Mini-project 2

- Mini-project 2 presentations: April 21 (last day of class).
- Start thinking about what you want to do.
- Please choose a topic relevant to what we saw in the second ½ of the class during this semester.
- Suggested projects on Canvas.
- E-mail me your ideas starting now (if you haven't already).

Solving ODEs

Ordinary Differential Equations

Solving ODEs

- Problem to solve: $$\frac{dy}{dt} = f(t, y)$$

Note equation is 1st order.

- $f(t,y)$ is a given function of time, y, and some parameters.
- y can be scalar or vector (i.e. system of equations)
- We are given $y(t = 0)$

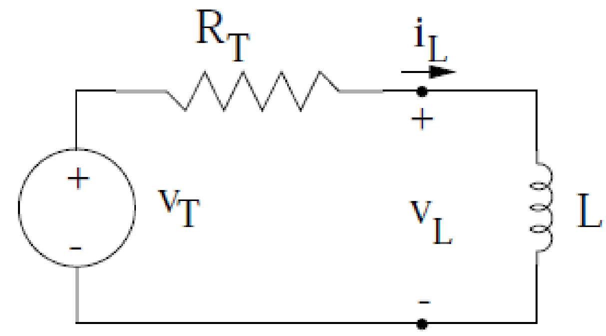
Frequently called "initial value problem"

- Goal: find $y(t)$.

Example

- Example (electronics):

$$L\frac{di_L}{dt}+R_T i_L=V_T(t)$$



- L and R are known values (parameters).

Inductor L

Resistor R

- *V*(*t*) is a known function of time (source).
- Your know *i*(*t=0*) = 0
- Goal: Determine *i*(*t*)

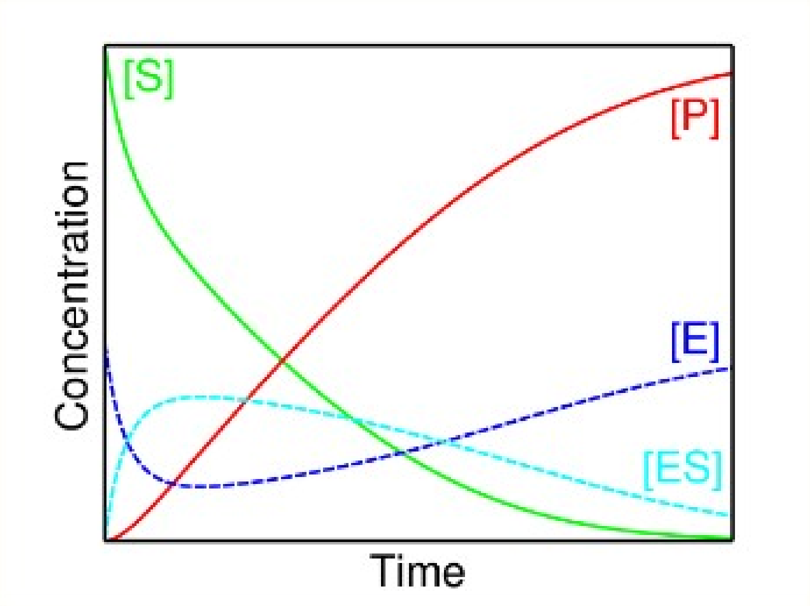
Current flow through circuit

Example – Chemical Engineering

Substituting this into Eq.(5.36) we obtain

$$\frac{d[P]}{dt} = k_{cat}[E]_{tot}\frac{[S]}{K_M + [S]} \tag{5.39}$$

where K_M is the so-called Michaelis constant defined as $K_M = \frac{k_r + k_{cat}}{k_f}$.



The changes in species concentration over time are shown in the figure. We mentioned earlier that the model is used to describe the breaking down of sucrose into glucose and fructose, while here we only have one product $[P]$. It turns out that glucose is a **competitive inhibitor** that also binds to the substrate ensuring that the reaction slows down. This reaction is not the only example of competitive inhibition

Difference between integrating a function and solving ODEs

- Integrate a function: *y* is on LHS only, *f*(*t*) can be evaluated everywhere.

$$y=\int_a^t dt_1 f(t_1) \rightarrow \frac{dy}{dt}=f(t)$$

- Integrate ODE: *y* is on both LHS and RHS. Only know function at current and past points.

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

Note ODE involves *y* on RHS

- Nonetheless, one usually says loosely "integrating an ODE".

First method: Forward Euler (1D)

- Original problem $\frac{dy}{dt} = f(t, y)$
- Taylor expansion

$$y(t_n + h) = y(t_n) + h \left. \frac{dy}{dt} \right|_{t_n} + O(h^2)$$

- A couple of definitions:

$$t_n = n \Delta t = n h \qquad y(t_n) = y_n$$

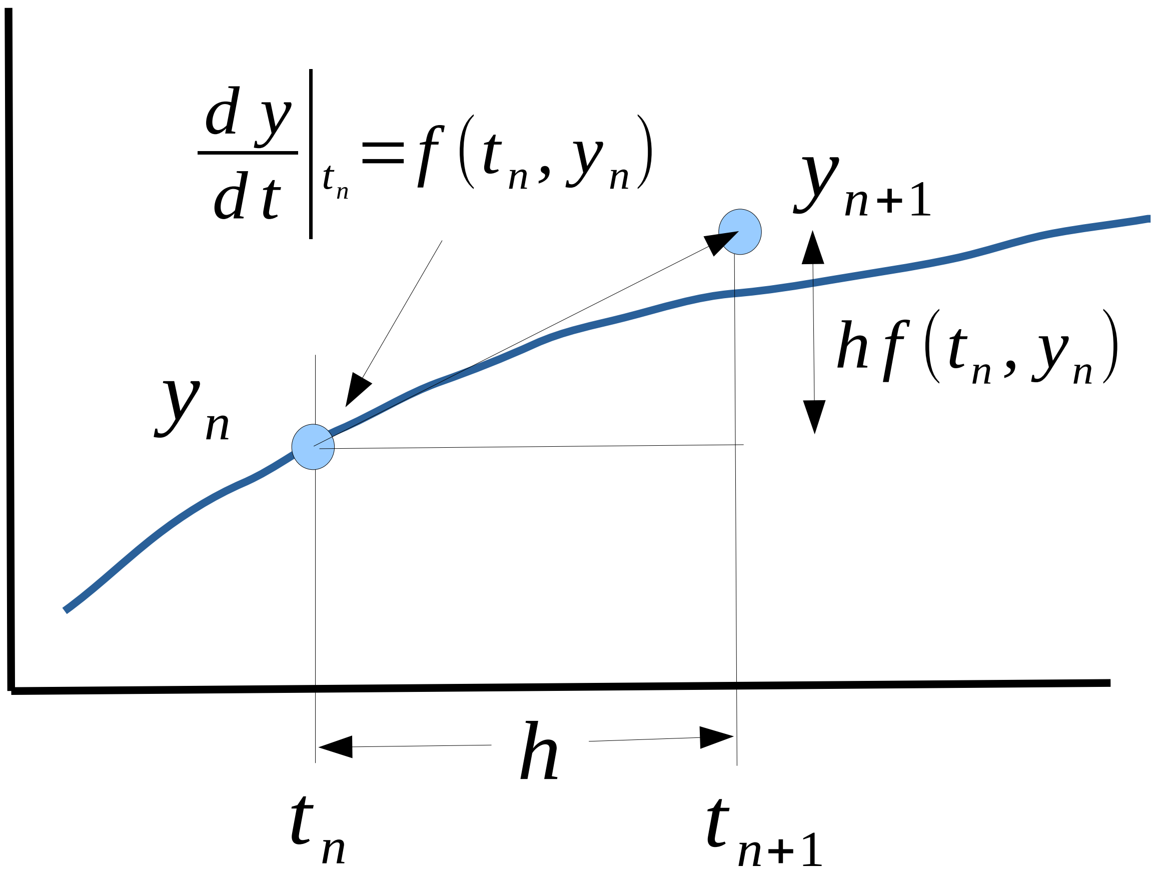
- Euler forward integration:

$$y_{n+1} = y_n + h f(t_n, y_n)$$

Value at n+1 is found
By evaluating f at time n

Forward Euler

$$y_{n+1} = y_n + h f(t_n, y_n)$$



Algorithm

1. Start at point $(t_n, y(t_n))$, stepsize h.
2. Compute $y_{n+1} = y_n + h*f(t_n, y_n)$. This is new y value at t_{n+1}
3. Is $t_{n+1} >= T_{end}$? If so, return answer to caller. Otherwise, go back to step 2 and compute next y.

```
function y = ForwardEuler(y0, N)
  % This function solves the differential equation
  % y' = f(y,t) using forward Euler integration.
  % It takes as inputs:
  % y0 = initial value of y
  % N = Number of points to compute

  global lambda;
  global alpha;
  global omega;
  global h;

  % create vector y
  y = zeros(1, N);
  t = 0;

  y(1) = y0;
  for n = 1:(N-1)
    y(n+1) = y(n) + h*f(y(n), t);
    t = t+h;
  end

end
```

This implementation returns vector y(n) representing development of y as function of, say, time.

Simple demo

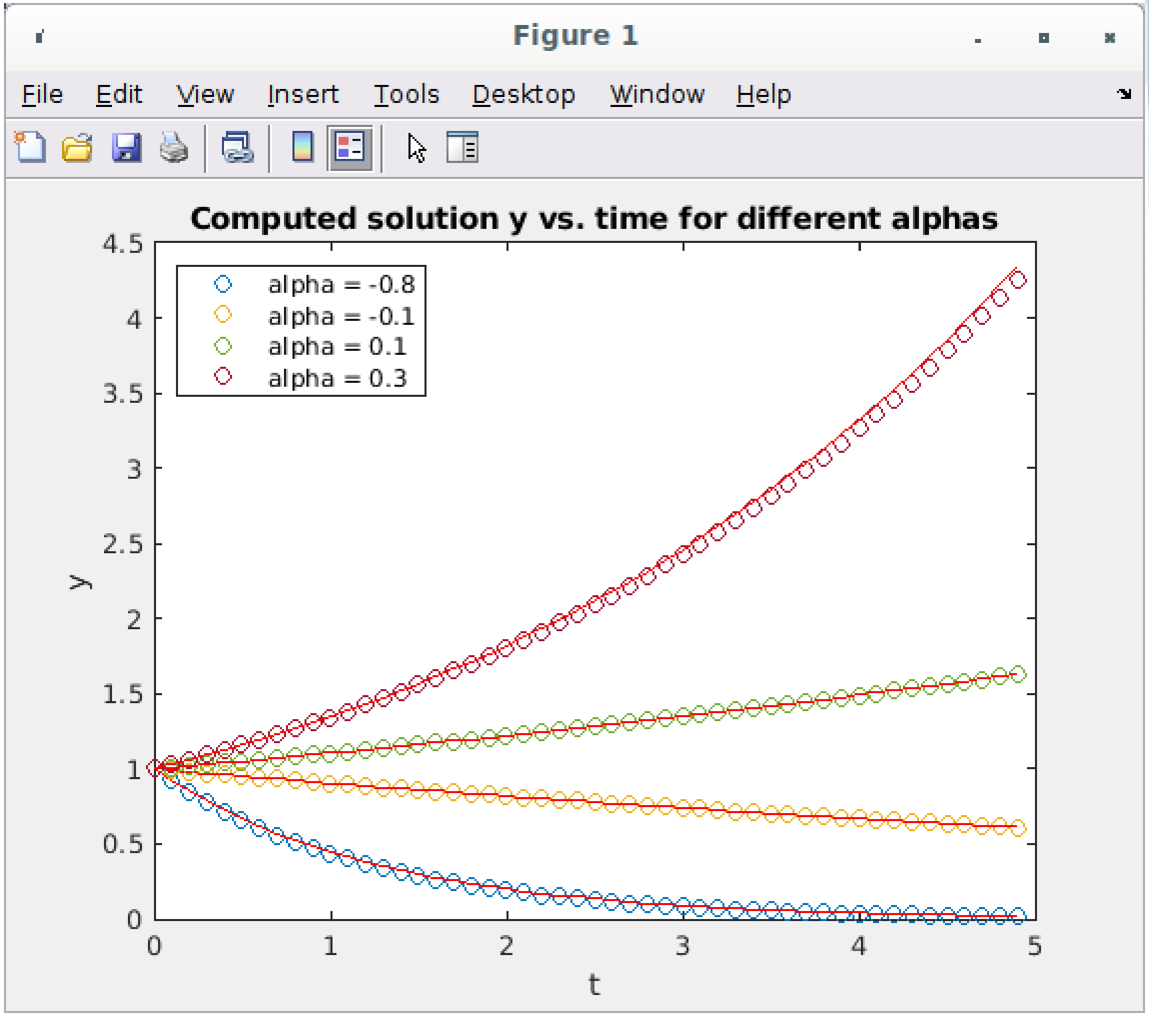
- IVP:

$$\frac{dy}{dt} = \alpha y \qquad y(0) = 1$$

- Analytic solution:

