Introduction to Scientific Computing

Differentiation
Integration

Computational Calculus

- The derivative for a function y(x)

$$\left.\frac{dy}{dx}\right|_{x_0} = \lim_{\Delta x \to 0} \frac{y(x_0 + \Delta x) - y(x_0)}{\Delta x}$$

- Two possible approximations

$$\left.\frac{dy}{dx}\right|_{x_0} \approx \frac{y(x_0 + \Delta x) - y(x_0)}{\Delta x}$$

$$\left.\frac{dy}{dx}\right|_{x_0} \approx \frac{y(x_0 + \Delta x) - y(x_0 - \Delta x)}{2\Delta x}$$

Accuracy of approximations

- **Taylor series expansion**

$$y(x_0+\Delta x)=y(x_0)+y'(x_0)\Delta x+\tfrac{1}{2}y''(x_0)\Delta x^2+\tfrac{1}{6}y'''(x_0)\Delta x^3+\ldots$$

$$y(x_0-\Delta x)=y(x_0)-y'(x_0)\Delta x+\tfrac{1}{2}y''(x_0)\Delta x^2-\tfrac{1}{6}y'''(x_0)\Delta x^3+\ldots$$

- **Two approximations**

$$y'(x_0)=\frac{y(x_0+\Delta x)-y(x_0)}{\Delta x}-\frac{1}{2}y''(x_0)\Delta x+\ldots$$

$$=\frac{y(x_0+\Delta x)-y(x_0)}{\Delta x}+O(\Delta x)$$

$$y'(x_0)=\frac{y(x_0+\Delta x)-y(x_0-\Delta x)}{2\Delta x}+O\left(\Delta x^2\right)$$

Choosing Δx

- If Δx is too large the approximation will suffer.
- If Δx is too small we can run into numerical precision issues.

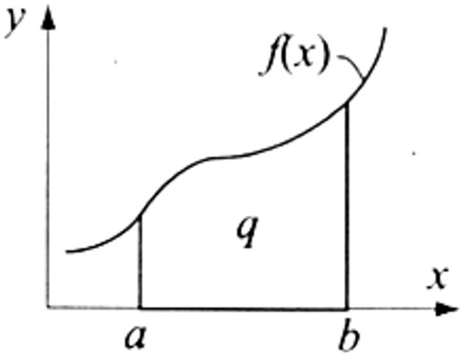
L12.m
differentiate.m
fn1.m

- MATLAB `diff(x)` function

Numerical Integration

·Approximating definite integrals

$$q = \int_a^b f(x)\,dx = \lim_{\Delta x \to 0}\left(\sum_{x=a}^{x=b} f(x)\Delta x\right)$$



- Function is *constant* over each interval
- Function varies *linearly* over each interval
  - Trapezoidal Rule
- Function varies *quadratically* over each interval
  - Simpson’s quadrature rule.
  - MATLAB `quad` function

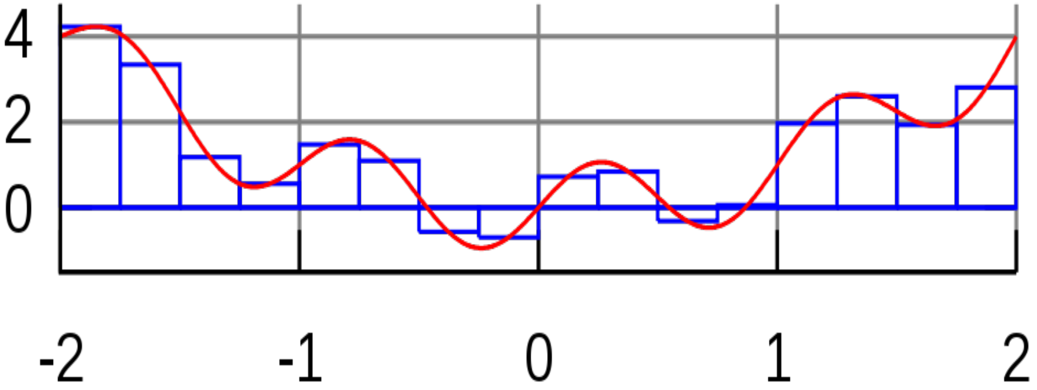
Approximating the area under the curve

- Let `y` be an array containing `= f(x)` over the interval

```
y = f(xmin:dx:xmax)
```

$$A = \int_{x_{\min}}^{x_{\max}} f(x)\,dx \approx y_1\Delta x + y_2\Delta x + y_3\Delta x + \cdots + y_{n-1}\Delta x$$

