Numerical Solution of Differential Equations

Part I

Introduction

Why Differential Equations ?

- Differential equations are used to simulate dynamic systems such as *robots, airplanes, hurricanes, internet traffic, social networks, etc.*
- Numerical solvers for differential equations form the *core of all simulators*

- Simulation software:
  - Matlab/Simulink/StateFlow (commercial)
    - http://www.mathworks.com/videos/hardware-in-the-loop-hil-testing-68840.html (Hardware in the Loop)
  - Scicos/Scilab (open source)
    - http://www.scicos.org/

Differential Equations

- Often we understand how pieces of a system "CHANGE" but we don't really understand how the system "works"

- If we observe the "sum" of changes over time we can see how the system "works"

- Change = derivative
- Sum = integration

Example: Predator-Prey

- The population levels of a predator-prey ecosystem (*e.g.*, elk-wolves) can be modeled as a pair of ***coupled first order differential equations***:

- ChangeInElkPopulation = ElkGrowthRate * ElkPopulation – WolfPredationRate * ElkPopulation* WolfPopulation change

- ChangeInWolfPopulation = −WolfDecayRate * WolfPopulation + WolfGrowthRate * WolfPopulation* ElkPopulation change

Example: Predator-Prey

The unknowns are the functions $N_1(t)$ and $N_2(t)$

- Given:
  - N_1 = ElkPopulation
  - N_2 = WolfPopulation
  - r_1 = ElkGrowthRate (in absence of Wolves)
  - r_2 = WolfDecayRate (in absence of Elk)
  - b_1 = WolfPredationRate
  - b_2 = WolfGrowthRate

- Then
  - $dN_1/dt = r_1N_1 - b_1N_1N_2$
  - $dN_2/dt = -r_2N_2 + b_2N_2N_1$

Example: Predator-Prey

- Given:
  - r_1 = ElkGrowthRate = 2
  - r_2 = WolfDecayRate = 1
  - b_1 = WolfPredationRate = 1.2
  - b_2 = WolfGrowthRate = 0.9

  - $dN_1/dt = r_1N_1 - b_1N_1N_2$ change
  - $dN_2/dt = -r_2N_2 + b_2N_2N_1$ change

- Initial Value:
  - $N_1(0)$ = ElkPopulation = 1.0
  - $N_2(0)$ = WolfPopulation = 0.5

Example: Predator-Prey matlab

- pred_prey_odes.m

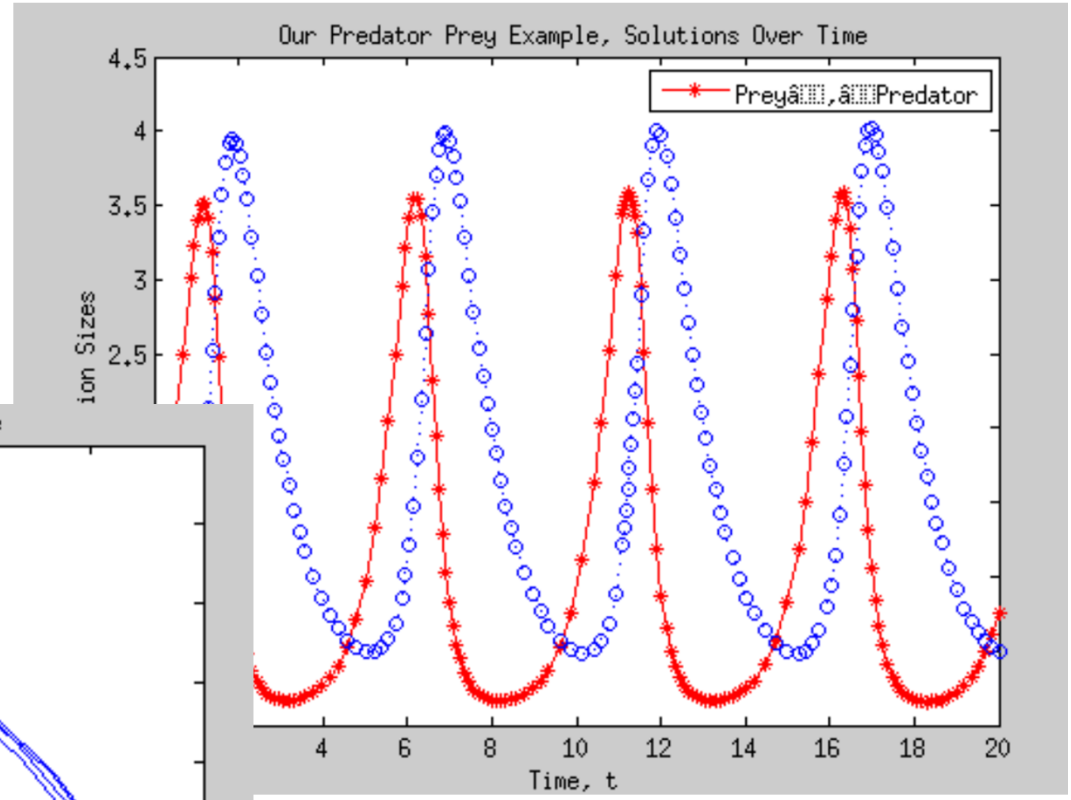
```
function pred_prey_odes = pred_prey_odes(t,x)
pred_prey_odes(1) = 2*x(1) - 1.2*x(1)*x(2);
pred_prey_odes(2) = -1*x(2) + 0.9*x(1)*x(2);
pred_prey_odes = [pred_prey_odes(1) pred_prey_odes(2)]';
```

