlab begins indexing at
    not 0!

    w(ii)=w(ii-1)+h.*taylor_fourth_order(
        -1),w(ii-1),h); % Taylor Differenc

end

    pp=plot(t,w,'rO',t,(t+1).^2-0.5.*exp(
        -');
    set(pp(1),'MarkerFaceColor','r');
    xlabel('t','FontSize',20);
    ylabel('y','FontSize',20);
    title([num2str(N) ' steps']);
    legend('Taylor Solution', 'Exact Solution'
        ,'Location','NorthWest')
    pdfname='taylor_method_example_N10.pdf';
    print('-dpdf',pdfname);
    [~,~]=system(['pdfcrop ' pdfname ' '
        pdfname]);
```



Taylor's Method

Algorithm: Matlab Implementation (fourth order)

```
clear % clear varaibles from workspace
close all % close all plots

f_taylor = @(t,y) y - t.^2 + 1; % defined f(t,
    y) as anonymous function

f_taylor_first_derivative = @(t,y) y - t.
    1 - 2.*t; % Analytical first derivativ

f_taylor_second_derivative = @(t,y) y - t
     2.*t-1; % Analytical second derivativ

f_taylor_third_derivative = @(t,y) y - t.
    2.*t-1; % Analytical third derivative

% Taylor's method factor for the differen
    equation:
taylor_fourth_order = @(t,y,h) f_taylor(t
     (h./2).*f_taylor_first_derivative(t,y
    ...
(h.^2/6).*f_taylor_second_derivative(t,y)
    .^3/24).*f_taylor_third_derivative(t,y

N=10; % number of time steps

a=0; % lower bound of time domain

b=2; % upper bound of time domain

alpha=0.5; % initial value of y

h=(b-a)/N; % step size (time)

t=a:h:b; % Calculate the time vector outs
    of the loop, for simplicity

w=zeros(size(t)); % preallocate w for spe
    same size as time variable

w(1)=alpha; % The initial value of y is a

for ii=2:N+1 % Matlab begins indexing at
    not 0!

     w(ii)=w(ii-1)+h.*taylor_fourth_order(
         -1),w(ii-1),h); % Taylor Differenc

end

     pp=plot(t,w,'rO',t,(t+1).^2-0.5.*exp(
         -');
     set(pp(1),'MarkerFaceColor','r');
     xlabel('t','FontSize',20);
     ylabel('y','FontSize',20);
     title([num2str(N) ' steps']);
     legend('Taylor Solution', 'Exact Solution'
         ,'Location','NorthWest')
     pdfname='taylor_method_example_N10.pdf';
     print('-dpdf',pdfname);
     [~,~]=system(['pdfcrop ' pdfname ' '
         pdfname]);
```

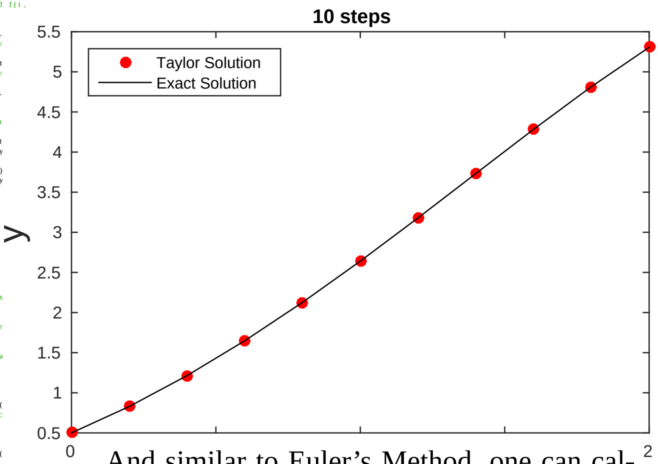


Much more accurate than Euler's Method!

Taylor's Method

Algorithm: Matlab Implementation (fourth order)

```
clear % clear varaibles from workspace
close all % close all plots

f_taylor = @(t,y) y - t.^2 + 1; % defined f(t,
   y) as anonymous function

f_taylor_first_derivative = @(t,y) y - t.
   1 - 2.*t; % Analytical first derivativ

f_taylor_second_derivative = @(t,y) y - t
    2.*t-1; % Analytical second derivativ

f_taylor_third_derivative = @(t,y) y - t.
   2.*t-1; % Analytical third derivative

% Taylor's method factor for the differen
   equation:
taylor_fourth_order = @(t,y,h) f_taylor(t
    (h./2).*f_taylor_first_derivative(t,y
   ...
(h.^2/6).*f_taylor_second_derivative(t,y)
   .^3/24).*f_taylor_third_derivative(t,y