$$A \approx \Delta x \sum_{i=1}^{n-1} y_i$$



Approximating the area under the curve

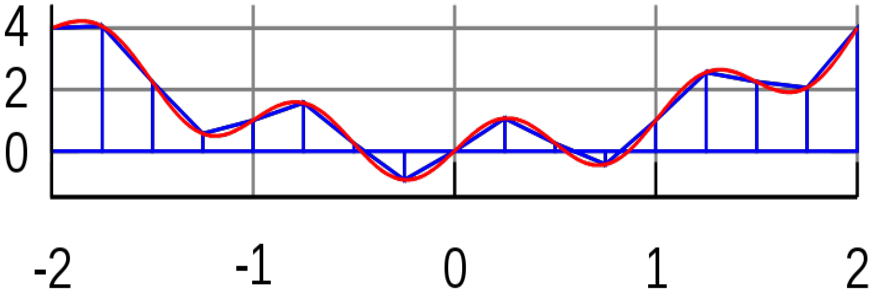
- Let `y` be an array containing `= f(x)` over the interval

```
y = f(xmin:dx:xmax)
```

- Trapezoidal Approximation, integrate.m

$$A = \int_{x_{\min}}^{x_{\max}} f(x)\,dx \approx (y_1 + y_2)\frac{\Delta x}{2} + (y_2 + y_3)\frac{\Delta x}{2} \cdots + (y_{n-1} + y_n)\frac{\Delta x}{2}$$

$$A \approx \frac{\Delta x}{2}\left(y_1 + y_n + 2\sum_{i=2}^{n-1} y_i\right)$$



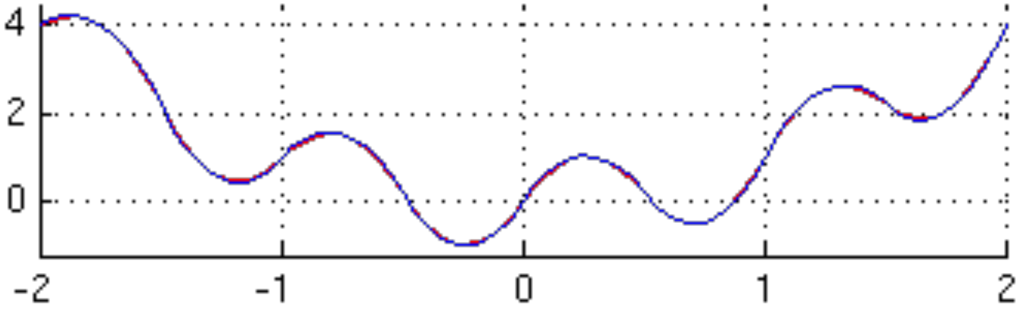
Approximating the area under the curve

- Let `y` be an array containing = `f(x)` over the interval

```
y = f(xmin:dx:xmax)
```

- Simpson's rule， simpson.m

$$A \approx \frac{h}{3}\left(y_1 + y_n + 4\sum_{i=1}^{n-1} y_{2i} + 2\sum_{i=1}^{n} y_{2i-1} \right), \quad h = \frac{x_{\max} - x_{\min}}{2n}$$