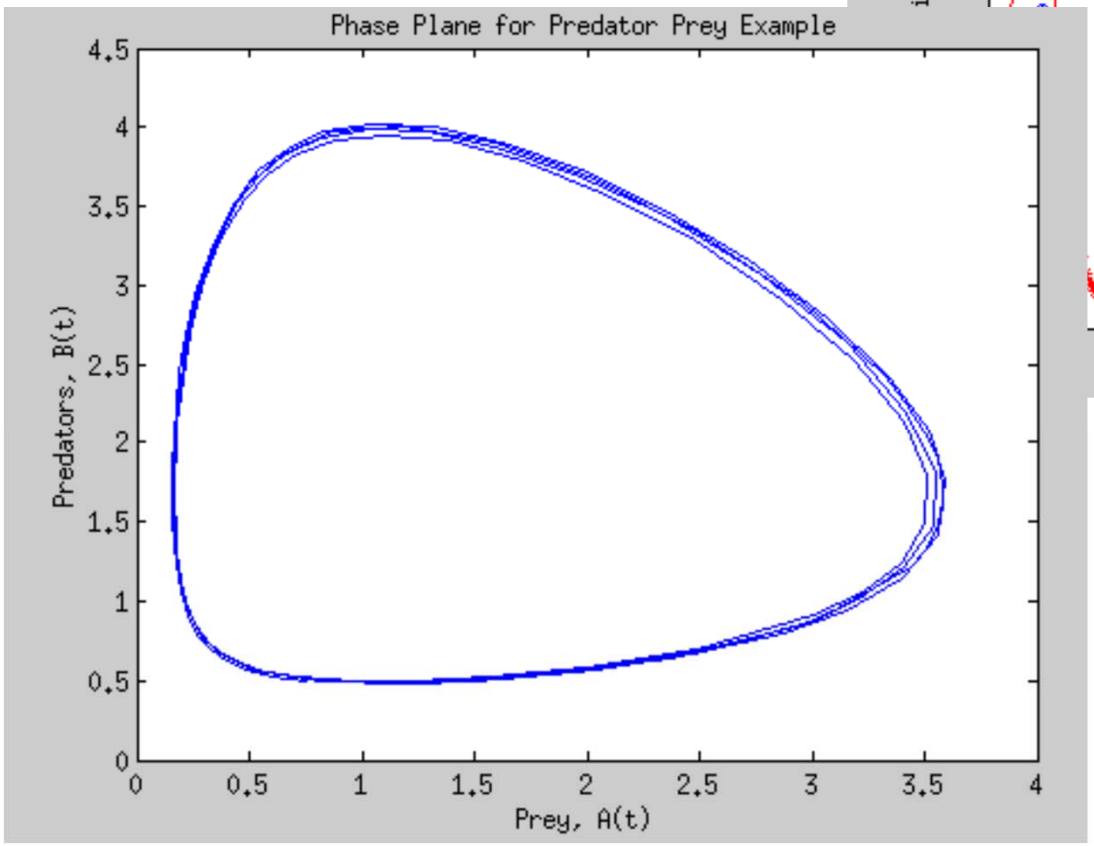
- PredPreyScript.m

```
t0 = 0;
tf = 20;
init_vals = [1; 0.5];
[t,x] = ode45('pred_prey_odes',[t0,tf],init_vals);
A = x(:,1);
B = x(:,2);
plot(t,A,'r*-',t,B,'bo:')
xlabel('Time, t')
ylabel('Population Sizes')
title('Our Predator Prey Example, Solutions Over Time')
legend('Prey’,’Predator')
figure
plot(A,B)
xlabel('Prey, A(t)')
ylabel('Predators, B(t)')
title('Phase Plane for Predator Prey Example')
```

Example: Predator-Prey Ecosystem

- How it works!





State Space form of Differential Equations

Key Result:

- Any ***n^{th} order differential*** equation

$$y^{(n)}(t) = f(t, y, y^{(1)}, y^{(2)}, \ldots, y^{(n-1)}, u(t))$$

can be written as a *system of 1st order differential equations*

$$\dot{x}_1(t) = f_1(x_1(t), x_2(t), \ldots, x_n(t), u(t))$$
$$\dot{x}_2(t) = f_2(x_1(t), x_2(t), \ldots, x_n(t), u(t))$$
$$\dot{x}_3(t) = f_3(x_1(t), x_2(t), \ldots, x_n(t), u(t))$$
$$\vdots$$
$$\dot{x}_n(t) = f_n(x_1(t), x_2(t), \ldots, x_n(t), u(t))$$

Important: The algorithms presented here can only be used if the system is expressed as a systems of 1st order differential equations

State Space form: Simple Case

- To write the *n^{th} order differential* equation

$$y^{(n)}(t) = f(t, y, y^{(1)}, y^{(2)}, \ldots, y^{(n-1)}, u(t))$$

as a *system of 1st order differential equations* we can proceed as follows:

1. **Define** a **new** set of **variables** ***$x_i(t)$*** called **"state variables"**

$$x_1(t) = y(t)$$
$$x_2(t) = y^{(1)}(t)$$
$$\vdots$$
$$x_n(t) = y^{(n-1)}(t)$$

State Space form: Simple Case

2. **Rewrite** the *n^{th} order* differential equation **as a system of 1st order equations with respect** to the **new** (state) **variables**:

$$
\begin{aligned}
\dot{x}_1(t) &= x_2(t) \\
\dot{x}_2(t) &= x_3(t) \\
\dot{x}_3(t) &= x_4(t) \\
&\vdots \\
\dot{x}_n(t) &= f(t, x_1(t), x_2(t), \ldots, x_n(t), u(t))
\end{aligned}
$$

Vectors and Vector Functions

- About vector notation:
  - **Bold** types are used for **vectors** in these notes

Vectors

$$\dot{\boldsymbol{x}}(t) = \frac{d}{dt}\begin{bmatrix} x_1(t) \\ x_2(t) \\ \vdots \\ x_n(t) \end{bmatrix}$$

$$\boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), t) = \begin{bmatrix} f_1(\boldsymbol{x}(t), \boldsymbol{u}(t), t) \\ f_2(\boldsymbol{x}(t), \boldsymbol{u}(t), t) \\ \vdots \\ f_n(\boldsymbol{x}(t), \boldsymbol{u}(t), t) \end{bmatrix}$$

Ex: Elk-Wolf-Hunter Ecosystem

$$\dot{x}_1(t) = b_1 x_1(t) - c_1 x_2(t) x_1(t) + k_1 u_1(t)$$
$$\dot{x}_2(t) = -b_2 x_2(t) + c_2 x_1(t) x_2(t) + k_2 u_2(t)$$

- x_1: population of *preys*
- x_2: population of *predators*
- b_1: growth rate of predators
- b_2: growth rate of preys
- c_1: decrease in preys due to predators
- c_2: decrease in predators due to preys
- k_1, k_2: hunters success constant

Model Parameters:

$$b_1 = 1.0, c_1 = 0.20, k_1 = 0.5$$
$$b_2 = 0.4, c_2 = 0.01, k_2 = 0.01$$

Ex: Elk-Wolf-Hunter Ecosystem

$$\dot{x}_1(t) = b_1 x_1(t) - c_1 x_2(t) x_1(t) + k_1 u_1(t)$$
$$\dot{x}_2(t) = -b_2 x_2(t) + c_2 x_1(t) x_2(t) + k_2 u_2(t)$$

- In this case the relevant vectors are

$$\boldsymbol{x}(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix}, \boldsymbol{u}(t) = \begin{bmatrix} u_1(t) \\ u_2(t) \end{bmatrix}, \boldsymbol{f}(\cdot) = \begin{bmatrix} f_1(\cdot) \\ f_2(\cdot) \end{bmatrix}$$

where the function vector is

$$f_1(\cdot) = b_1 x_1(t) - c_1 x_2(t) x_1(t) + k_1 u_1(t)$$
$$f_2(\cdot) = -b_2 x_2(t) + c_2 x_1(t) x_2(t) + k_2 u_2(t)$$

- To evaluate each ***vector*** (in C) you need a ***"for loop"***

Summary: State-Space form

- Any ***n^{th} order differential*** equation can be written as a ***system of n 1st order differential equations*** *in "state-space form"*

- In vector notation: $\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), t)$

- Explicitly:

$$\begin{aligned}
\dot{x}_1(t) &= f_1(\boldsymbol{x}(t), \boldsymbol{u}(t), t) \\
\dot{x}_2(t) &= f_2(\boldsymbol{x}(t), \boldsymbol{u}(t), t) \\
&\vdots \\
\dot{x}_n(t) &= f_n(\boldsymbol{x}(t), \boldsymbol{u}(t), t)
\end{aligned}$$