N=10; % number of time steps

a=0; % lower bound of time domain

b=2; % upper bound of time domain

alpha=0.5; % initial value of y

h=(b-a)/N; % step size (time)

t=a:h:b; % Calculate the time vector outs
   of the loop, for simplicity

w=zeros(size(t)); % preallocate w for spe
   same size as time variable

w(1)=alpha; % The initial value of y is a

for ii=2:N+1 % Matlab begins indexing at
   not 0!

    w(ii)=w(ii-1)+h.*taylor_fourth_order(
       -1),w(ii-1),h); % Taylor Differenc

end

    pp=plot(t,w,'rO',t,(t+1).^2-0.5.*exp(
       -');
    set(pp(1),'MarkerFaceColor','r');
    xlabel('t','FontSize',20);
    ylabel('y','FontSize',20);
    title([num2str(N) ' steps']);
    legend('Taylor Solution', 'Exact Solution'
       ,'Location','NorthWest')
    pdfname='taylor_method_example_N10.pdf';
    print('-dpdf',pdfname);
    [~,~]=system(['pdfcrop ' pdfname ' '
       pdfname]);
```



And similar to Euler's Method, one can calculate bounds on the error *if* upper bounds on the derivatives are known.

Lecture Outline

Runge-Kutta vs. Taylor

- The Taylor methods are good because they have high-order truncation errors: you can make them more accurate by adding more terms.

Runge-Kutta vs. Taylor

- The Taylor methods are good because they have high-order truncation errors: you can make them more accurate by adding more terms.
- But to add more terms, you need to compute the derivatives of $f(t, y)$, which can be complicated and time consuming (or impossible!)

Runge-Kutta vs. Taylor

- The Taylor methods are good because they have high-order truncation errors: you can make them more accurate by adding more terms.
- But to add more terms, you need to compute the derivatives of $f(t, y)$, which can be complicated and time consuming (or impossible!)
- **Runge-Kutta** methods also have high-order truncation errors while eliminating the need to compute and analytical derivatives of $f(t, y)$

Runge-Kutta Methods

Runge-Kutta Methods substitute analytical derivatives of $f(t, y)$ with an approximation of the derivatives from the Taylor polynomial expansions, retaining orders such that the error (the remainder R_n) is sufficiently small (compared to the order of the method).

- Requires Taylor's Theorem in two variables (see advanced calculus textbooks)

Runge-Kutta Methods

Runge-Kutta Methods substitute analytical derivatives of $f(t, y)$ with an approximation of the derivatives from the Taylor polynomial expansions, retaining orders such that the error (the remainder R_n) is sufficiently small (compared to the order of the method).

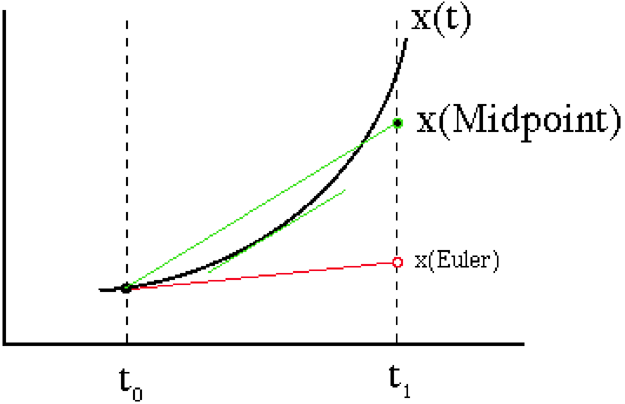
- Requires Taylor's Theorem in two variables (see advanced calculus textbooks)

The **Midpoint method** (a specific Runge-Kutta method) replaces $T^{(2)}$ by $f(t + (h/2), y + (h/2)f(t, y))$. It has local truncation error $O(h^3)$.

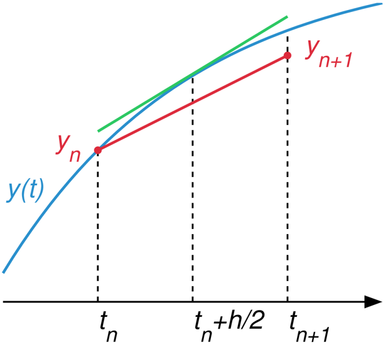
Runge-Kutta Methods

Midpoint Method

Midpoint method: This is a refinement of the Euler method, which uses the midpoint derivative instead of the start-point derivative, increasing the algorithm's accuracy:



Midpoint method: This is a refinement of the Euler method, which uses the midpoint derivative instead of the start-point derivative, increasing the algorithm's accuracy:



The midpoint method computes y_{n+1} so that the red chord is approximately parallel to the tangent line at the midpoint (the green line).

Midpoint method: This is a refinement of the Euler method, which uses the midpoint derivative instead of the first endpoint derivative, increasing the algorithm's accuracy:

$$y(t+h) \approx y(t) + hf\left(t + \frac{h}{2}, y\left(t + \frac{h}{2}\right)\right)$$

Midpoint method: This is a refinement of the Euler method, which uses the midpoint derivative instead of the first endpoint derivative, increasing the algorithm's accuracy:

$$y(t+h) \approx y(t) + hf\left(t + \frac{h}{2}, y\left(t + \frac{h}{2}\right)\right)$$