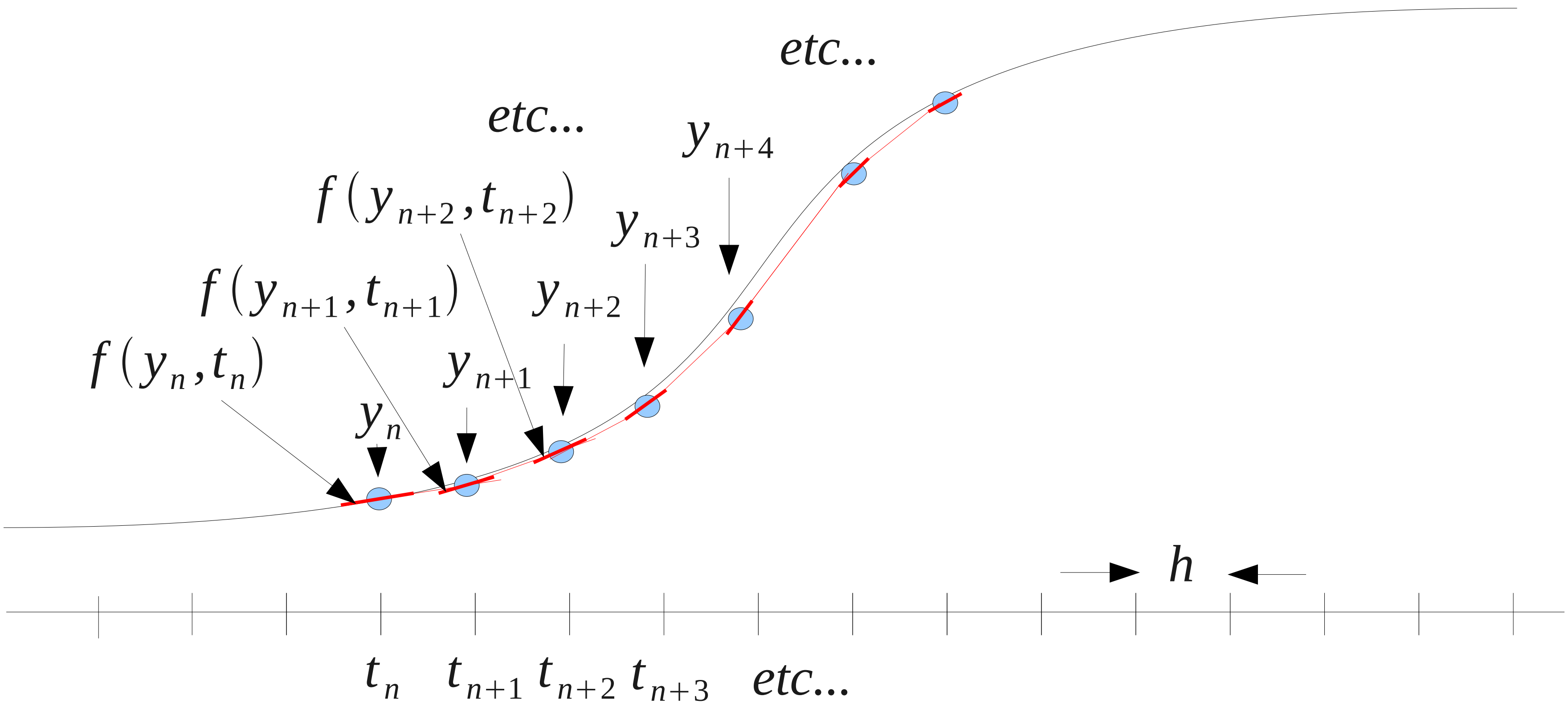
$$y(t) = e^{\alpha t}$$

- Note deviation of computed solution from analytic result.



Forward Euler

- Evaluate slope $f(y_n, t_n)$ at each point t_n.
- Step forward based upon slope at t_n.
- Method is not that accurate....



Concept: Truncation Errors -- local and global

- Original problem (IVP)

$$\frac{dy}{dt} = f(t, y) \qquad y(0) = C$$

- Mathematically true solution:

$$y_{true} = \int_0^t f(t, y_{true})\, dt + C$$

- Forward Euler solution:

$$y_{n+1} = y_n + h f(t_n, y_n) \qquad y_0 = C$$

- Local truncation error

$$e_n = y_{true}(t_n) - y_n$$

Error observed at every step between true and computed solution

Recall derivation of forward Euler

- Original ODE

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

- Taylor expansion of y(t+h)

We ignored this term – truncation error

$$y(t_n+h)=y(t_n)+h\left.\frac{dy}{dt}\right|_{t_n}+O(h^2)$$

- Local truncation error

Every step introduces another error of O(h^2)

$$e_n=y_{true}(t_n)-y_n$$

$$e_n=y_{true}(t_n)-(y_{n-1}+hf(t_{n-1},y_{n-1})+O(h^2))$$

- Forward Euler LTE = O(h^2)

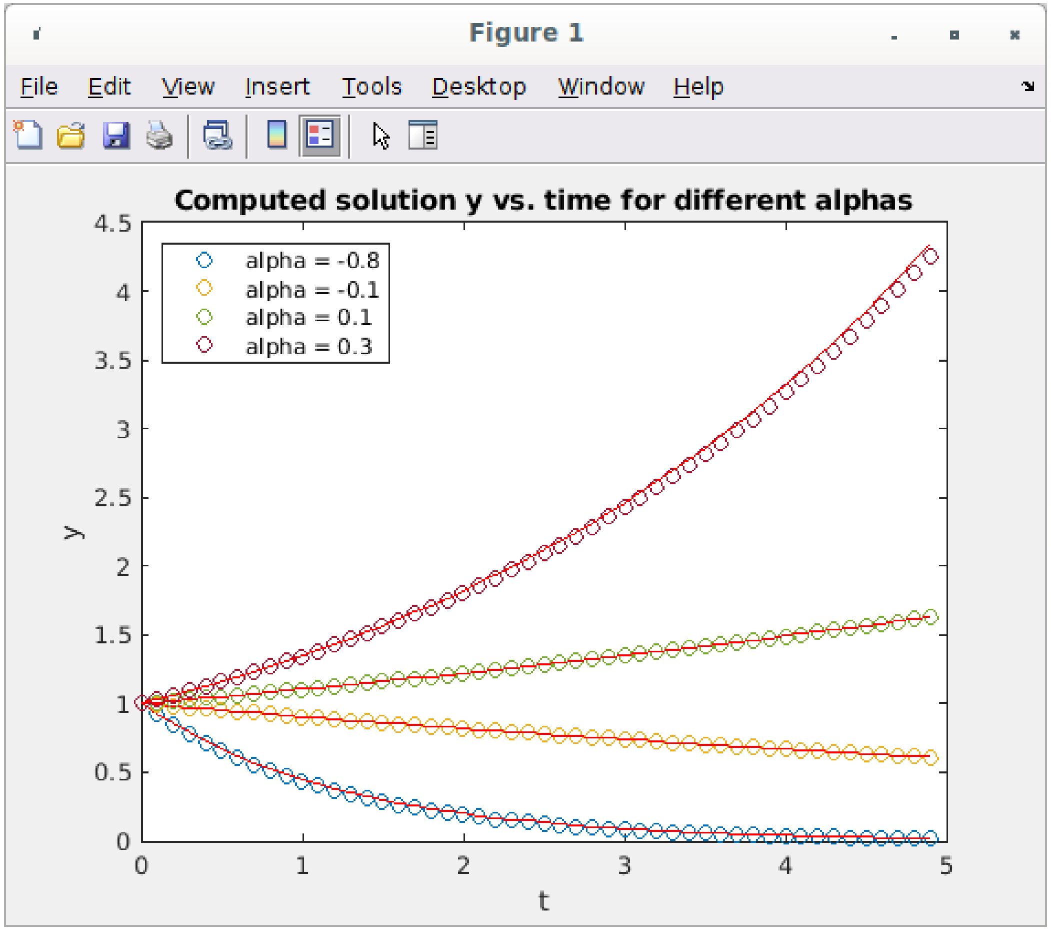
Global truncation error

- We are interested in a solution over a fixed interval length T = N*h
- We are interested in how the error over T scales as we vary h. For fixed interval T we have N= T/h.
  - Therefore, GTE

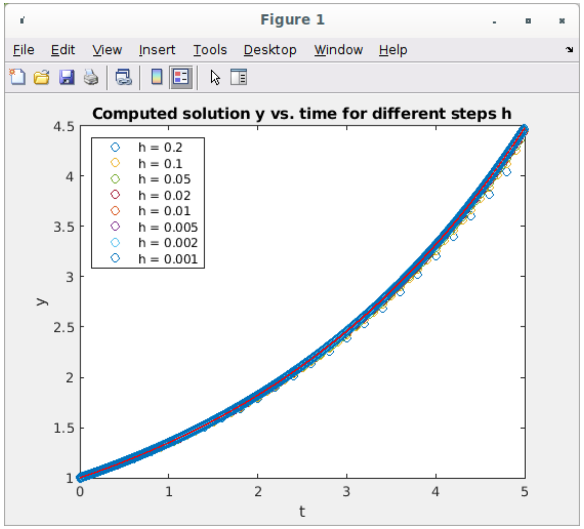
    = N*LTE

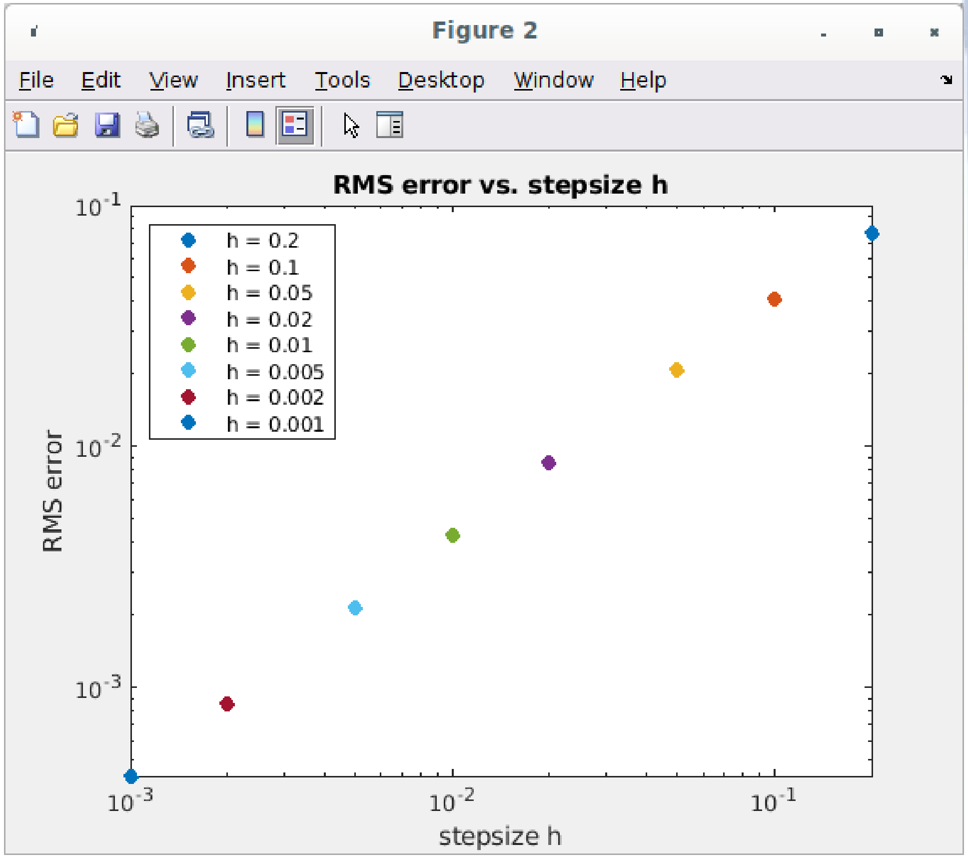
    $= O(h^2)/h$

    = O(h).



GTE for forward Euler IVP





- Compute RMS error of computed solution

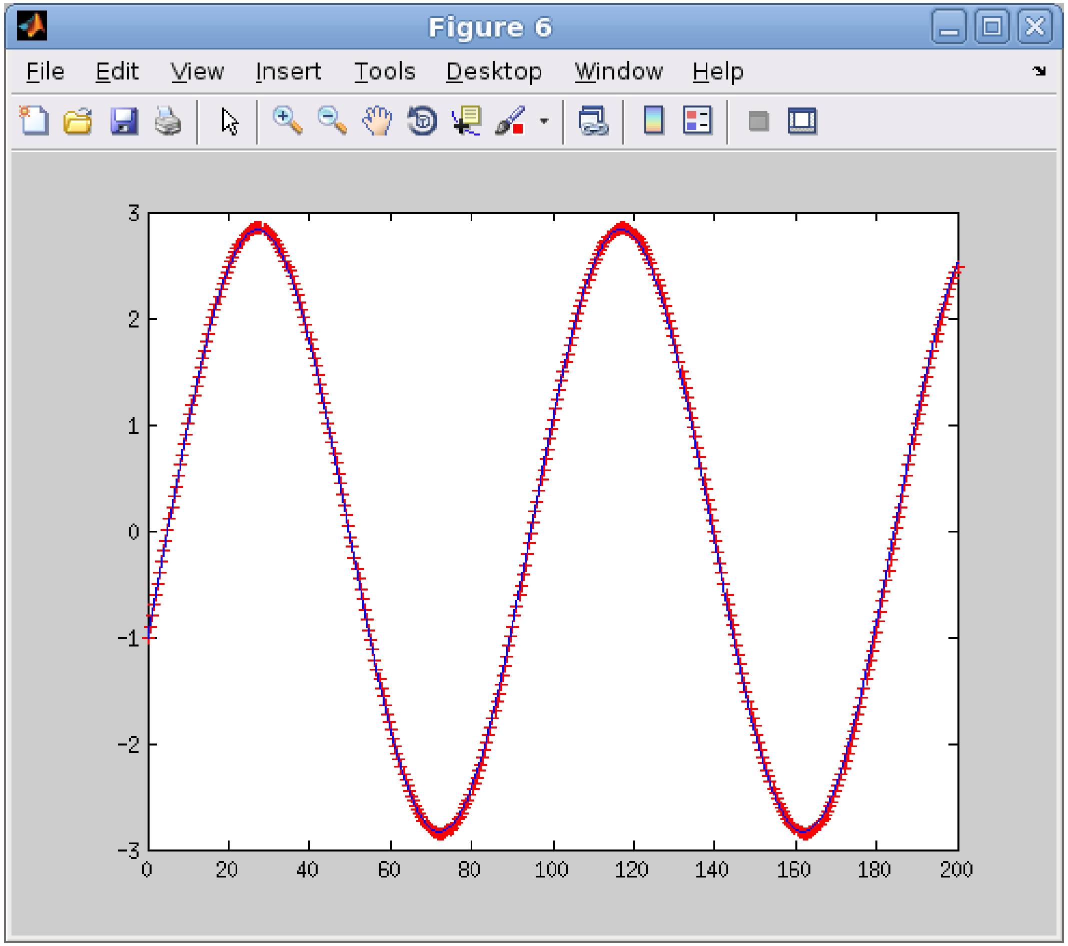
$$RMS = \sqrt{\frac{1}{N}\sum_{n=0}^{N}\left(y_{true}(t_n) - y_n\right)^2}$$