Solving Differential Equations Numerically

The **original** problem

Find a function *x(t)* that satisfies the *differential equation for* $t \in [t_o, t_f)$

$$\dot{x}(t) = f(t, x(t), u(t)), \quad x(t_o) = x_o$$

The **numerical** problem

Find a function *x(t)* that at a ***finite number of points***, satisfies

$t_N = t_f$

$$\begin{aligned} \dot{x}(t_i) &= f(t_i, x(t_i), u(t_i)), \quad x(t_o) = x_o, \\ i &= 0, 1, \ldots, N \end{aligned}$$

Explicit vs. Implicit Methods

- We **could** solve the DiffEq by ***direct integration***

$$x(t_{k+1}) = x(t_k) + \int_{t_k}^{t_{k+1}} f(\tau, x(\tau))d\tau$$

- **Or** using *forward rectangular integration* we obtain a simple "explicit method"

$$x(t_{k+1}) \approx x(t_k) + h_k f(t_k, x(t_k))$$

$h_k = t_{k+1} - t_k$

- **Or** using *backward rectangular integration* leads to an "implicit method"

$$x(t_{k+1}) \approx x(t_k) + h_k f(t_{k+1}, x(t_{k+1}))$$

Implicit: The Right-hand side depends on what we want to find

Summary

- Differential equations are solved by " numerical integration"

$$x(t_{k+1}) = x(t_k) + \int_{t_k}^{t_{k+1}} f(\tau, x(\tau))d\tau$$

- The choice of ***numerical integration*** leads to different ***"differential equations solvers"*** and determines
  - If the method is ***explicit or implicit***
  - The ***accuracy*** of the algorithm

- In this course we will discus the most important explicit methods

Numerical Solution of Differential Equations

Part II

One Step Methods

One Step and Multistep Methods

- We will start studying ***one step algorithms*** (methods)

- In *one step* methods *the computation of* $x(t_{i+1})$ ***requires knowing*** $x(t_i)$,the previous value.

- Multistep methods use more neighboring values to update the solution

Recall: Explicit vs. Implicit Integration

- We could solve the differential equation

- Using *forward rectangular integration* we obtain a simple "explicit method"

$$x(t_{k+1}) \approx x(t_k) + h_k f(t_k, x(t_k))$$

$h_k = t_{k+1} - t_k$

- Using *backward rectangular integration* leads to an "implicit method"

$$x(t_{k+1}) \approx x(t_k) + h_k f(t_{k+1}, x(t_{k+1}))$$

Implicit: The Right-hand side depends on what we want to find

Forward Euler Method (One Step)

- Consider a 1st order *ordinary differential equation* (ODE)

$$\dot{x}(t) = f(t, x(t)), \quad \boxed{x(t_o) = x_o}$$

Initial Conditions

- ***Choose*** a *fixed* (constant) *time* ***step h***, that is, set $t_k = k\,h$
- Suppose that we already have the solution at time t_k and want to find it a time t_{k+1} (update)
- Obtain the update equation (algorithm) by ***numerical integration*** *over* (t_k, t_{k+1}) using the **forward rectangular** rule

Euler's Method: Derivation

- By direct integration in the interval (t_k, t_{k+1})

$$x(t_{k+1}) = x(t_k) + \int_{t_k}^{t_{k+1}} f(\tau, x(\tau))d\tau$$

- Using *forward rectangular integration* and setting $h = t_{k+1} - t_k$

$$\int_{t_k}^{t_{k+1}} f(\tau, x(\tau))d\tau \approx hf(t_k, x(t_k))$$

- The resulting (update) equation is:

$$\boxed{x(t_{k+1}) \approx x(t_k) + hf(t_k, x(t_k))}$$

(This is the simplest *one-step explicit* method)

Euler's Method: Update Equation

- The forward "update equation" for Euler's method can be written as:

$$x[k+1] = x[k] + h\,f(t[k], x[k]),\ \ k = 0, 1, \ldots$$

where:

$h_k = t_{k+1} - t_k$

$$x(t_k) = x[k] = x_k, \quad t_k = t[k] = t_k \quad (= kh)$$

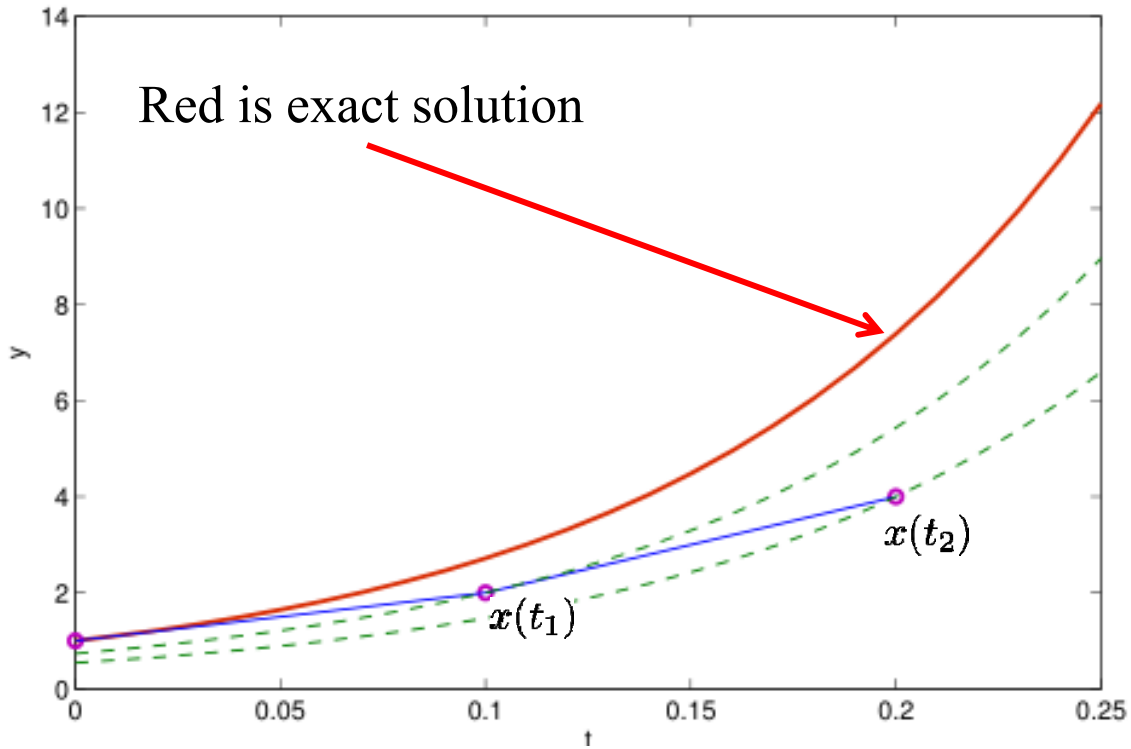
So: $$\boxed{x_{k+1} = x_k + h\,f(t_k, x_k),\ \ k = 0, 1, \ldots}$$

Note:

For "good results" Euler's method requires a **very *"small" h***. The "appropriate" value of *h* depends on the differential equation (keep in mind that if ***h*** is too small it will cause round-off problems)

Euler's Method in Pictures

- Two steps of Euler with $h=0.1$



- Dashed lines are "other trajectories" of the ODE passing through the approximate solutions obtained at each step.
- The ***broken line (blue) is the approximate solution*** found using Euler's method. Each (blue) segment is tangent to the trajectories at the beginning of each step

How Accurate is the Numerical Solution ?