One cannot use this equation to find $y(t+h)$ as one does not know y at $t + h/2$. So we approximate $y(t + h/2)$ using a Taylor expansion (this is the *Runge-Kutta* step):

$$y\left(t + \frac{h}{2}\right) \approx y(t) + \frac{h}{2}y'(t) = y(t) + \frac{h}{2}f(t, y(t))$$

Midpoint method: This is a refinement of the Euler method, which uses the midpoint derivative instead of the first endpoint derivative, increasing the algorithm's accuracy:

$$y(t+h) \approx y(t) + hf\left(t + \frac{h}{2}, y\left(t + \frac{h}{2}\right)\right)$$

One cannot use this equation to find $y(t+h)$ as one does not know y at $t + h/2$. So we approximate $y(t + h/2)$ using a Taylor expansion (this is the *Runge-Kutta* step):

$$y\left(t + \frac{h}{2}\right) \approx y(t) + \frac{h}{2}y'(t) = y(t) + \frac{h}{2}f(t, y(t))$$

which gives us the Midpoint method:

$$y(t+h) \approx y(t) + hf\left(t + \frac{h}{2}, y(t) + \frac{h}{2}f(t, y(t))\right)$$

Midpoint method (a Runge-Kutta method of order two): The *difference equation* for the midpoint method is given by:

$$\begin{aligned} w_0 &= \alpha \\ k_1 &= hf(t_i, w_i) \\ k_2 &= hf\left(t_i + \frac{h}{2}, w_i + \frac{1}{2}k_1\right) \\ w_{i+1} &= w_i + k_2 \end{aligned}$$

for each $i = 0, 1, \ldots, N-1$.

The total accumulated error is $O(h^2)$.

Runge-Kutta Methods: RK4

Order n Runge-Kutta methods take the Taylor method of order n and approximate the analytical derivatives with numerical derivatives.

Runge-Kutta Methods: RK4

Order n Runge-Kutta methods take the Taylor method of order n and approximate the analytical derivatives with numerical derivatives.

Reminder: **Taylor's Method** of order n:

$$w_0 = \alpha \tag{16}$$

$$w_{i+1} = w_i + hT^{(n)}(t_i, w_i) \qquad \text{for each } i = 0, 1, \ldots, N-1 \tag{17}$$

where:

$$T^{(n)}(t_i, w_i) = f(t_i, w_i) + \frac{h}{2} f'(t_i, w_i) + \cdots + \frac{h^{n-1}}{n!} f^{(n-1)}(t_i, w_i) \tag{18}$$

Runge-Kutta Methods: RK4

The Runge-Kutta Order Four method is also known as "RK4", "classical Runge–Kutta method" or simply "*the* Runge–Kutta method". Its difference equation is given by:

$$
\begin{aligned}
w_0 &= \alpha \\
k_1 &= hf(t_i, w_i) \\
k_2 &= hf\left(t_i + \frac{h}{2}, w_i + \frac{1}{2}k_1\right) \\
k_3 &= hf\left(t_i + \frac{h}{2}, w_i + \frac{1}{2}k_2\right) \\
k_4 &= hf\left(t_{i+1}, w_i + k_3\right) \\
w_{i+1} &= w_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)
\end{aligned}
$$

for each $i = 0, 1, \ldots, N-1$. The total accumulated error is $O(h^4)$.

Runge-Kutta Methods: RK4

- We could implement this ourselves in Matlab (or any other language) just like we did with the Taylor and Euler methods.

Runge-Kutta Methods: RK4

- We could implement this ourselves in Matlab (or any other language) just like we did with the Taylor and Euler methods.
- But the point of using Matlab is that we don't have to!

Runge-Kutta Methods: RK4

- We could implement this ourselves in Matlab (or any other language) just like we did with the Taylor and Euler methods.
- But the point of using Matlab is that we don't have to!
- Many algorithms are already coded and ready for use in Matlab, sometimes via additional-cost toolboxes.

Runge-Kutta Methods: RK4

- We could implement this ourselves in Matlab (or any other language) just like we did with the Taylor and Euler methods.
- But the point of using Matlab is that we don't have to!
- Many algorithms are already coded and ready for use in Matlab, sometimes via additional-cost toolboxes.
- The RK4 algorithm is implemented in the Matlab function `ode45`.

Runge-Kutta Methods: RK4

- We could implement this ourselves in Matlab (or any other language) just like we did with the Taylor and Euler methods.
- But the point of using Matlab is that we don't have to!
- Many algorithms are already coded and ready for use in Matlab, sometimes via additional-cost toolboxes.
- The RK4 algorithm is implemented in the Matlab function `ode45`.
- You can see the source code for this function by `dbtype ode45.m` (for file location: `which ode45.m`).

Matlab ode45

help ode45:

```
[TOUT,YOUT] = ode45(ODEFUN,TSPAN,Y0) with TSPAN = [T0
TFINAL] integrates the system of differential
equations y' = f(t,y) from time T0 to TFINAL with
initial conditions Y0.  ODEFUN is a function handle.
For a scalar T and a vector Y, ODEFUN(T,Y) must return
a column vector corresponding to f(t,y).  Each row in
the solution array YOUT corresponds to a time returned
in the column vector TOUT. To obtain solutions at
specific times T0,T1,...,TFINAL (all increasing or
all decreasing), use TSPAN = [T0 T1 ... TFINAL].
```

Lecture Outline

Homework

- Please complete Matlab Onramp before the next class: February 22
- The following **homework assignment** is due at 4pm March 1st (**two weeks from today**)
  - Please email me (`rogliore@physics.wustl.edu`) your completed homework assignment as a Matlab script file (.m) (or multiple Matlab script files)
- The next class period, February 22, will be a time where you can work on the code and I will be available to answer any code-level questions you have about the assignment
- The third class period, March 1, we will discuss the HW assignment and further applications of these ideas

Homework Assignment

Edward Lorenz, a meterologist, created a simplified mathematical model for nonlinear atmospheric thermal convection in 1962. Lorenz's model frequently arises in other types of systems, e.g. dynamos and electrical circuits. Now known as the Lorenz equations, this model is a system of three ordinary differential equations:

$$\frac{dx}{dt} = \sigma(y - x),$$
$$\frac{dy}{dt} = x(\rho - z) - y,$$
$$\frac{dz}{dt} = xy - \beta z.$$

Note the last two equations involve quadratic nonlinearities. The intensity of the fluid motion is parameterized by the variable x; y and z are related to temperature variations in the horizontal and vertical directions.

Homework Assignment (continued)

Use Matlab's RK4 solver `ode45` to solve this system of ODEs with the following starting points and parameters.

1. With $\sigma = 1$, $\beta = 1$, and $\rho = 1$, solve the system of Lorenz Equations for $x(t = 0) = 1$, $y(t = 0) = 1$, and $z(t = 0) = 1$. Plot the orbit of the solution as a three-dimensional plot for times 0–100.
2. For the Earth's atmosphere reasonable values are $\sigma = 10$ and $\beta = 8/3$. Also set $\rho = 28$; and using starting values: $x(t = 0) = 5$, $y(t = 0) = 5$, and $z(t = 0) = 5$; solve the system of Lorenz Equations for $t = [0, 20]$. Plot the orbit of the solution as a three-dimensional plot for t=0–20. Also plot z vs. x. Do any of the orbits that appear to overlap in this plot actually overlap when viewed in the three-dimensional plot?
3. Plot x, y, and z vs. time on one graph using Matlab's `subplot` function.

Homework Assignment (continued)

4. Use the same parameters as in #2 but add a very small number (e.g. 10^{-6}) to one of the starting values. Plot x, y, and z vs. time for *both* of these curves (one red, one blue). Solve the equation for longer times to see when the two solutions diverge from each other.
5. Find a value of ρ (while keeping $\sigma = 10$ and $\beta = 8/3$) such that the solution does not depend sensitively on the initial values. Plot both curves for x, y, and z vs. time as you did in #4.
6. For ρ=70, $\sigma = 10$, $\beta = 8/3$, initial starting value (5,5,5), over a time range 0–50, calculate and plot one solution using the default maximum step size for `ode45`: $0.1 \times (t_{\text{final}} - t_{\text{initial}})$, and another solution for $1/1000^{\text{th}}$ of the default. Is this behavior related to the sensitivity on initial starting values you explored in #4?

Additional Slide: Lipschitz Condition

A function $f(t, y)$ is said to satisfy a **Lipschitz Condition** in the variable y on a set D in $\mathbb{R}^2$ if a constant $L > 0$ exists with the property that

$$|f(t, y_1) - f(t, y_2)| \leq L|y_1 - y_2|$$

whenever (t, y_1), (t, y_2) exist in D. The constant L is called a Lipschitz constant for f.

Example: If $D = \{(t, y)|1 \leq t \leq 2, -3 \leq y \leq 4\}$ and $f(t, y) = t|y|$, then for each pair of points (t, y_1) and (t, y_2) in D we have

$$|f(t, y_1) - f(t, y_2)| = |t|y_1| - t|y_2|| = |t|||y_1| - |y_2|| \leq 2|y_1 - y_2|$$

Thus, f satisfies a Lipschitz condition on D in the variable y with Lipschitz constant $L = 2$.