- Plot RMS error vs. stepsize h.
- Observe error scales as $RMS \sim O(h)$

Another Forward Euler solution

$$\frac{dy}{dt} = -\lambda y + \alpha \sin(\omega t)$$

- Red crosses are Forward Euler solution
- Blue line is exact solution.
- h = 0.5

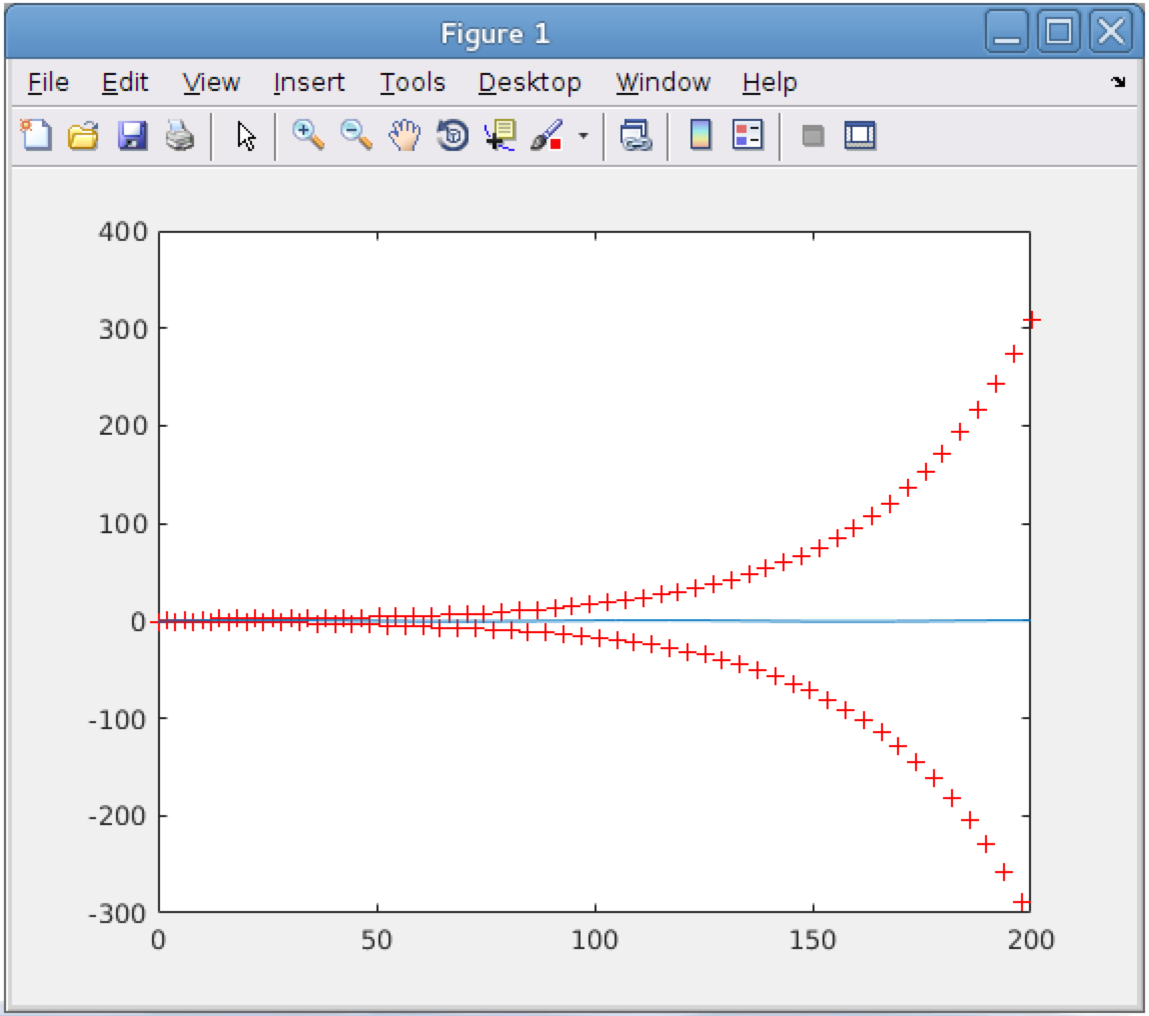


Class12/1DFirstOrderEqn

Try different step size...

$$\frac{dy}{dt} = -\lambda y + \alpha \sin(\omega t)$$

- h = 2.0
- What happened?????



Concept: Stability Region for 1st order, linear ODE

- Find behavior of forward Euler for simple linear ODE as function of input parameter.

- Consider first order system:

Error

$$\frac{dy}{dt}=\lambda y \quad \text{with decomposition} \quad y^{comp}=y^{true}+e$$

- Forward Euler discretized version

$$y_{n+1}^{comp}=y_n^{comp}+\lambda h\, y_n^{comp} \qquad \text{and} \quad e_{n+1}=e_n+\lambda h e_n$$

Forward Euler – stability region

- Stability means: Errors don't grow with *n*.

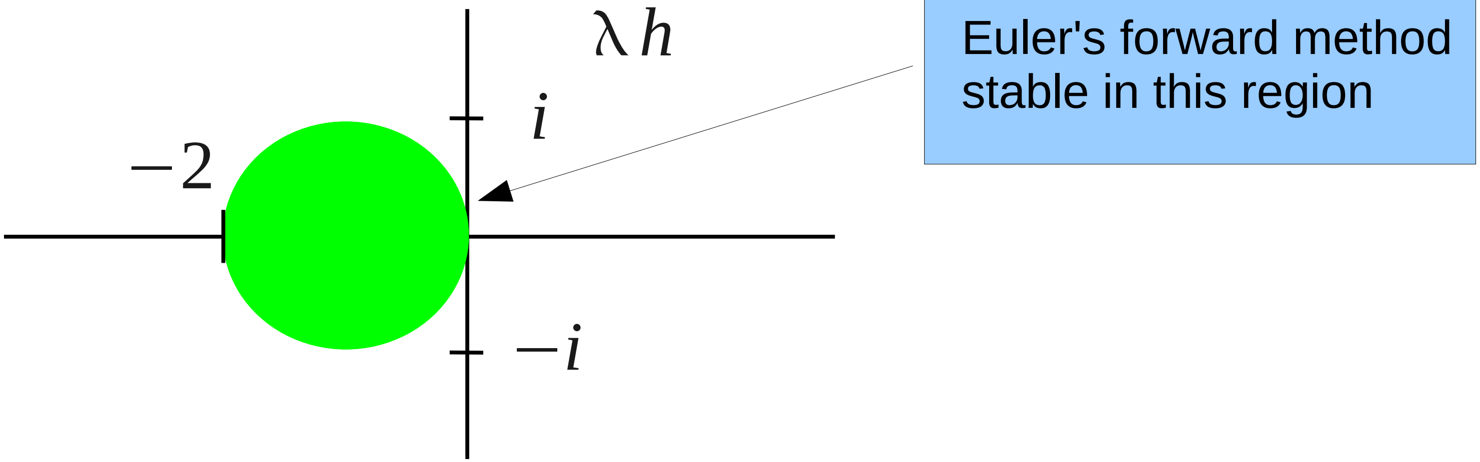
$$e_{n+1} = e_n + \lambda h e_n = (1+\lambda h) e_n$$

- Solution is

$$e_{n+1} = (1+\lambda h)^{n+1} e_0$$

- Solution (error)decays to zero when

$$|1+\lambda h| \leq 1$$



- For stability, h must decrease as λ increases.

Concept: Stability

- Stability refers to tendency of error to grow and diverge on its own
  - Error decreases over time – method is stable.
  - Error increases over time – method is unstable.
- Stability is a property of the numerical method.
- Some methods are stable, some are not.
  - Forward Euler is stable over a limited parameter range for first order linear ODE in 1D.

Next: what about 2nd order equations?

- Very important case due to e.g. Newton's 3rd law:

$$m\frac{d^2x}{dt^2}=f(x,dx/dt,t)$$

- 2nd order ODE can be written as two 1st order ODEs:
  - Define:

$$\frac{d^2y}{dt^2}+p(t,y)\frac{dy}{dt}+q(t,y)y(t)=f(t)$$

$$y_1(t)=y(t) \qquad y_2(t)=y'(t)$$

Generalization to Nth order is obvioius

$$\frac{dy_2}{dt}=-p(t)y_2(t)-q(t)y_1(t)+f(t,y(t))$$

$$\frac{dy_1}{dt}=y_2(t)$$

What about Euler's method in ND?

- Good news: Solvers work as before, except variables are vectors
- Forward Euler:

$$\frac{d\vec{y}}{dt} = \vec{f}(t, \vec{y})$$

$$\vec{y}_{n+1} = \vec{y}_n + h\vec{f}(t_n, \vec{y}_n)$$

```
% create vector y
y = zeros(1, N);
t = 0;

y(1) = y0;
for n = 1:(N-1)
  y(n+1) = y(n) + h*f(y(n), t);
  t = t+h;
end
```

1D

```
% create vector y
rows = length(y0);
y = zeros(rows, N);
t = 0;

y(:,1) = y0;
for n = 1:(N-1)
  y(:,n+1) = y(:,n) + h*f(y(:,n), t);
  t = t+h;
end
```

ND

Simple example: harmonic oscillator

- As second order ODE:

$$\frac{d^2 y}{dt^2} = -\omega^2 y$$

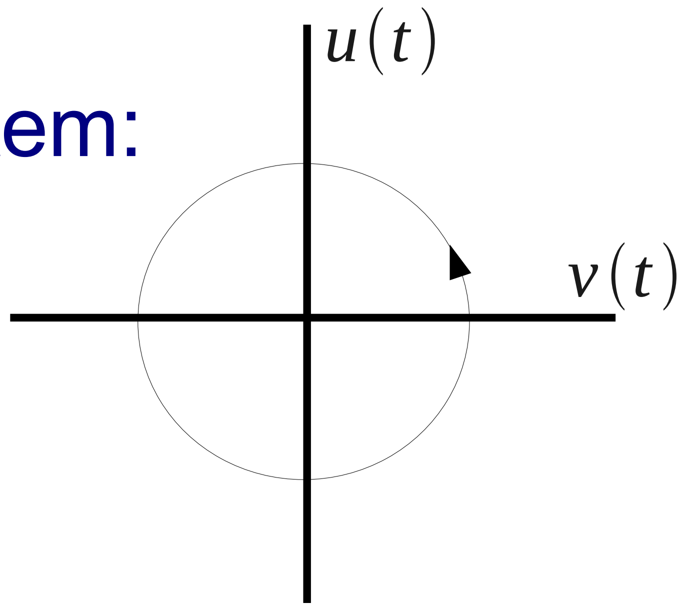
- Solutions:

$$\sin(\omega t) \qquad \cos(\omega t)$$

- Written as first order system:

$$\frac{du}{dt} = v$$

$$\frac{dv}{dt} = -\omega^2 u$$



Phase plot

Discretizing the Harmonic Oscillator

- Start with continuous ODE system:

$$\frac{du}{dt} = v \qquad \frac{dv}{dt} = -\omega^2 u$$

- Discretize (Forward Euler):

$$u_{n+1} = u_n + \Delta t\, v_n \qquad v_{n+1} = v_n - \omega^2 \Delta t\, u_n$$

- Since system is linear, write as matrix equation:

Propagator matrix

$$\begin{pmatrix} u_{n+1} \\ v_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & \Delta t \\ -\omega^2 \Delta t & 1 \end{pmatrix} \begin{pmatrix} u_n \\ v_n \end{pmatrix}$$

Forward Euler Algorithm for H.O.