- When solving differential equations numerically there are ***three sources of error:***
  - **Rounding Error:**
    - *Due to floating point* representation. May be large if the steps *h* are too small.
  - **Local Error:** (easy to estimate, see next)
    - Introduced by assuming that $x(t_k) = x[k]$ is the true solution (recall figure).
  - **Global Error:**
    - Represents how far we have strayed from the true solution (assuming no rounding error.)

Euler's Method: **Local Error Analysis**

- From (Taylor series expansion)

$$x(t_{k+1}) = x(t_k) + h \underbrace{f(t_k, x(t_k))}_{x^{(1)}(t_k)} + \frac{h^2}{2!} x^{(2)}(\xi)$$

the ***local error*** is

$$f(t_k, x(t_k)) - \frac{x(t_{k+1}) - x(t_k)}{h} = -\frac{h}{2} x^{(2)}(\xi)$$

- Therefore the *local error* is $O(h)$

  We say that *Euler's methods is order 1*

Note: If the error is $O(h^q)$ the ***method is of order q***

Euler's Method: Summary

- Euler's method belongs to the class of ***one-step methods*** (it uses only the current estimate $x(t_k)$ to compute the next estimate $x(t_{k+1})$).

- It is ***not practical*** since it requires a very small (integration) step h, (because the local error is $O(h)$)

- Euler's method is important for its simplicity and its use in the ***derivation of more accurate algorithms.***

Improving Accuracy

- How can we improve the accuracy of the numerical solution of differential equations ?

- There are ***two main approaches:***
  1. ***Reduce*** the ***step size, h***
  2. ***Increase the order of the algorithm*** (*e.g.*, using better integration algorithms, etc.)

- We will explore the algorithm order.

Heun's Method

- A 2nd order algorithm called *Heun's method* can be obtained using *trapezoidal integration*

$$x(t_{k+1}) = x(t_k) + \frac{h}{2}(f(t_k, x(t_k)) + f(t_{k+1}, x(t_{k+1})))$$

"implicit method"

- Then it is converted to an explicit method by estimating $x(t_{k+1})$ using Euler's algorithm

$$x(t_{k+1}) \approx x(t_k) + hf(t_k, x(t_k))$$

Heun's Method

- The result is

Now explicit

$$x(t_{k+1}) = x(t_k) + \frac{h}{2}\{f(t_k, x(t_k)) + f(t_{k+1}, x(t_k) + hf(t_k, x(t_k)))\}$$

- Note that the correction (2nd term) is the ***average between***
  - The ***prediction*** based on the ***slope at*** t_k

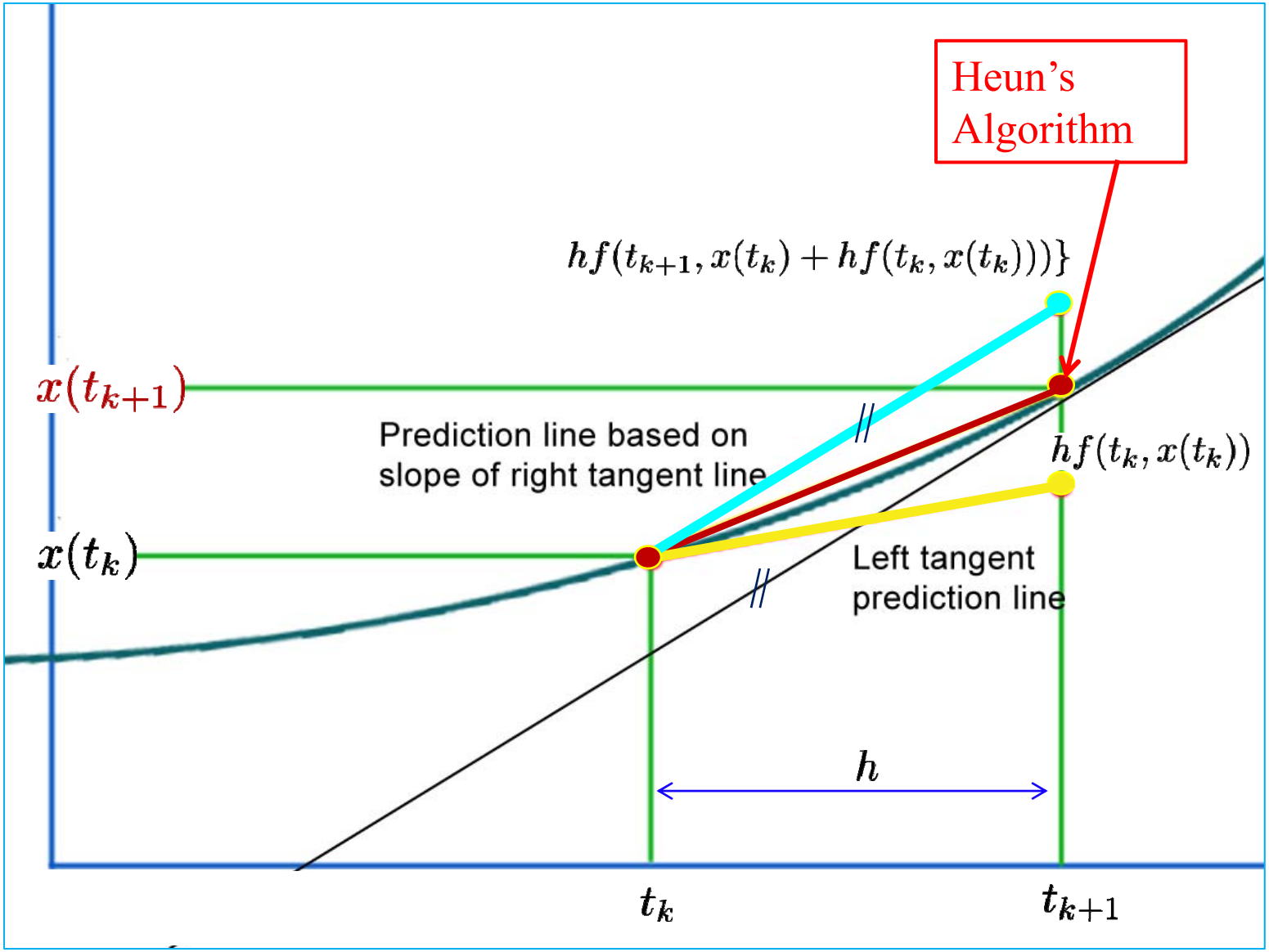
    $$hf(t_k, x(t_k))$$

  - The ***prediction*** based on the ***slope at*** t_{k+1}

    $$hf(t_{k+1}, x(t_k) + hf(t_k, x(t_k)))\}$$

(see next slide for an illustration)

Heun's Method in Pictures



Heun's Method

$$x(t_{k+1}) = x(t_k) + \frac{h}{2}\{f(t_k, x(t_k)) + f(t_{k+1}, x(t_k) + hf(t_k, x(t_k)))\}$$

- A alternative way to organize this algorithm is in <u>*two stages*</u>, k_1 and k_2

$$k_1 = f(t_k, x(t_k)),$$
$$k_2 = f(t_k + h, x(t_k) + hk_1),$$
$$x(t_{k+1}) = x(t_k) + h\left(\frac{1}{2}k_1 + \frac{1}{2}k_2\right)$$

The values k_1 and k_2 are the ***slopes*** of the solution at times t_k and t_{k+1} respectively

Summary: One Step Methods

- The simplest explicit "one step" methods are:
- Euler's method (local error $\mathcal{O}(h)$) First order

$$x(t_{k+1}) = x(t_k) + hf(t_k, x(t_k))$$

- Heun's method (local error $\mathcal{O}(h^2)$) Second order

$$\begin{aligned} k_1 &= f(t_k, x(t_k)), \\ k_2 &= f(t_k + h, x(t_k) + hk_1), \\ x(t_{k+1}) &= x(t_k) + h\left(\frac{1}{2}k_1 + \frac{1}{2}k_2\right) \end{aligned}$$

Reference: One Step Methods

- The general form of a "one step" algorithm is

$$\frac{1}{h_i}(x(t_{i+1}) - x(t_i)) = \phi(f, t_i, x(t_i), x(t_{i+1}), h_i)$$

- In *one step* methods *the* ***computation of*** $x(t_{i+1})$ ***requires knowing*** $x(t_i)$, the previous value.
  - If it depends on more past values the method is called multistep.