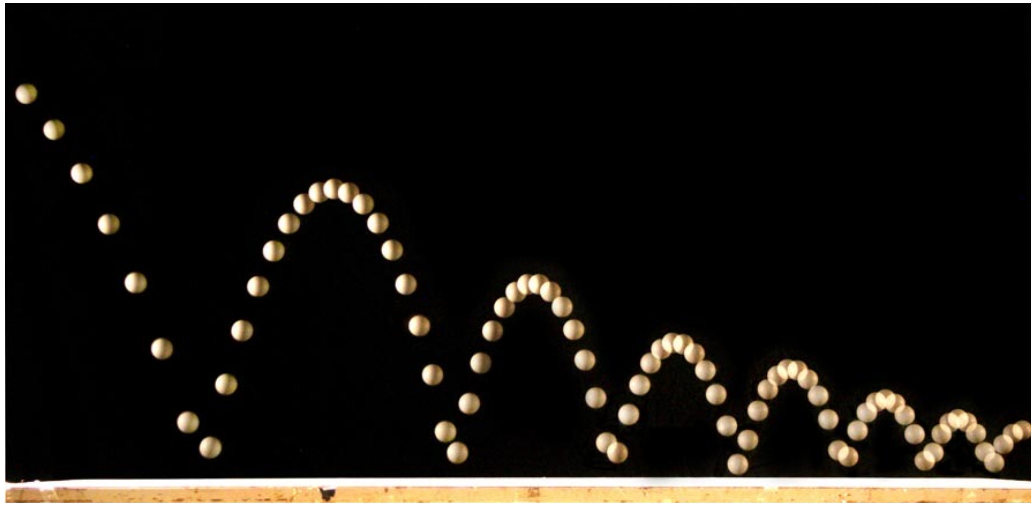


Dynamical Systems

Dynamical systems

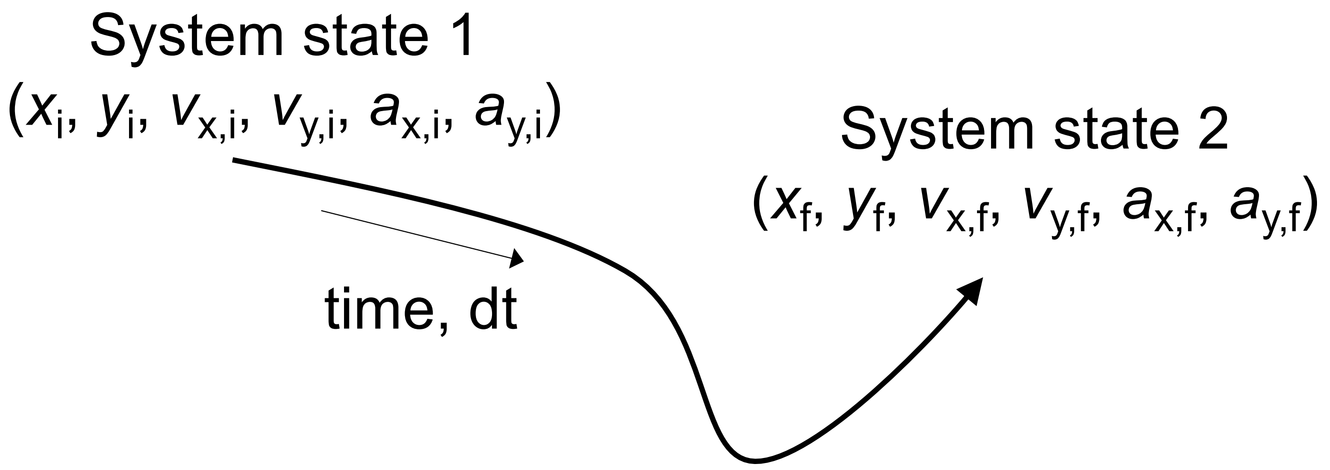
- Dynamical systems are any systems governed by transitions from one state to another state

- Most often, in physical systems, time (*t*) guides the trajectory of the system



Dynamical systems

- Discrete analysis of dynamical systems is tackled by analyzing the transition from a current state to a new state



System state 1

(x_i, y_i, $v_{\mathrm{x,i}}$, $v_{\mathrm{y,i}}$, $a_{\mathrm{x,i}}$, $a_{\mathrm{y,i}}$)

time, dt

System state 2

(x_f, y_f, $v_{\mathrm{x,f}}$, $v_{\mathrm{y,f}}$, $a_{\mathrm{x,f}}$, $a_{\mathrm{y,f}}$)

Dynamical systems

- These systems are governed by differential equations that tell us how the state of the system changes over time

$$\frac{dx}{dt} = v_x \qquad \frac{dv_x}{dt} = a_x$$

$$\frac{dy}{dt} = v_y \qquad \frac{dv_y}{dt} = a_y$$

Baseball.m

Computational simulation of dynamical systems

- **In order to simulate these systems on a computer, we compute the state of the system at evenly spaced points in time**

  e.g. *t* = 0.1, 0.2, 0.3, ... seconds (aka d*t* = 0.1 sec)

- **The general computational method is, at each instant in time,**
  - Feed in the current state of the system (jj)
  - Compute next state of the system (jj+1)

Computational simulation of dynamical systems

- Time is discretized by breaking up the dynamic equations into the current state and the next state often using first order approximations

Differential representation	Discrete representation	MATLAB representation
$v_x = \dfrac{dx}{dt} \approx \dfrac{x(t+dt)-x(t)}{dt}$	$\Delta x = v_x \Delta t$	x(jj+1)=x(jj)+v_x(jj) x timestep
$a_x = \dfrac{dv_x}{dt} \approx \dfrac{v_x(t+dt)-v_x(t)}{dt}$	$\Delta v_x = a_x \Delta t$	v(jj+1)=v(jj)+a_x(jj) x timestep

Discretizing Time

- **In our program the differential equations take on the form of expressions which we use to compute the next state as a function of the current state.**

In our example

$$\frac{dx}{dt} \approx \frac{x(t+dt)-x(t)}{dt} = v_x$$

```
x(t+dt) = x(t) + vx(t)*dt
```

$$\frac{dy}{dt} \approx \frac{y(t+dt)-y(t)}{dt} = v_y$$

```
y(t+dt) = y(t) + vy(t)*dt
```

$$\frac{dv_x}{dt} \approx \frac{v_x(t+dt)-v_x(t)}{dt} = a_x = 0$$

```
vx(t+dt) = vx(t)
```

$$\frac{dv_y}{dt} \approx \frac{v_y(t+dt)-v_y(t)}{dt} = a_y = -g$$

```
vy(t+dt) = vy(t) - g*dt
```