1. Start at point $\begin{pmatrix} u_0 \\ v_0 \end{pmatrix}$ $\quad t=t_0$

2. Take Euler step using propagator matrix

$$\begin{pmatrix} u_1 \\ v_1 \end{pmatrix} = \begin{pmatrix} 1 & \Delta t \\ -\omega^2 \Delta t & 1 \end{pmatrix} \begin{pmatrix} u_0 \\ v_0 \end{pmatrix} \qquad t=t+\Delta t$$

3. Check if t > Tmax
   - If yes, return.
   - If no, go back to 2 and take another step

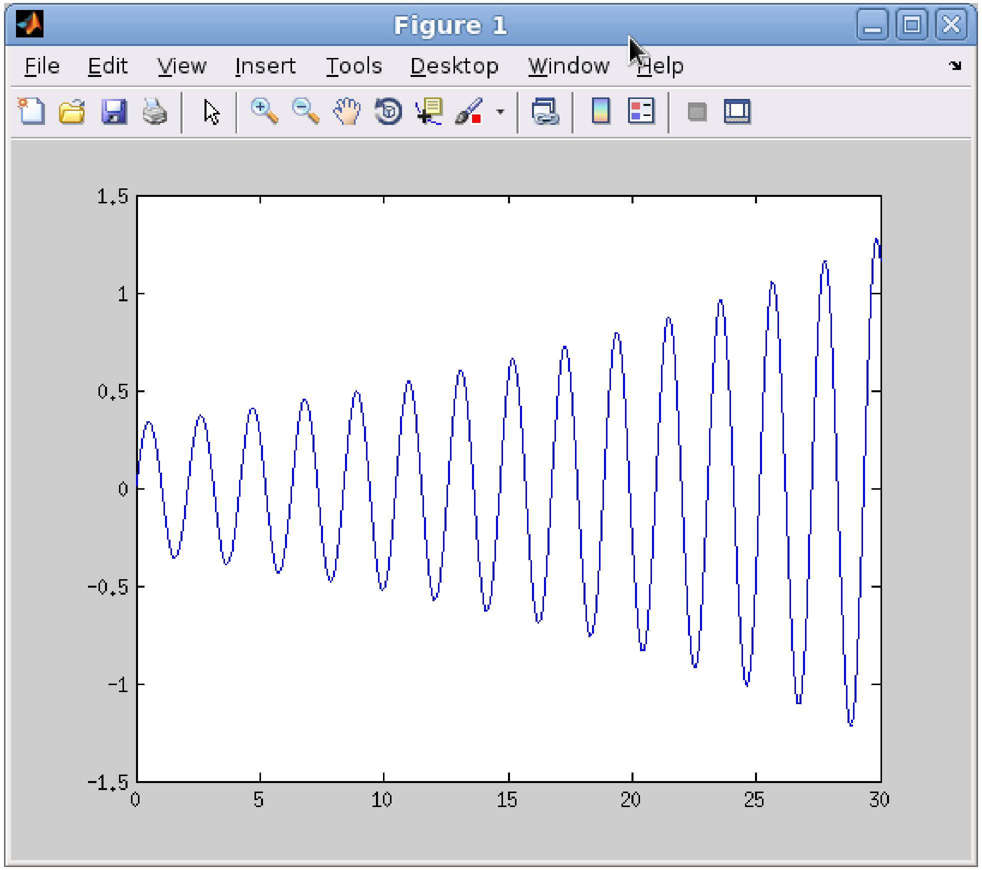
General form: $$\begin{pmatrix} u_{n+1} \\ v_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & \Delta t \\ -\omega^2 \Delta t & 1 \end{pmatrix} \begin{pmatrix} u_n \\ v_n \end{pmatrix}$$

```
function harmonic_oscillator()
  % This uses forward Euler to compute the solution
  % to the harmonic oscillator problem

  % Set up time axis of problem
  Tend = 30;
  deltat = .01;
  N = Tend/deltat;
  omega = 3;
  t = linspace(0, Tend, N);

  % Use u vector to store computed values of y(1, :)
  u = zeros(1, N);

  % Initial cond
  y = [0; 1];

  % Propagation matrix
  A = [1, deltat; -omega*omega*deltat, 1];

  for ctr = 1:N
    y = A*y;
    u(ctr) = y(1);
  end

  plot(t, u)
end
```

Harmonic oscillator – numerical solution using Forward Euler

$$\frac{du}{dt} = v$$

$$\frac{dv}{dt} = -\omega^2 u$$



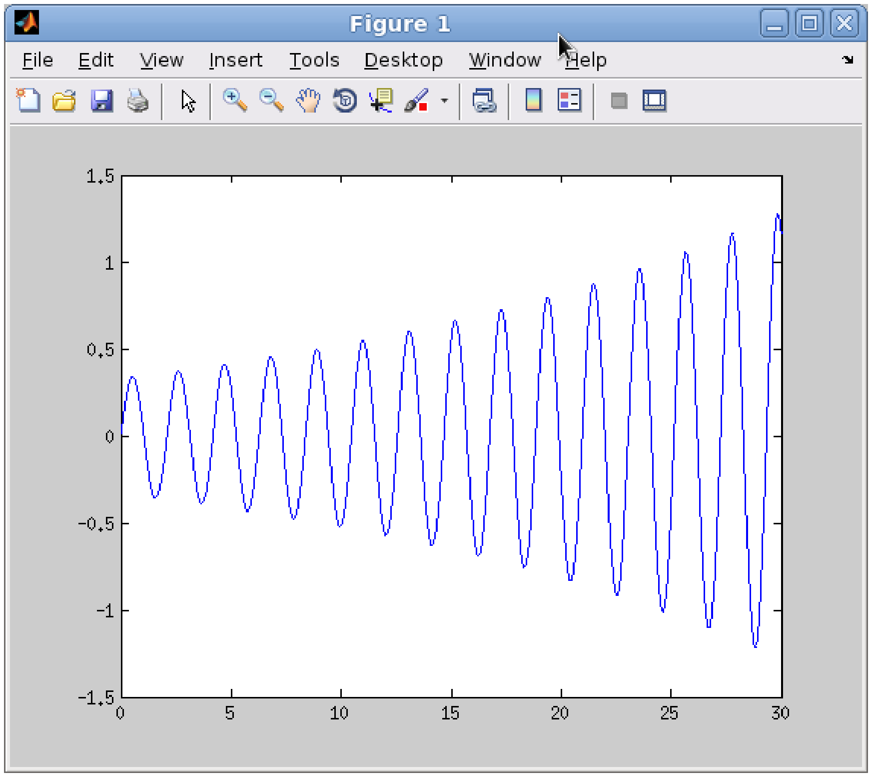
$$h = .01$$

$$\omega = 3$$

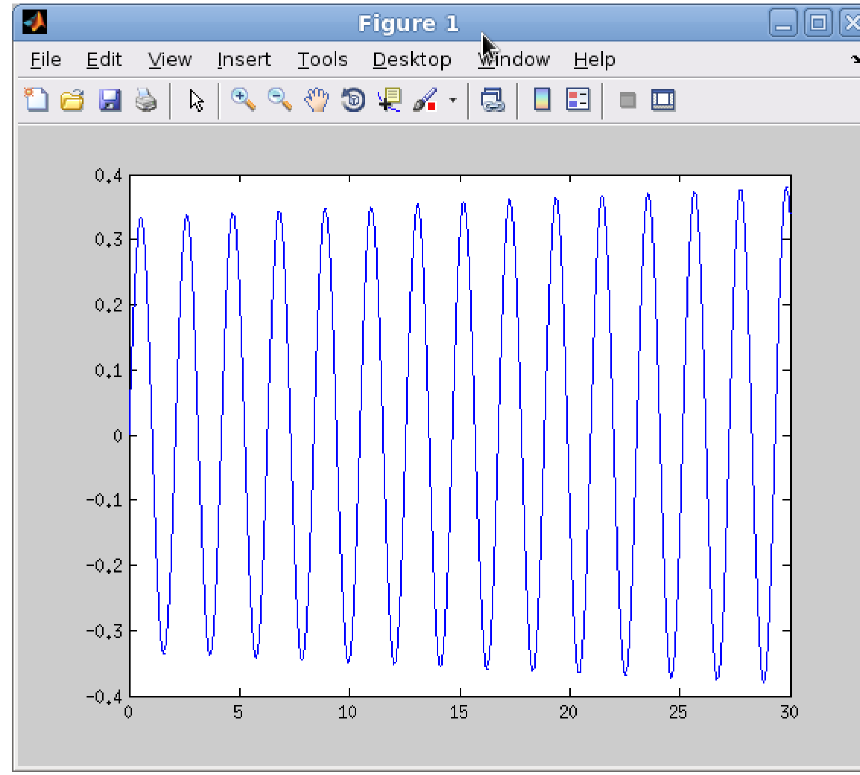
- Known general solution: $u(t) = A\sin(\omega t) + B\cos(\omega t)$
- Something is wrong.....

Solution from forward Euler

- $\omega=3$ for both runs



$\Delta t=0.01$



$\Delta t=0.001$

Stability improves, but solution always runs away.

- Stability improves for decreasing Δt
- Simulation time increases with decreasing Δt

Stability analysis for harmonic oscillator using forward Euler

- Forward Euler equations

Propagator matrix

$$u_{n+1} = u_n + \Delta t\, v_n$$

$$v_{n+1} = v_n - \omega^2 \Delta t\, u_n$$

Matrix form

$$\begin{pmatrix} u_{n+1} \\ v_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & \Delta t \\ -\omega^2 \Delta t & 1 \end{pmatrix} \begin{pmatrix} u_n \\ v_n \end{pmatrix}$$

- Assume solution

$$\begin{pmatrix} u_n \\ v_n \end{pmatrix} = \begin{pmatrix} i e_n e^{-i\omega t_n} \\ e_n e^{-i\omega t_n} \end{pmatrix}$$

Note: This is equivalent to

$$e_n \begin{pmatrix} \sin(\omega t_n) \\ \cos(\omega t_n) \end{pmatrix}$$

Vector magnitude =1 for correct solution

Stability analysis

- Substituting assumed soln into fwd Euler equation:

$$e_{n+1} e^{-i\omega t_{n+1}} \begin{pmatrix} i \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & \Delta t \\ -\omega^2 \Delta t & 1 \end{pmatrix} \cdot e_n e^{-i\omega t_n} \begin{pmatrix} i \\ 1 \end{pmatrix}$$

- Simplify

$$e_{n+1} e^{-i(\omega t_{n+1} - \omega t_{n+1})} \begin{pmatrix} i \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & \Delta t \\ -\omega^2 \Delta t & 1 \end{pmatrix} \cdot e_n \begin{pmatrix} i \\ 1 \end{pmatrix}$$

$$e_{n+1} e^{-i\Delta t} \begin{pmatrix} i \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & \Delta t \\ -\omega^2 \Delta t & 1 \end{pmatrix} \cdot e_n \begin{pmatrix} i \\ 1 \end{pmatrix}$$

- From last page,

$$e_{n+1}e^{-i\Delta t}\begin{pmatrix} i \\ 1 \end{pmatrix}=\begin{pmatrix} 1 & \Delta t \\ -\omega^2\Delta t & 1 \end{pmatrix}\cdot e_n\begin{pmatrix} i \\ 1 \end{pmatrix}$$

- Define

$$g=\left(\frac{e_{n+1}}{e_n}\right)e^{-i\omega\Delta t}$$

g is growth factor

to get:

$$g\begin{pmatrix} i \\ 1 \end{pmatrix}=\begin{pmatrix} 1 & \Delta t \\ -\omega^2\Delta t & 1 \end{pmatrix}\begin{pmatrix} i \\ 1 \end{pmatrix}$$

- This is an eigenvalue equation, *g* is eigenvalue.

Find growth factor *g*

- Eigenvalues *g* found from roots of characteristic equation:

$$det\begin{pmatrix} 1-g & \Delta t \\ -\omega^2 \Delta t & 1-g \end{pmatrix} = 0$$

$$(1-g)^2 + \omega^2 \Delta t^2 = 0$$

- Solve for *g*

$$g = 1 \pm i \omega \Delta t$$

- Recall definition of g: $g = \left(\frac{e_{n+1}}{e_n}\right) e^{-i\omega \Delta t}$

- Therefore:

$$|e_{n+1}| = |g e_n| \qquad |g| > 1$$

Implies exponentially growing error – This system is always unstable

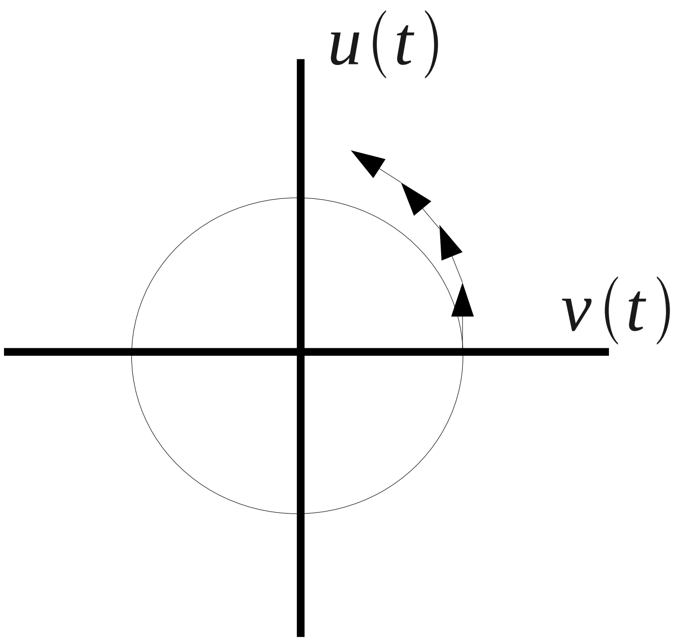
Conclusion of stability analysis

- Growth at each step:

$$|e_{n+1}| = |g\,e_n|$$

- Growth factor:

$$|g| > 1$$



- Forward Euler computed solution spirals outward from true solution.
  - Rate of divergence depends upon step size.
  - Reason: Slope used is always wrong.

Remarks

Stable for small step-sizes.