Notes:

- If ϕ *depends on* $x(t_{i+1})$ then the method is *implicit*, otherwise it is *explicit*
- *Implicit methods are computationally more expensive and harder to implement*

Numerical Solution of Differential Equations

Part III

Runge-Kutta Methods

Runge-Kutta Methods

- There are two main approaches to improve the accuracy differential equations solvers
  1. Reduce the step size, *h*
  2. Increase the order of the algorithm (e.g., using better integration algorithms, etc.)

- Here we will explore a family of higher order algorithms known collectively as Runge-Kutta (RK)

Explicit Runge-Kutta Methods

- *Runge-Kutta* (RK) methods use combinations of "Euler-like" *stages* to *reduce the local error without using higher order derivatives.*

- The general form of explicit *RK algorithms* is:

$$x_{k+1} = x_k + h\,\varphi(t_k, x_k, h)$$

where $\varphi(\cdot)$ is called the ***increment function***

- The increment function for an *s-stage RK method* is of the form

$$\varphi(\cdot) = b_1 k_1(\cdot) + b_2 k_2(\cdot) + \cdots + b_s k_s(\cdot)$$

Note that $\varphi(\cdot)$ is just a **weighted sum** of the stage values

Euler's Method is **RK1**

- Euler's method can be considered a *one-stage RK* method. It can be written as

$$k_1 = f(t_k, x_k),$$
$$x_{k+1} = x_k + h k_1$$

- Euler's method coincides with the Runge-Kutta 1 algorithm (there is only one RK1)
- Its ***local error*** is $\mathcal{O}(h)$

Heun's Method is an **RK2**

- Heun's method can be regarded as a *two-stage RK* method (k_1 and k_2 are the stages)

$$
\begin{aligned}
k_1 &= f(t_k, x_k), \\
k_2 &= f(t_k + h, x_k + h k_1), \\
x_{k+1} &= x_k + h\left(\tfrac{1}{2}k_1 + \tfrac{1}{2}k_2\right)
\end{aligned}
$$

- It is an *RK2* method (there are other possible <u>RK2</u> methods,… more later)
- The ***local error*** is $\mathcal{O}(h^2)$

RK – General form

$$k_1 = f(t_k, x_k),$$
$$x_{k+1} = x_k + h k_1$$

RK1

$$k_1 = f(t_k, x_k),$$
$$k_2 = f(t_k + h, x_k + h k_1),$$
$$x_{k+1} = x_k + h\left(\tfrac{1}{2}k_1 + \tfrac{1}{2}k_2\right)$$

RK2

$$x_{k+1} = x_k + h \sum_{i=1}^{s} b_i\, k_i$$

RK-s

$$k_1 = f(t_k, x_k)$$
$$k_2 = f(t_k + c_2 h, x_k + h\, a_{21} k_1)$$
$$\vdots$$
$$k_s = f(t_k + c_s h, x_k + h \sum_{j=1}^{s-1} a_{s,j} k_j)$$

Where: $\mathbf{b_i}$ **&** $\mathbf{c_s}$
- some weighting factor

General Explicit RK algorithms

- It is possible to construct ***RK algorithms of any desired order***
- For orders greater than one the construction is ***non-unique*** so there are many possible *RK2 (* $\mathcal{O}(h^2)$*)*, *RK3 (* $\mathcal{O}(h^3)$*)*, *RK4 (* $\mathcal{O}(h^4)$*)*, etc.
- An ***RKn*** *method requires* ***n-stages***
- The ***efficiency*** of *RK* algorithms depends on the ***number of functional evaluations*** required ***at each step***.

Explicit ***s-stage RK*** algorithms

$$x_{k+1} = x_k + h \sum_{i=1}^{s} b_i \, k_i$$

$$k_1 = f(t_k, x_k)$$

$$k_2 = f(t_k + c_2 h, x_k + h \, a_{21} k_1)$$

$$\vdots$$

$$k_s = f(t_k + c_s h, x_k + h \sum_{j=1}^{s-1} a_{s,j} k_j)$$

Tableau

$\boldsymbol{c}$	A
	$\boldsymbol{b}^T$

$$c_i = \sum_{j=1}^{s} a_{i,j}, \quad i = 1, \ldots, s$$

The values of ***c*** are determined by the rows of ***A***

Explicit s-stage RK Algorithms

- *All RK algorithms* can be represented in a "tableau" (i.e., a table)

$$\begin{array}{c|c} \boldsymbol{c} & A \\ \hline & \boldsymbol{b}^T \end{array}$$

- For *explicit RK* methods $\boldsymbol{A}$ is ***strictly lower triangular***

$$\begin{array}{c|ccccc} c_1 \rightarrow 0 & 0 & & & & \\ c_2 & a_{21} & 0 & & & \\ c_3 & a_{31} & a_{32} & 0 & & \\ \vdots & \vdots & & & \ddots & \\ c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & 0 \\ \hline & b_1 & b_2 & \cdots & c_{s-1} & c_s \end{array}$$

Note:
Origin 1

Common RK2 Algorithms

$$\begin{array}{c|c} \boldsymbol{c} & A \\ \hline & \boldsymbol{b}^T \end{array}$$

- The tables below show several explicit 2nd order RK methods (note that $c_1 = 0$)

$$\begin{array}{c|cc} 0 & & \\ \frac{1}{2} & \frac{1}{2} & \\ \hline & 0 & 1 \end{array} \qquad \begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} \qquad \begin{array}{c|cc} 0 & & \\ \frac{2}{3} & \frac{2}{3} & \\ \hline & \frac{1}{4} & \frac{3}{4} \end{array}$$

Note: "c" is the sum of the rows of "A", "b" has to sum to 1
Tables are **origin 1** not 0. The Index of the "A" matrix starts at the value shown, not in the blank space.