- Forward Euler is usually unstable.
- Can be stable for some parameter values (1D case).
- In general, how do you know if your simulation is stable or not?
  - Unstable simulation usually blows up. Solutions tend to infinity.
  - You should play around with h to verify results don't depend on step size.
- Use better method than Forward Euler

Next: Backward Euler method

- Start with $\frac{dy}{dt} = f(t, y)$
- Consider Taylor's expansion centered around t_{n+1}:

$$y(t_{n+1} - h) = y(t_{n+1}) - h \left.\frac{dy}{dt}\right|_{t_{n+1}} + O(h^2)$$

$$y(t_n)$$

- Rearrange to get Euler's backward method:

$$y_{n+1} = y_n + h \left.\frac{dy}{dt}\right|_{t_{n+1}} + O(h^2)$$

$$= y_n + hf(t_{n+1}, y_{n+1})$$

Value at n+1 is found
By evaluating f at time n+1

Backward Euler...

- Concept: forward Euler = explicit method, backward Euler = implicit method.
- You need *f(y+h)* to compute *f(y+h)*. How to get *f(y+h)*??

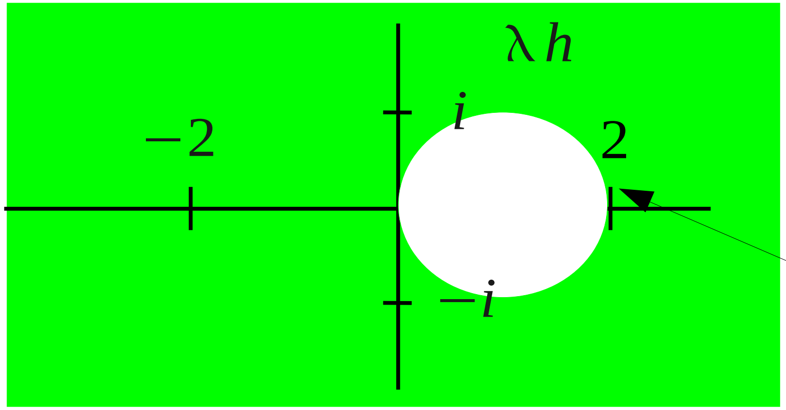
$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1}) \rightarrow$$

Find u such that $g(u) = y_n + hf(t_{n+1}, u) - u = 0$

  - Iteration sometimes works.
  - Newton's method works, but you need analytic derivative.
  - Secant method works.
- GTE = O(h)

Consider stability of backward Euler

- Recall linear equation: $\frac{dy}{dt} = \lambda y$
- Recall decomposition into into true + error terms. But now $e_{n+1} = e_n + \lambda h e_{n+1}$
- This implies error grows as $e_n = \left(\frac{1}{1 - \lambda h} \right)^n e_0$
- Stable for $\left| \frac{1}{1 - \lambda h} \right| \leq 1$

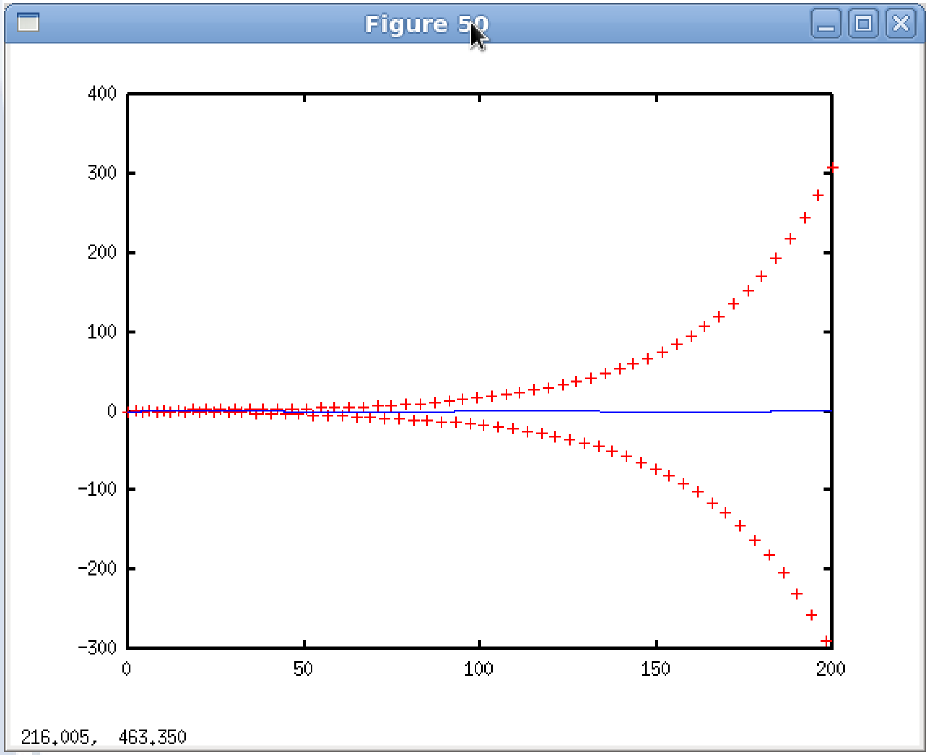


Euler's backward method stable in this region

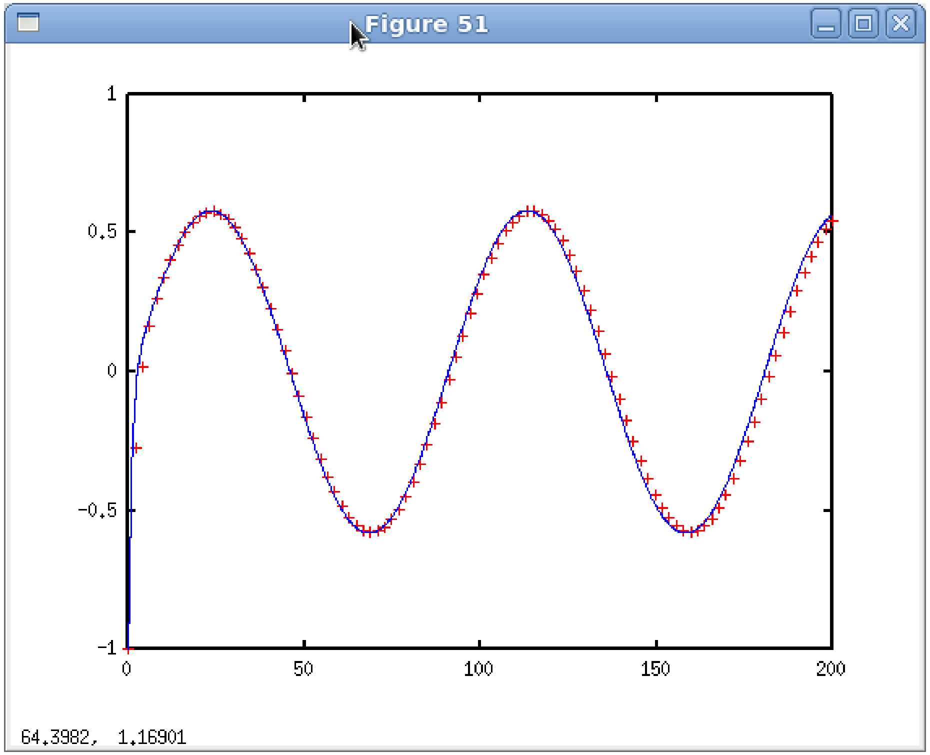
Forward vs. Backward Euler

$$\frac{dy}{dt} = -\lambda\, y + \alpha \sin(\omega t)$$

$$\lambda = 1.03 \qquad \alpha = 0.6 \qquad \omega = 0.07 \qquad h = 2$$



Forward Euler => unstable for these parameters



Backward Euler => stable for these parameters

Improved Euler method (Heun's method)

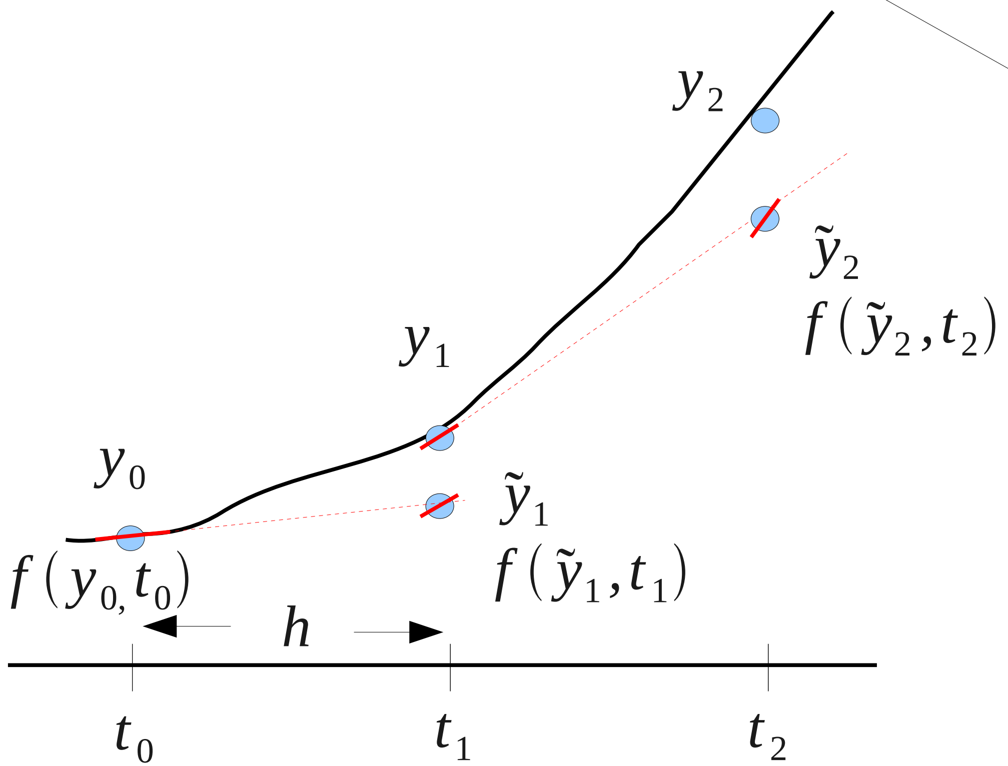
- Compute solution using

$$y_{n+1} = y_n + \frac{h}{2}\left(f(t_n, y_n) + f(t_{n+1}, \tilde{y}_{n+1})\right)$$

Use average slope Between t_n and t_{n+1}

where $\tilde{y}_{n+1} = y_n + h f(t_n, y_n)$

Estimate y_{n+1} using forward Euler



- Explicit method (“forward” method)
- First step equivalent to trapezoidal method for integration.

Heun's method algorithm

$$\frac{d\vec{y}}{dt} = \vec{f}(t, \vec{y})$$

Equation to integrate

1. Start at y_0
2. Loop on n:

$$f_n = f(t_n, y_n)$$

Value of RHS at time t_n

$$\tilde{y}_{n+1} = y_n + h f_n$$

Get next y_{n+1} value using Forward Euler

$$f_{n+1} = f(t_n + h, \tilde{y}_{n+1})$$

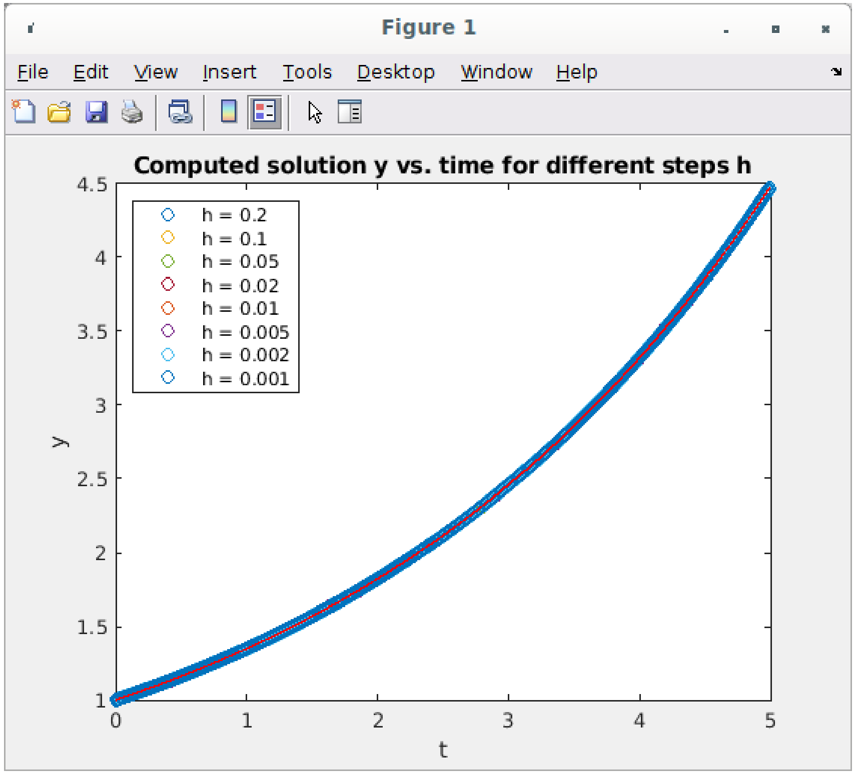
Now evaluate f_{n+1} at next step using this new value for y_{n+1}