2-stage RK - Heun's algorithm

$s = 2$

Given: $\begin{array}{c|c} c & A \\ \hline & b^T \end{array} = \begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$

$k_1 = f(t_k, x_k)$

$$\boxed{k_s = f(t_k + c_s h, x_k + h \sum_{j=1}^{s-1} a_{s,j} k_j)}$$

From table:

$a_{1,1} = 1$, $c_2 = 1$

$b_1 = ½$; $b_2 = ½$

$k_2 = f(t_k+c_2h, x_k+h(a_{1,1}k_1))$

$k_2 = f(t_k+ 1h, x_k+h(1*k_1))$

$k_2 = f(t_k+ h, x_k+hk_1)$

Note:
Origin 1

$$x_{k+1} = x_k + h \sum_{i=1}^{s} b_i k_i$$

$x_{k+1} = x_k+h(½k_1+½k_2)$

Other 2-stage RK algorithms

s = 2

Given: $\begin{array}{c|c} \boldsymbol{c} & A \\ \hline & \boldsymbol{b}^T \end{array} = \begin{array}{c|cc} 0 & & \\ \frac{1}{2} & \frac{1}{2} & \\ \hline & 0 & 1 \end{array}$

$k_1 = f(t_k, x_k)$

$$\boxed{k_s = f(t_k + c_s h, x_k + h \sum_{j=1}^{s-1} a_{s,j} k_j)}$$

From table:

$a_{1,1} = 1/2$, $c_2 = ½$

$b_1 = 0$; $b_2 = 1$

$k_2 = f(t_k+c_2h, x_k+h(a_{1,1}k_1))$

$k_2 = f(t_k+ ½h, x_k+h(½*k_1))$

$k_2 = f(t_k+ ½h, x_k+h½k_1)$

$$x_{k+1} = x_k + h \sum_{i=1}^{s} b_i k_i$$

$x_{k+1} = x_k+h(0k_1+1k_2) = x_k+hk_2$

Common RK3 Algorithms

$$\begin{array}{c|c} \boldsymbol{c} & A \\ \hline & \boldsymbol{b}^T \end{array}$$

- The tableaux for some explicit 3rd order RK methods are (note that $c_1 = 0$)

$$\begin{array}{c|ccc} 0 & & & \\ \frac{2}{3} & \frac{2}{3} & & \\ \frac{2}{3} & 0 & \frac{2}{3} & \\ \hline & \frac{1}{4} & \frac{3}{8} & \frac{3}{8} \end{array}$$

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

$$\begin{array}{c|ccc} 0 & & & \\ \frac{1}{3} & \frac{1}{3} & & \\ 1 & -1 & 2 & \\ \hline & 0 & \frac{3}{4} & \frac{1}{4} \end{array}$$

$$\begin{array}{c|ccc} 0 & & & \\ 1 & 1 & & \\ \frac{1}{2} & \frac{1}{4} & \frac{1}{4} & \\ \hline & \frac{1}{6} & \frac{1}{6} & \frac{2}{3} \end{array}$$

$$\begin{array}{c|ccc} 0 & & & \\ \frac{2}{3} & \frac{2}{3} & & \\ \frac{2}{3} & \frac{1}{3} & \frac{1}{3} & \\ \hline & \frac{1}{4} & 0 & \frac{3}{4} \end{array}$$

$$\begin{array}{c|ccc} 0 & & & \\ \frac{2}{3} & \frac{2}{3} & & \\ 0 & -1 & 1 & \\ \hline & 0 & \frac{3}{4} & \frac{1}{4} \end{array}$$

Classic RK3

s = 3

Given: $\begin{array}{c|c} c & A \\ \hline & b^T \end{array} = \begin{array}{c|ccc} 0 & & & \\ \frac{2}{3} & \frac{2}{3} & & \\ \frac{2}{3} & 0 & \frac{2}{3} & \\ \hline & \frac{1}{4} & \frac{3}{8} & \frac{3}{8} \end{array}$

$k_1 = f(t_k, x_k)$

$$k_s = f(t_k + c_s h, x_k + h \sum_{j=1}^{s-1} a_{s,j} k_j)$$

From table:

$a_{1,1}=2/3$, $a_{2,2}=2/3$ $c_2=c_3=2/3$

$b_1= ¼$; $b_2=b_3 = 3/8$;

$k_2 = f(t_k+c_2h, x_k+h(a_{1,1}k_1)) = f(t_k+ 2/3h, x_k+h(2/3*k_1))$

$k_3 = f(t_k+c_3h, x_k+h(a_{2,2}k_2)) = f(t_k+ 2/3h, x_k+h(2/3*k_2))$

$$x_{k+1} = x_k + h \sum_{i=1}^{s} b_i k_i$$

$x_{k+1} = x_k+h(1/4k_1+3/8k_2+3/8k_3)$

Classical RK Algorithms

- The most common **"Classical RK algorithms"** of order 2, 3 and 4.

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

Classical RK2

$$\begin{array}{c|ccc} 0 & & & \\ \frac{2}{3} & \frac{2}{3} & & \\ \frac{2}{3} & 0 & \frac{2}{3} & \\ \hline & \frac{1}{4} & \frac{3}{8} & \frac{3}{8} \end{array}$$

Classical RK3

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

Classical RK4

Note: The *tableaux* of these algorithms all have *diagonal A*. This can be exploited for an ***efficient implementation***

Classic RK4

Given:

$$\begin{array}{c|c} c & A \\ \hline & b^T \end{array}$$

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

s = 4

$k_1 = f(t_k, x_k)$

$$k_s = f(t_k + c_s h, x_k + h \sum_{j=1}^{s-1} a_{s,j} k_j)$$

$a_{1,1} = a_{2,2} = 1/2$, $a_{3,3} = 1$,

$b_1 = b_4 = 1/6$; $b_2 = b_3 = 1/3$

$c_2 = c_3 = 1/2$, $c_4 = 1$,

$k_2 = f(t_k + c_2 h, x_k + h(a_{1,1} k_1)) = f(t_k + 1/2h, x_k + h(1/2 * k_1))$

$k_3 = f(t_k + c_3 h, x_k + h(a_{2,2} k_2)) = f(t_k + 1/2h, x_k + h(1/2 * k_2))$

$k_4 = f(t_k + c_4 h, x_k + h(a_{3,3} k_3)) = f(t_k + 1h, x_k + h(1 * k_3))$

$$x_{k+1} = x_k + h \sum_{i=1}^{s} b_i k_i$$

$$x_{k+1} = x_k + h(1/6k_1 + 1/3k_2 + 1/3k_3 + 1/6k_4)$$

A Fourth Order RK4 Algorithm

- The *Classical 4th order RK* method (RK4) with ***local error O(h⁴)*** is

$$
\begin{aligned}
k_1 &= f(t_k, x_k), \\
k_2 &= f(t_k + \tfrac{1}{2}h, x_k + \tfrac{1}{2}h\,k_1), \\
k_3 &= f(t_k + \tfrac{1}{2}h, x_k + \tfrac{1}{2}h\,k_2), \\
k_4 &= f(t_k + h, x_k + h\,k_3), \\
x_{k+1} &= x_k + h\left(\tfrac{1}{6}k_1 + \tfrac{1}{3}k_2 + \tfrac{1}{3}k_3 + \tfrac{1}{6}k_4\right)
\end{aligned}
$$

This method is one of the most widely used ODE solvers !

Runge-Kutta (RK) Algorithms

Advantages

- Simplicity in concept and implementation
- Flexibility in changing the step size
- Flexibility in handling discontinuities (e.g. simulation of collisions, etc.)

Disadvantages

- Number of function evaluations for high order methods is high compared to *multistep methods* of the same order

max. order achievable	2	3	4	5	6	7
func. evals per time step	2	3	4	6	7	9

Summary: Runge-Kutta Algorithms

- The **classical *RK4*** *algorithm* is the most widely used *one-step method* for solving ODEs numerically.
- For general use RK methods are a very good choice

Warning !

Convergence of these algorithms ***depends on the differential equations* being solved**, not just on the size of the increment h

ODEs with "Inputs"

- All the algorithms presented can be used without modification with ***n^{th} order differential equations*** in state-variable form *with state vector* ***x(t)*** and *input vector* ***u(t)***
- Example: Heun's method (RK2)

$$\boldsymbol{k}_1 = \boldsymbol{f}(t_k, \boldsymbol{x}_k, \boldsymbol{u}(t_k)),$$
$$\boldsymbol{k}_2 = \boldsymbol{f}(t_k + h, \boldsymbol{x}_k + h\boldsymbol{k}_1, \boldsymbol{u}(t_k + h)),$$
$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + h\tfrac{1}{2}(\boldsymbol{k}_1 + \boldsymbol{k}_2)$$

Note that ***f*** and ***u*** are evaluated at the same time instant

$\boldsymbol{k}_1, \boldsymbol{k}_2, \boldsymbol{x}$ and $\boldsymbol{u}$ are vectors;
$\boldsymbol{f}(\cdot)$ is a vector valued function

Practical C Implementations

- Simulation is defined using ***vector notation.***

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), t)$$

- C does not natively support vectors BUT does support arrays.
  - Implement the **vector** operation as a **"for"** loop across the array elements.

Simulator Design

- Simulators require **functions** (differential equations) to **simulate**
  - Want a general purpose implementation
- Use function pointers
  - Handy way to change evaluation functions in a general way.

Reminder - Function Pointers

- Functions are just addresses in memory and can also be accessed via pointers.

```
int fun1(int a);

int main(int argc, char *argv[]) {
   /* Declare a function pointer with 1 parameter that returns an int */
   int(*functionPtr)(int);
   int ans;

   functionPtr = &fun1;          /* Initialize the pointer */
    ans = (*functionPtr)(5);   /* Use the function pointer*/
       return(ans);
       } /* main() */
```

  - Notice you can pass values and return values

Homework Function Pointers

```
typedef void (*funArgs)(simParm *sim,
                double t, double *x,
                double *u, double *dx);
```

- Create a new datatype "*funArgs" that is of type function pointer
  - **It will take the 5 parameters (as shown)**
  - **It will return NOTHING**
    - Type void

Using Function Pointers

```
void eu(simParm *sim, double t0,
 double *x0, double *u, funArgs f);
```

- Use the Euler simulator with the differential equation identified by function pointer **f**
  - Type "funArgs" has been previously defined to take 5 arguments and returns nothing

Function pointer Example

- **`eu(sim, 0, x0, u, InvPend);`**
  - **`InvPend` is the name of a 5 variable differential function.**
- **`eu(sim, 0, x0, u, OtherProb);`**
  - **`OtherProb` is the name of some OTHER 5 variable differential.**

Using a Function Pointer

void eu(simParm *sim, double t0, double *x0, double *u, funArgs f)

- The Euler simulation program that is called with a pointer to the differential equation "f" to simulate.

f(sim, t0, x0, u, xp);

- Call SOME function pointed to by "f".
- It could be **InvPend OR OtherProb**
  - **Or any other 5 parameter function**

Summary: Solution of ODEs

- To solve (simulate) high order ODEs it is necessary to ***first write the equations in state-space form (system of 1st order ODEs)***

- The most common explicit one-step methods used for simulation (solving ODEs) are:
  1) (Forward) ***Euler***, ***O(h)***,
  2) ***Heun's*** *(RK2)*, ***O(h^2)***
  3) ***Classical RK4***, ***O(h^4)***

- These methods may occasionally *fail even when the step size is small (this may occur when the ODE is stiff)*

Summary: Solution of ODEs

1) Forward ***Euler***

$$k_1 = f(t_k, x_k)$$

$$x_{k+1} = x_k + hk_1$$

2) ***Heun's*** *(RK2)*

$$k_1 = f(t_k, x_k)$$

$$k_2 = f(t_k + h, x_k + hk_1)$$

$$x_{k+1} = x_k + h(\tfrac{1}{2}k_1 + \tfrac{1}{2}k_2)$$

Summary: Solution of ODEs

3) ***Classical RK3***

$$k_1 = f(t_k, x_k)$$
$$k_2 = f(t_k + {}^2/_3 h, x_k + {}^2/_3 h k_1)$$
$$k_3 = f(t_k + {}^2/_3 h, x_k + {}^2/_3 h k_2)$$
$$x_{k+1} = x_k + h({}^1/_4 k_1 + {}^3/_8 k_2 + {}^3/_8 k_3)$$

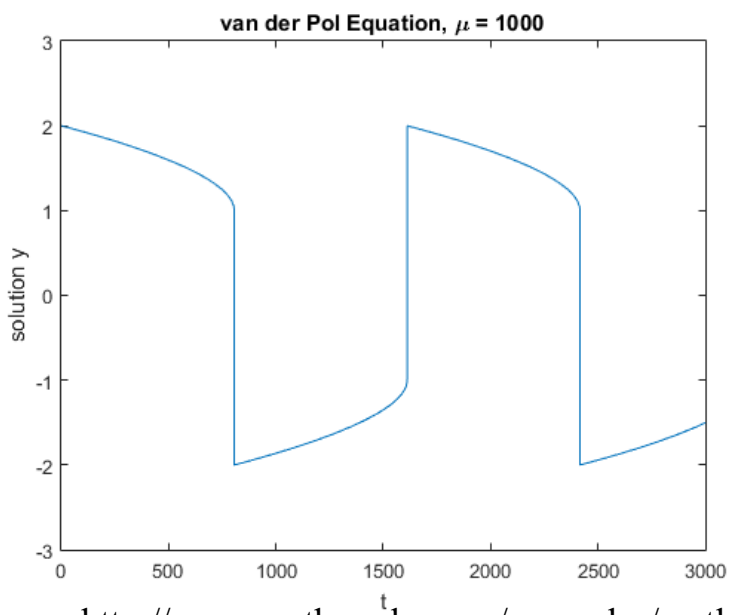
4) ***Classical RK4***

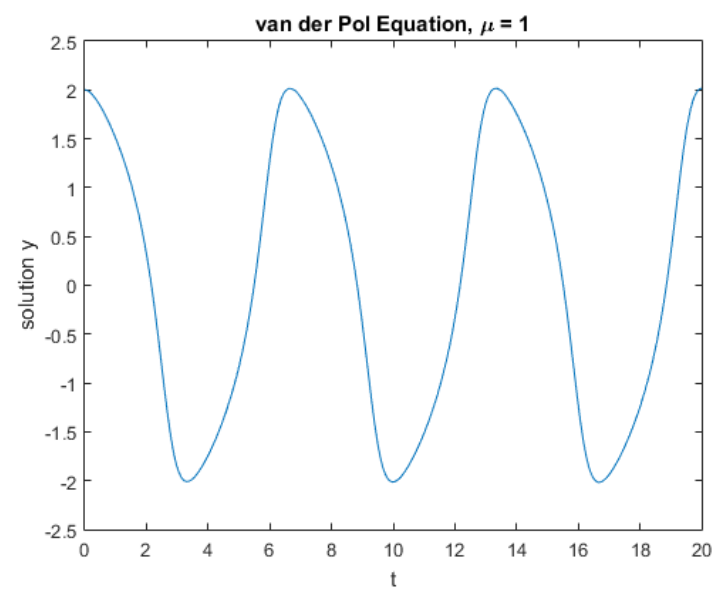
$$k_1 = f(t_k, x_k)$$
$$k_2 = f(t_k + ½h, x_k + ½hk_1)$$
$$k_3 = f(t_k + ½h, x_k + ½hk_2)$$
$$k_4 = f(t_k + h, x_k + hk_3)$$
$$x_{k+1} = x_k + h({}^1/_6 k_1 + {}^1/_3 k_2 + {}^1/_3 k_3 + {}^1/_6 k_4)$$