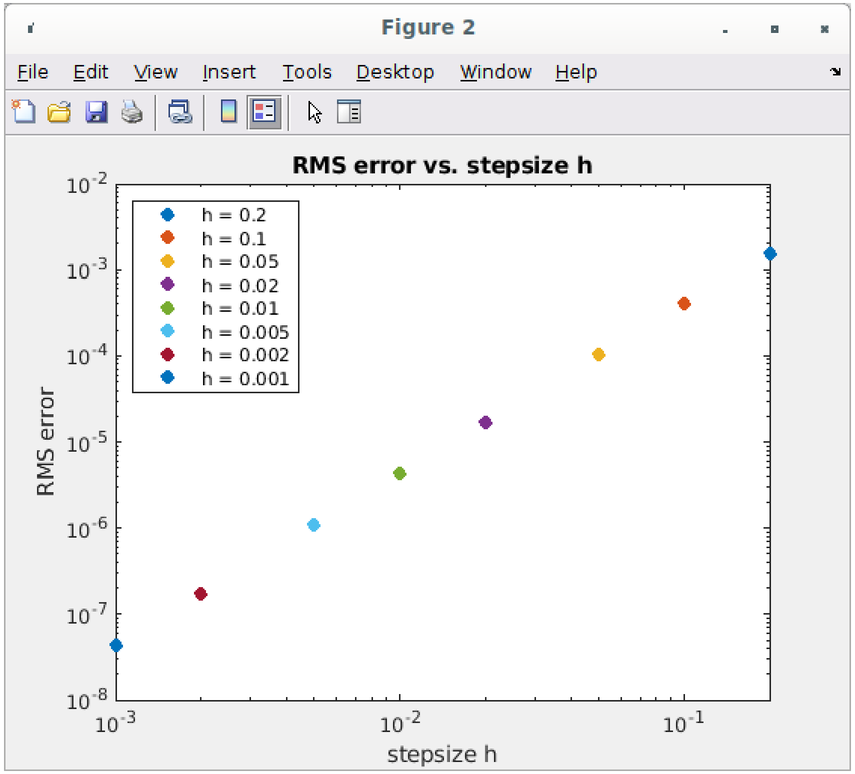
$$y_{n+1} = y_n + \frac{h}{2}\left(f_n + f_{n+1}\right)$$

Get new y_{n+1} value using average slope between t_n and t_{n+1}

3. End loop
4. Return vector y_n

GTE of Heun's method





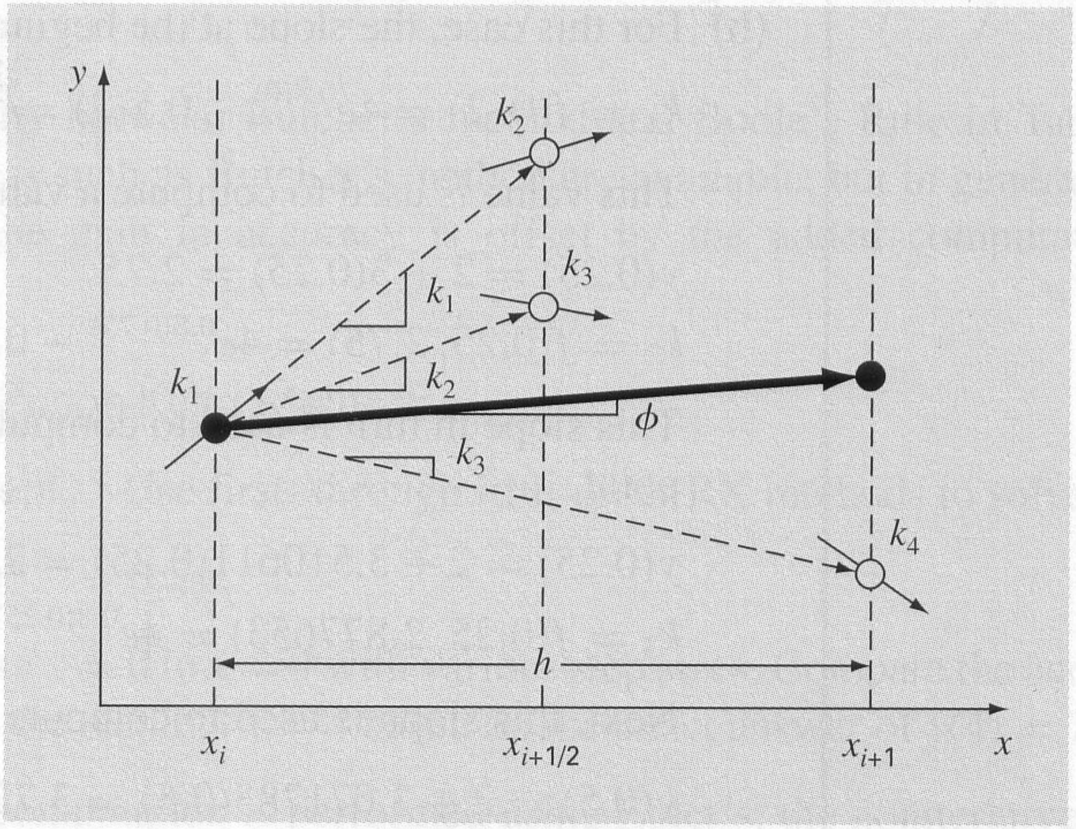
$$\frac{dy}{dt} = \alpha y \qquad y(0) = 1$$

$$RMS \sim O(h^2)$$

Runge-Kutta methods

- Main idea: Extend idea in Heun's method to compute more points between y_n and y_{n+1} for increased accuracy.

Graphical depiction of the slope estimates comprising the fourth-order RK method.



4^{th} order Runge-Kutta

- All-purpose workhorse

$$k_1 = hf(y_n, t)$$

$$k_2 = hf(y_n + k_1/2, t + h/2)$$

$$k_3 = hf(y_n + k_2/2, t + h/2)$$

$$k_4 = hf(y_n + k_3, t + h)$$

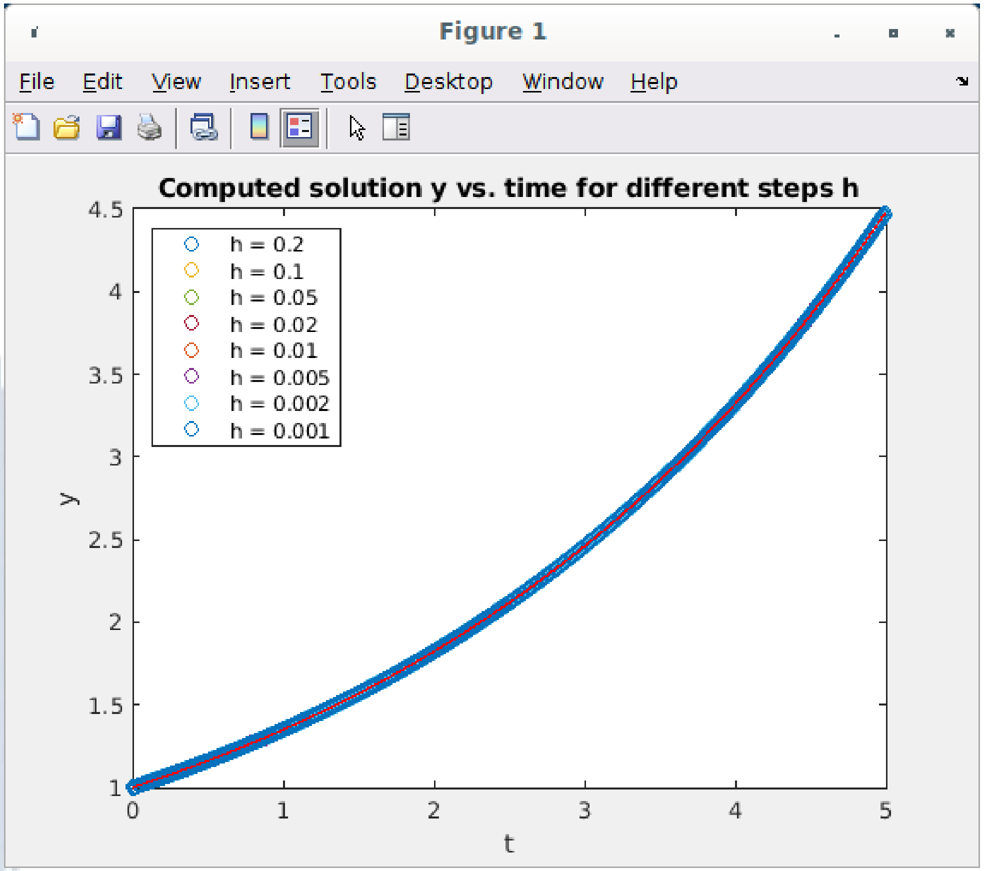
$$y_{n+1} = y_n + (k_1 + 2\mathrm{k}_2 + 2\mathrm{k}_2 + k_4)/6$$

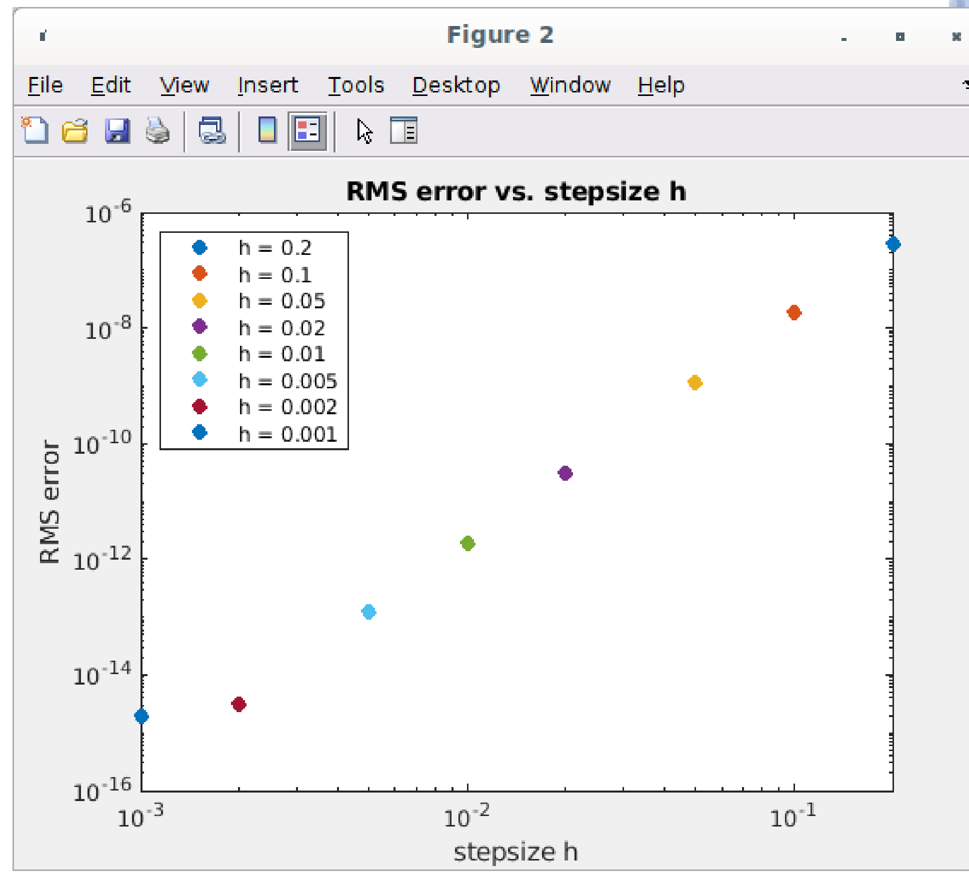
- Example implementation on Blackboard.
- Matlab: ode45, ode23.

4th order Runge Kutta

```
function y = RK4(y0, N, h)
  % This function solves the system
  % y' = f(y,t) using 4nd order Runge-Kutta
  % It takes as inputs:
  % y0 = initial value of y
  % N = Number of points to compute

  % preallocate vector y
  rows = length(y0);
  y = zeros(rows, N);
  t = 0;

  y(:,1) = y0;
  for n = 1:(N-1)
    k1 = h*f(y(:,n), t);
    k2 = h*f(y(:,n) + k1/2, t+h/2);
    k3 = h*f(y(:,n) + k2/2, t+h/2);
    k4 = h*f(y(:,n) + k3, t+h);
    y(:,n+1) = y(:,n) + (k1 + 2*k2 + 2*k3 + k4)/6;
    t = t+h;
  end

end
```

RK4 – GTE





$$\frac{dy}{dt} = \alpha y \qquad y(0) = 1$$

$$RMS \sim O(h^4)$$

Explicit Runge Kutta stability regions

- Stability regions for order N
- This is for explicit RK.
- Implicit RK also exists for dealing with stiff systems.
- Matlab: ode15s, ode23s, etc.