Simulate: van der Pol Oscillator

- A oscillator with nonlinear damping governed by the second-order differential equation:

$$y''(t) + \epsilon(y^2(t) - 1)y'(t) + y(t) = 0$$





http://www.mathworks.com/examples/matlab/1125-differential-equations

ε = strength of the damping

Work: Van der Pol Oscillator

Given:

$$y''(t) + \varepsilon(y(t)^2 - 1)y'(t) + y(t) = 0$$

Rewrite – solving for y″

$$y''(t) = \varepsilon(1 - y(t)^2)y'(t) - y(t)$$

Define state variables:

$$x_1(t) = y(t)$$

$$x_2(t) = y'(t)$$

Work: Van der Pol Oscillator

$$y''(t) = \varepsilon(1 - y(t)^2)y'(t) - y(t)$$

given: $x_1(t) = y(t)$ & $x_2(t) = y'(t)$

Rewrite using state variables:

$$y''(t) = \varepsilon(1 - x_1(t)^2)x_2(t) - x_1(t)$$

State space form:

$$\dot{x}_1(t) = x_2(t)$$

$$\dot{x}_2(t) = \epsilon(1 - x_1(t)^2)x_2(t) - x_1(t)$$

van der Pol Oscillator matlab

- vanderpol.m
  - function dydt = vanderpol(t,y,Mu)
  - % Copyright 1984-2014 The MathWorks, Inc.
  - % the update equation VECTOR

  - **dydt = [y(2); Ep*(1-y(1)^2)*y(2)-y(1)];**

Note: Matlab DOES vectors so dydt has two parts

$$\dot{x}_1(t) = x_2(t)$$

$$\dot{x}_2(t) = \epsilon(1 - x_1(t)^2)x_2(t) - x_1(t)$$

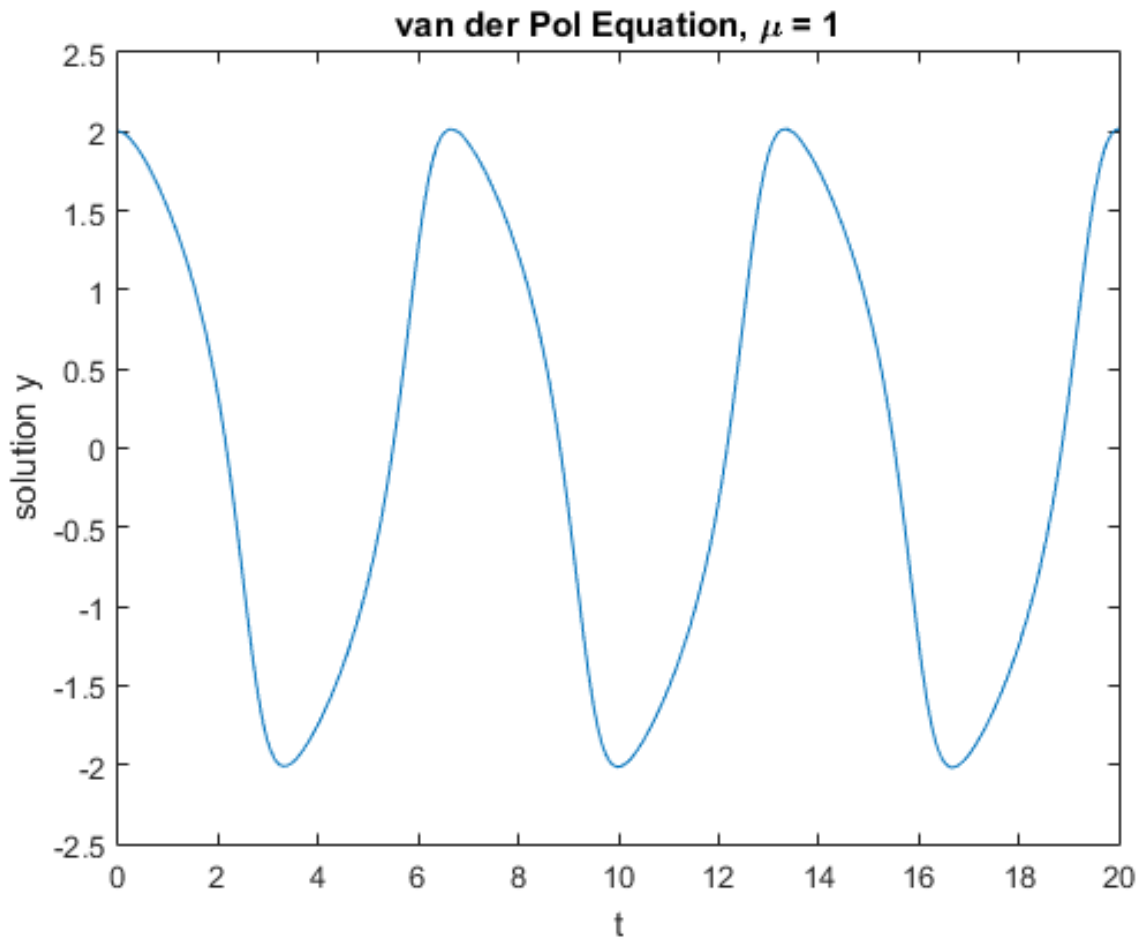
van der Pol Oscillator matlab

- Simulation using the built in ODE45 solver

- demo1.m
  - tspan = [0, 20]; %Simulation range
  - y0 = [2; 0]; %initial values
  - **Ep = 1;** %decay rate
  - ode = @(t,y) vanderpol(t,y,Ep);
  - **[t,y] = ode45(ode, tspan, y0);**
  - plot(t,y(:,1))
  - xlabel('t')
  - ylabel('solution y')
  - title('van der Pol Equation, \Ep = 1')

Result: van der Pol Oscillator

- Demo 1 Ep=1



van der Pol Oscillator matlab

- Simulation using the built in ODE15 solver

- demo2.m
  - tspan = [0, 3000];
  - y0 = [2; 0];
  - **Ep = 1000;**
  - ode = @(t,y) vanderpol(t,y,Ep);
  - [t,y] = ode15s(ode, tspan, y0);
  - plot(t,y(:,1))
  - title('van der Pol Equation, \Ep = 1000')
  - axis([0 3000 -3 3])
  - xlabel('t')
  - ylabel('solution y')

Result: van der Pol Oscillator

- Demo 2 Ep=1000