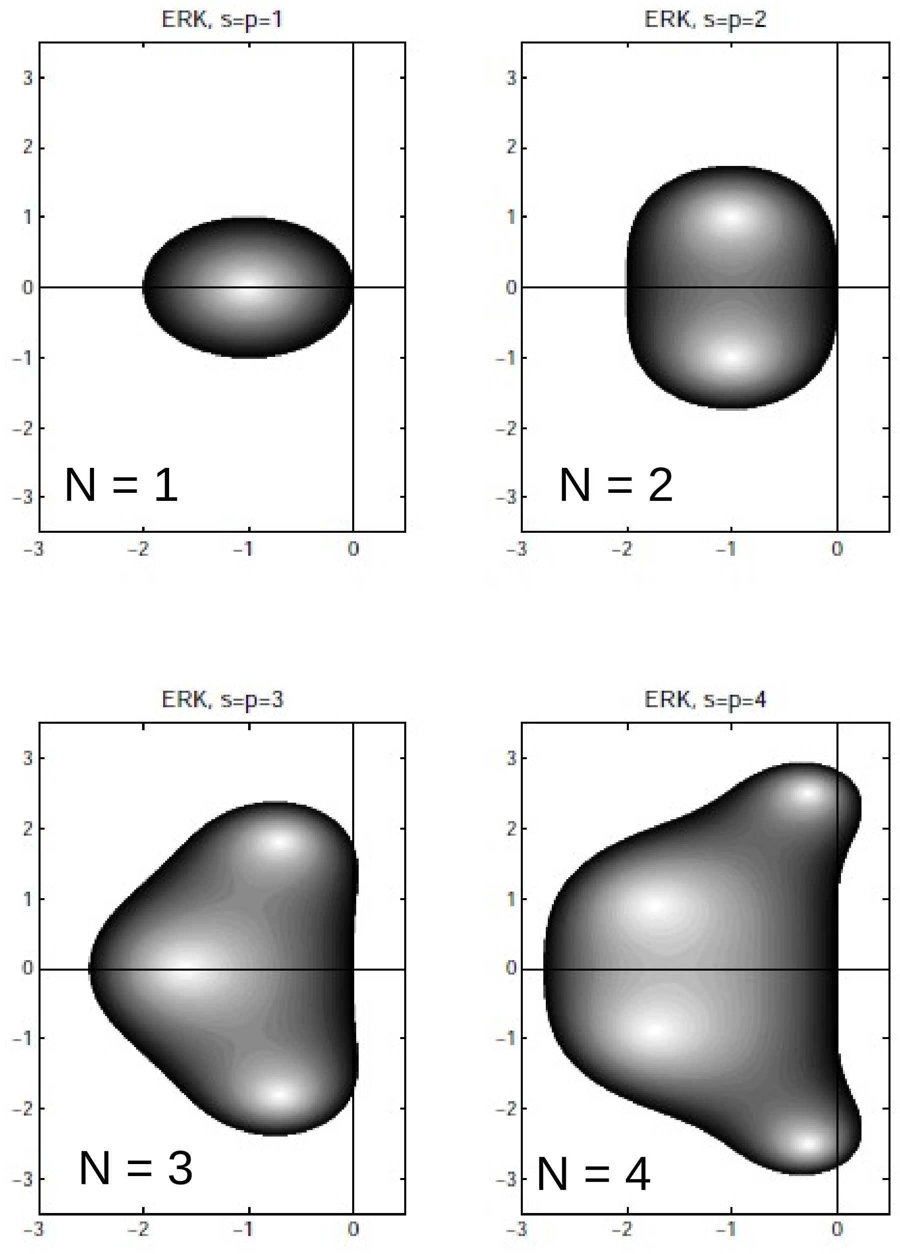


Figure 10.4: Explicit Runge-Kutta Stability Regions

Example: the van der Pol equation

- Nonlinear ODE.
- Equation comes from analysis of self-oscillations in vacuum tubes.

$$\frac{d^2x}{dt^2} - \epsilon(1-x^2)\frac{dx}{dt} + x = 0$$

- Written as a system

$$\frac{dx}{dt} = y$$

$$\frac{dy}{dt} = \epsilon(1-x^2)y - x$$

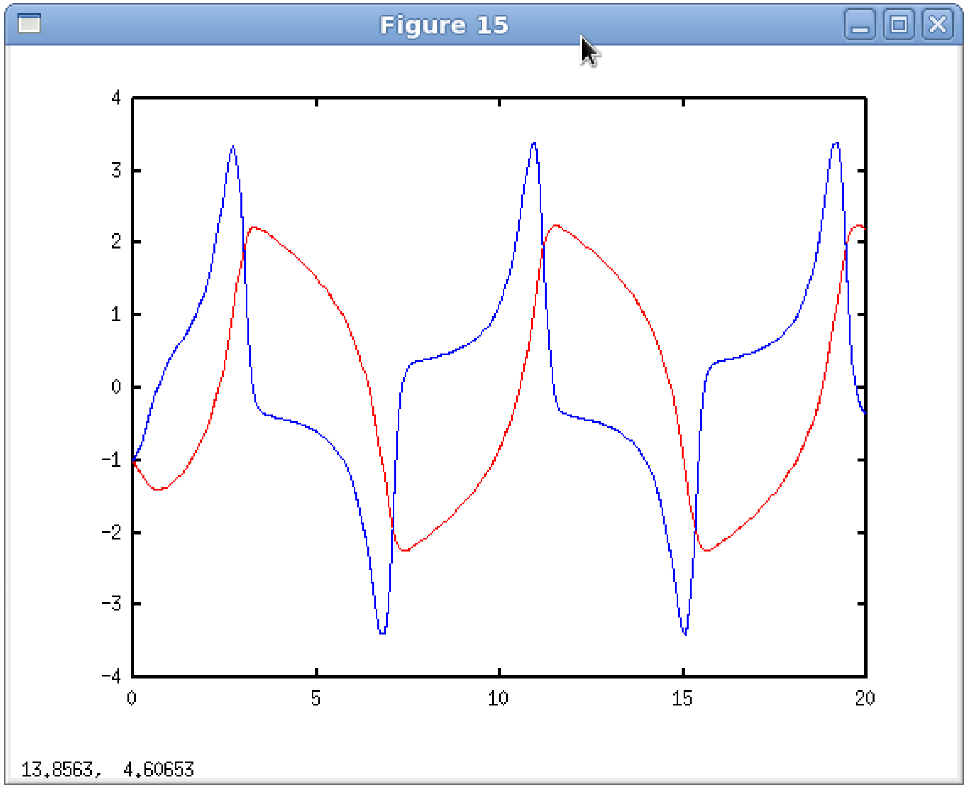
```
function y = ForwardEuler(y0, N, h)
  global epsilon;

  % create vector y
  rows = length(y0);
  y = zeros(rows, N);
  t = 0;

  y(:,1) = y0;
  for n = 1:(N-1)
    y(:,n+1) = y(:,n) + h*f(y(:,n), t);
    t = t+h;
  end

end
```

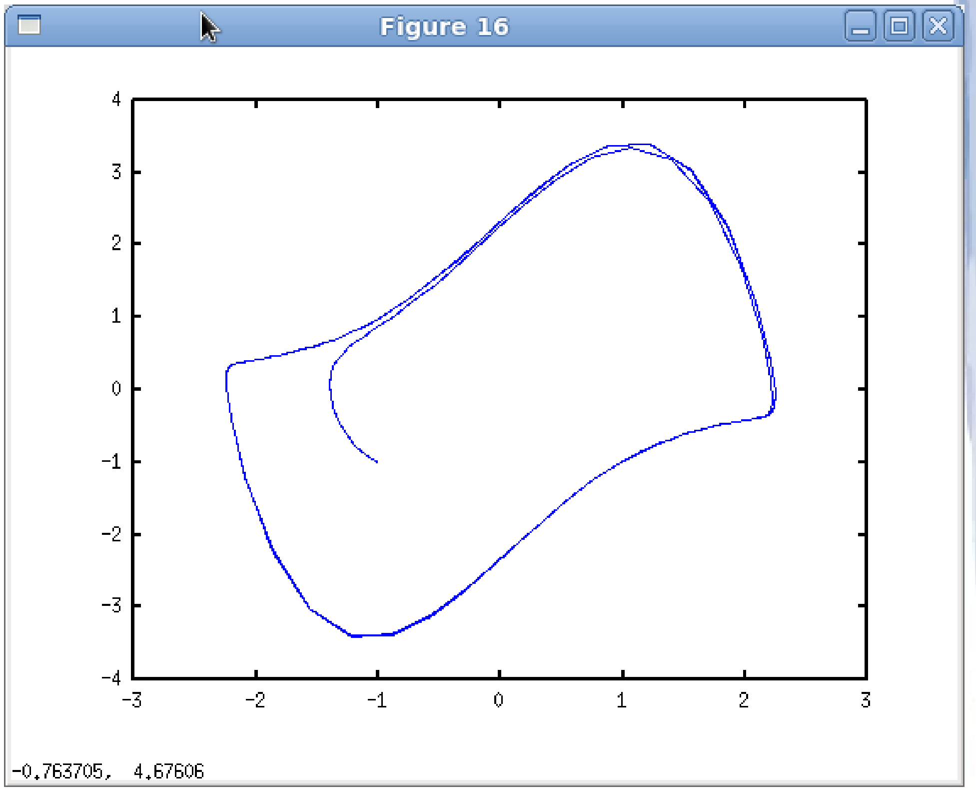
```
function dydt = f(y, t)
  % This returns the van der Pol system.
  % y is a col vector
  global epsilon

  dydt = zeros(2,1);

  dydt(1) = y(2);
  dydt(2) = epsilon*(1-y(1)*y(1))*y(2) - y(1);

end
```

```
function y = RK4(y0, N, h)
  % This function solves the system
  % y' = f(y,t) using 4nd order Runge-Kutta
  % It takes as inputs:
  % y0 = initial value of y
  % N = Number of points to compute

  global epsilon;

  % create vector y
  rows = length(y0);
  y = zeros(rows, N);
  t = 0;

  y(:,1) = y0;
  for n = 1:(N-1)
    k1 = h*f(y(:,n), t);
    k2 = h*f(y(:,n) + k1/2, t+h/2);
    k3 = h*f(y(:,n) + k2/2, t+h/2);
    k4 = h*f(y(:,n) + k3, t+h);
    y(:,n+1) = y(:,n) + (k1 + 2*k2 + 2*k3 + k4)/6;
    t = t+h;
  end

end
```

```
function TestRK4()
  % This function calls RK4 with the
  % variables needed to run it.

  global epsilon;

  % Set up parameters in equation
  epsilon = 1.5;

  % Step size to use
  h = .1;

  % Length of time to compute
  Tmax = 20;

  % Number of points to compute
  N = Tmax/h;

  % Initial condition
  y0 = [-1;-1];

  % Time vector -- used in plotting
  t = linspace(0, h*N, N);

  % Computed solution using 4th order Runge-Kutta
  y = RK4(y0, N, h);

  figure
  plot(y(1,:), y(2,:))

end
```

Numerical solution





Time solution

Phase space

- Behavior for Forward Euler and RK4 is similar.
- Both methods stable for this system & parameter values.

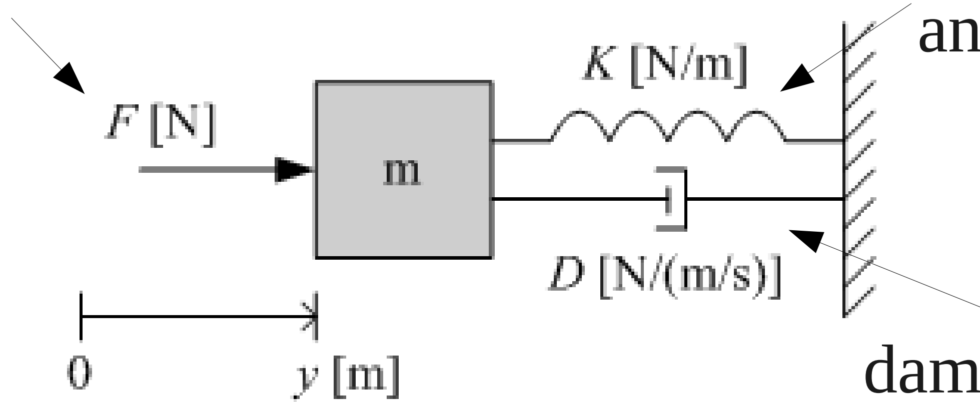
Different example: Duffing's equation

$$\frac{d^2 y}{dt^2} + \delta \frac{dy}{dt} + \alpha y + \beta y^3 = \gamma \cos(\omega t)$$

- Model of driven mass-spring system with nonlinear spring

forcing function $\gamma \cos(\omega t)$

spring constant α and non-linearity β



damping factor δ

- Classic example of simple system evincing complicated behavior

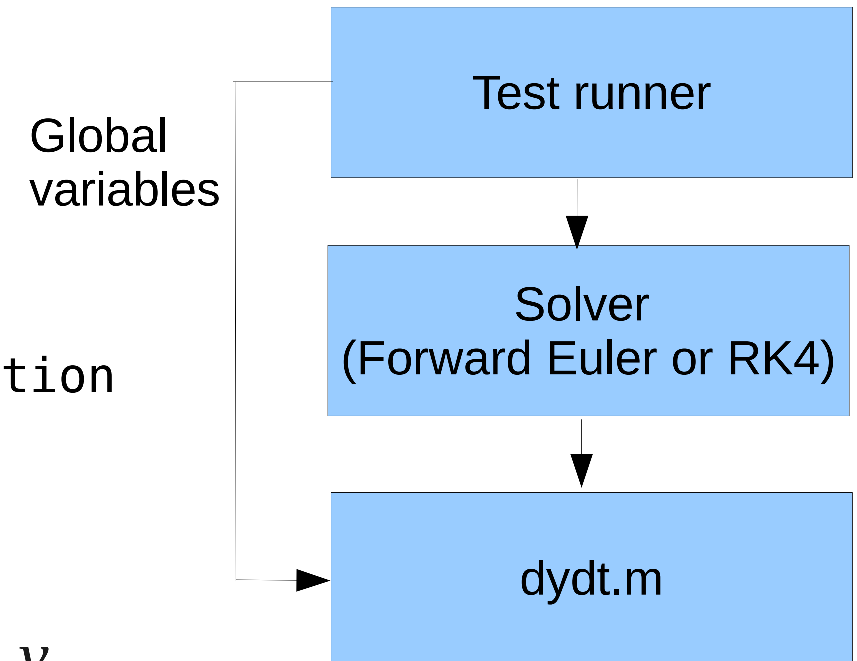
Write as 1st order system

$$\frac{d^2y}{dt^2}+\delta\frac{dy}{dt}+\alpha y+\beta y^3=\gamma\cos(\omega t)$$

- Set $y_1=y$
- Define $y_2=\frac{dy_1}{dt}$
- Then $\frac{dy_2}{dt}=\frac{d^2y_1}{dt^2}=-\delta\frac{dy_1}{dt}-\alpha y_1-\beta y_1^3+\gamma\cos(\omega t)$
- So the system to solve is

$$\frac{dy_1}{dt}=y_2 \qquad \frac{dy_2}{dt}=-\delta\frac{dy_1}{dt}-\alpha y_1-\beta y_1^3+\gamma\cos(\omega t)$$

Solver architecture




```
function dydt = f(y, t)
  % This returns the Duffing equation
  % y is a col vector

  global alpha
  global beta
  global delta
  global gamma
  global omega

  dydt = zeros(2,1);

  dydt(1) = y(2);
  dydt(2) = -delta*y(2) - alpha*y(1) - beta*y(1)*y(1)*y(1) +
gamma*cos(omega*t);

end
```

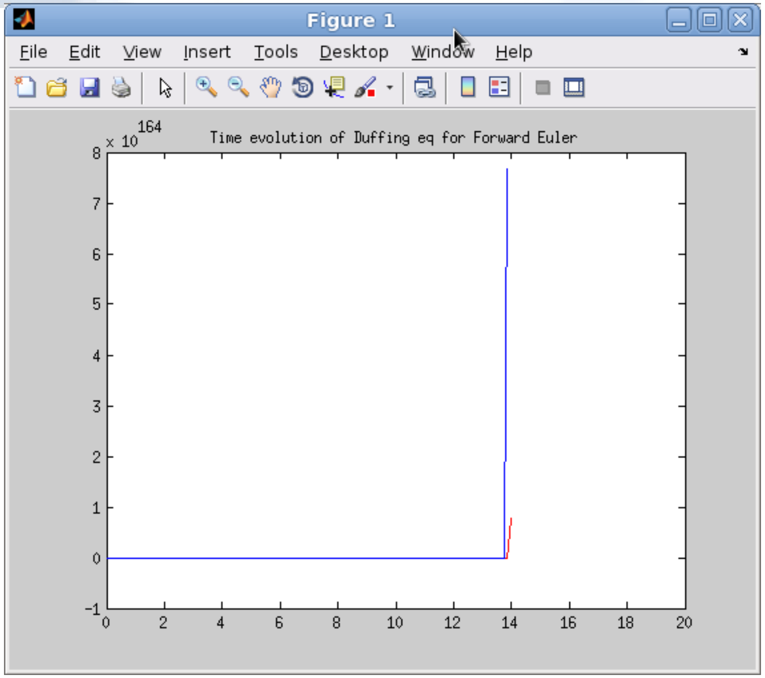
$$\frac{d\,y_1}{dt} = y_2$$

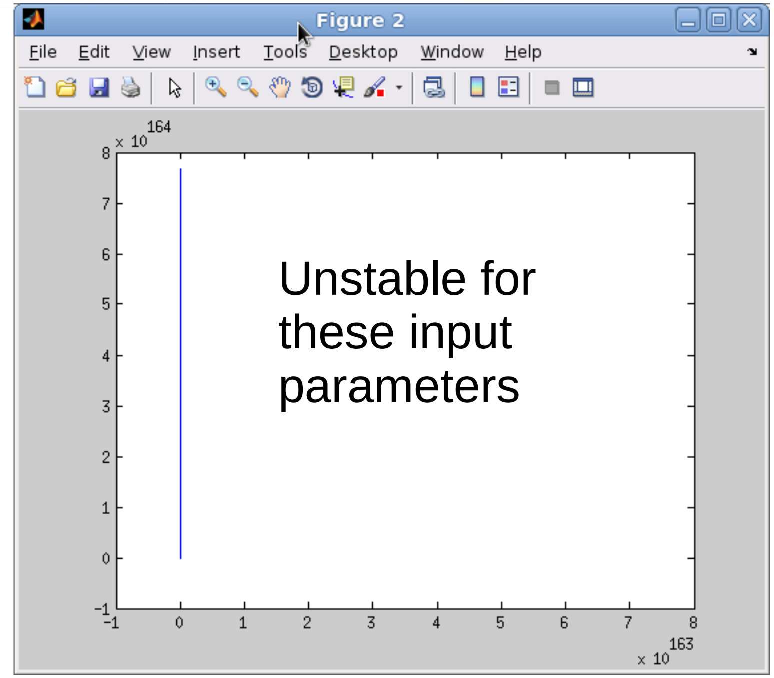
$$\frac{d\,y_2}{dt} = -\delta\,\frac{d\,y_1}{dt} - \alpha\,y_1 - \beta\,y_1^3 + \gamma\cos(\omega\,t)$$

```
function TestRK4()

  % Set up parameters in equation
  delta = .2;
  alpha = 1.9;
  beta = .5;
  gamma = 1;
  omega = 1;

  % Step size to use
  h = .1;

  % Length of time to compute
  Tmax = 20;

  % Number of points to compute
  N = Tmax/h;

  % Initial condition
  y0 = [-1;-1];

  % Time vector -- used in plotting
  t = linspace(0, h*N, N);

  % Computed solution using 4th order Runge-Kutta
  y = RK4(y0, N, h);

  figure(3)
  plot(t, y(1,:),'r')
  hold on
  plot(t, y(2,:),'b')
  title('Time evolution of Duffing eq for RK4')

  figure(4)
  plot(y(1,:), y(2,:))
  title('Phase plot of Duffing eq for RK4')

end
```

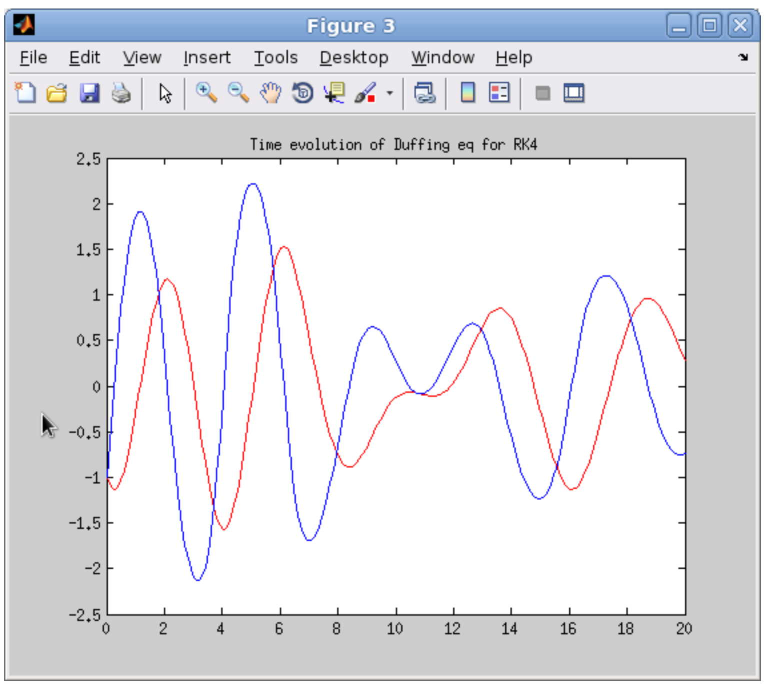
Same parameters used for RK4 and Forward Euler

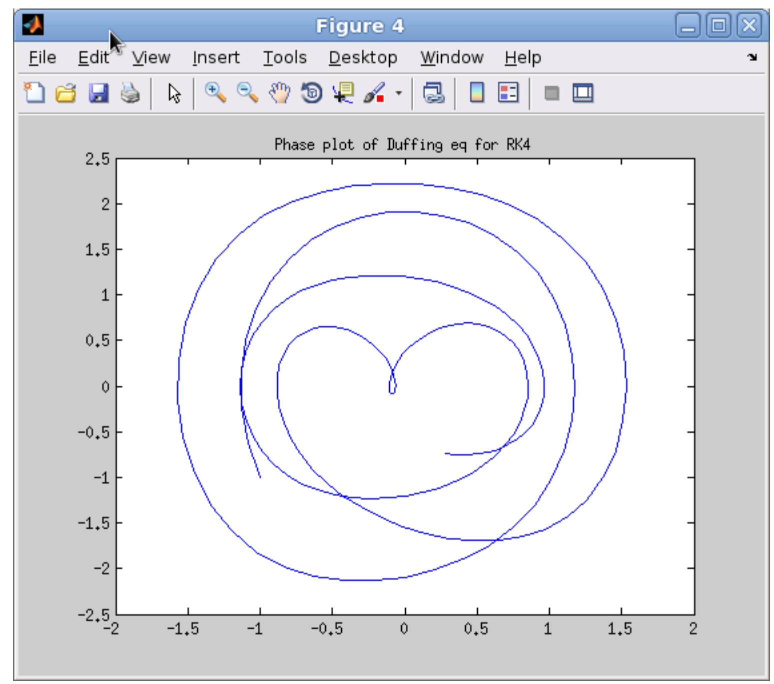
Fwd Euler





RK4



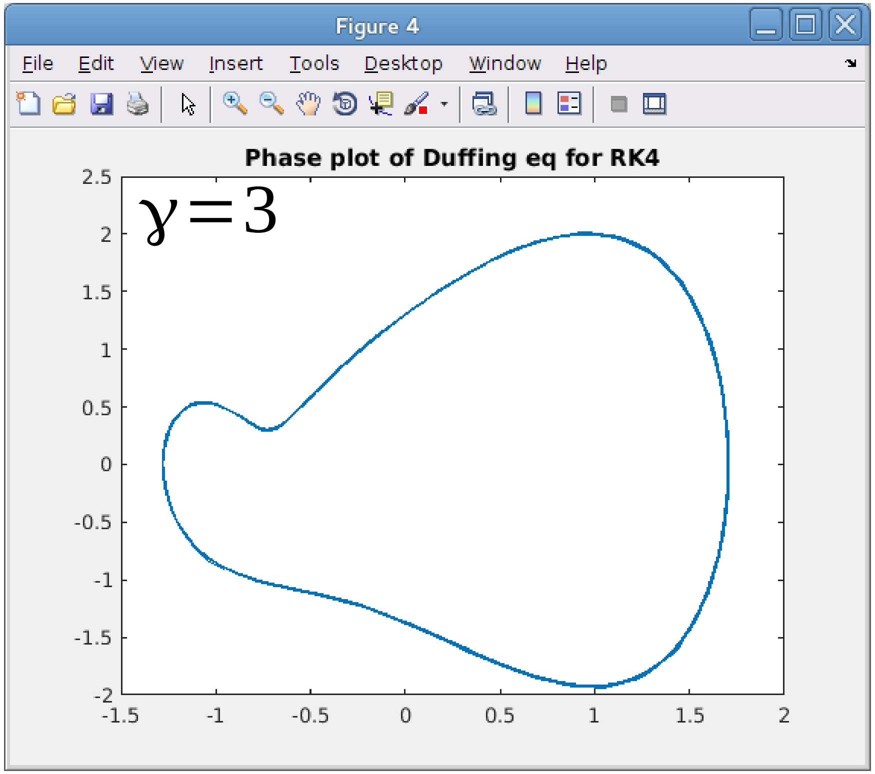


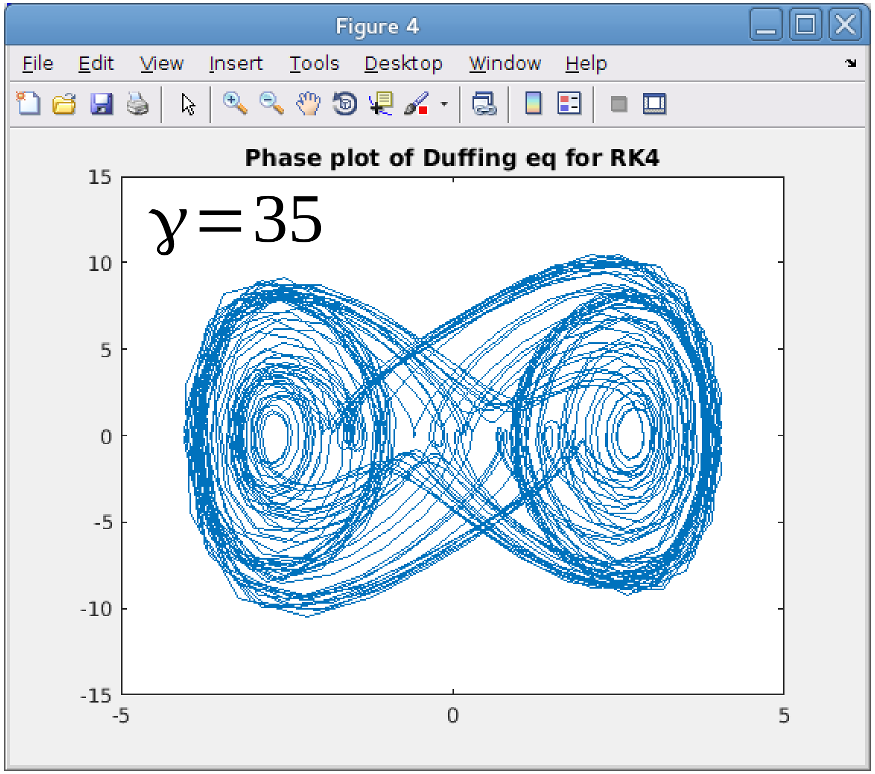
Time evolution

Phase plot

Duffing's Oscillator and Chaos

$$\frac{d^2 y}{dt^2} + \delta \frac{dy}{dt} + \alpha y + \beta y^3 = \gamma \cos(\omega t)$$





- Duffing's oscillator evidences chaotic behavior for large drive γ

First session summary

- Solving ODEs (initial value problems):
  - Euler's method(s): Explicit and implicit.
  - Concept: Local and global truncation error
  - 1D vs. ND systems of ODEs.
  - Concept: Stability
  - Runge Kutta (4^{th} order)
- Higher-order ODEs
  - Harmonic oscillator
  - van der Pol oscillator
  - Duffing's equation