BaseballSimulator.m

Model with viscous drag

- Viscous drag adds another acceleration to the ball which is proportional to the balls velocity and acts to slow it down, *i.e.* -c*(vx, vy) where c is the viscous drag coefficient.

$$a_x = \frac{f_x}{m} = -c \cdot v_x$$

$$a_y = \frac{f_y}{m} = -c \cdot v_y - g$$

- We can account for drag by simply modifying our update equations:

```
x(t+dt) = x(t) + vx(t)*dt
y(t+dt) = y(t) + vy(t)*dt
vx(t+dt) = vx(t) - c*vx(t)*dt
vy(t+dt) = vy(t) - (g + c*vy(t))*dt
```

BaseballSimulatorWithDrag.m

Model with Collisions

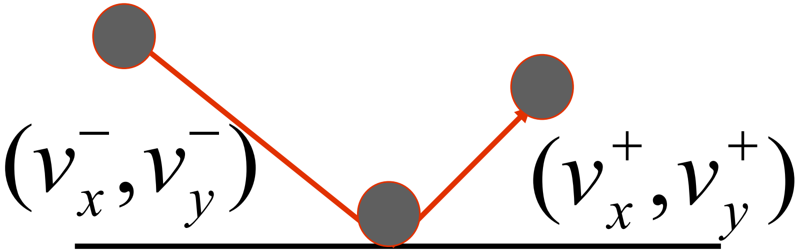
- **In the real world, we would expect our ball to bounce when it hits the ground.**

- **In our simulation we can deal with this fact as follows:**
  - Detect when the collision occurs by monitoring the height of the ball `y(t)` and checking when it goes negative
  - Determine exactly when the collision occurs (collision time)
  - Model the collision
  - Restart the simulation immediately after the collision (`dt` -collision time)

Modeling a collision

- We model a collision with the ground (or a horizontal plane) by flipping the sign of the y velocity

- Loss of energy is included via the coefficient of restitution, ***ϕ***, a number between 0 and 1, which relates velocities before and after the collision

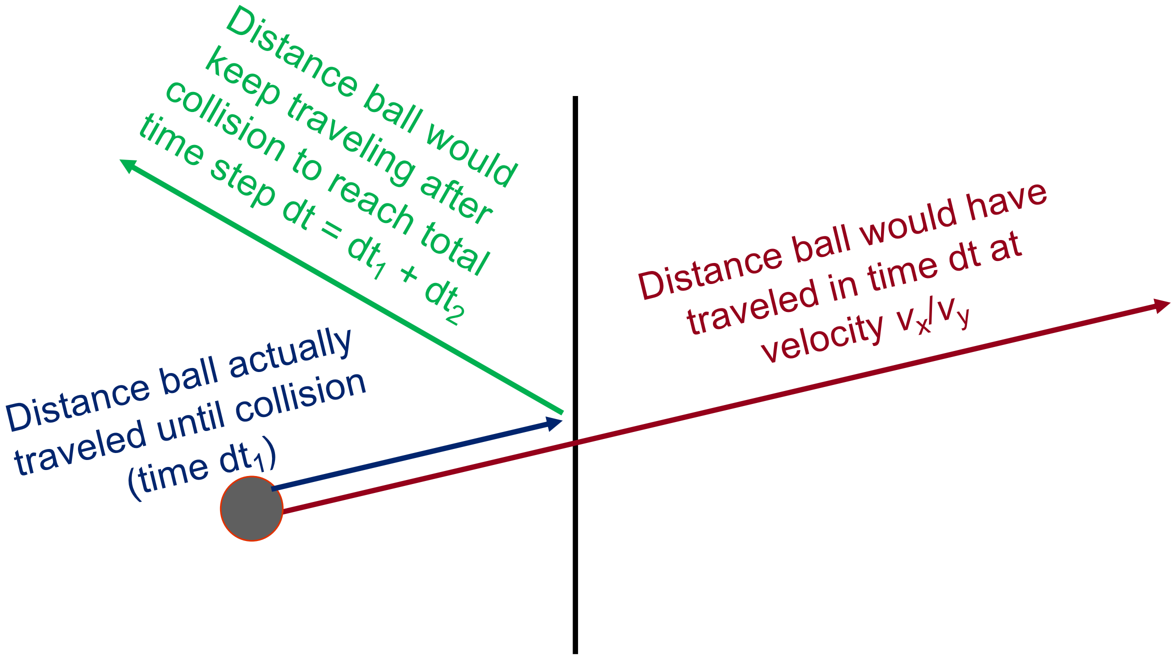


$$v_x^+ = \phi v_x^-$$

$$v_y^+ = -\phi v_y^-$$

Modeling a collision

- **How do we deal with finite time steps when modeling a collision?**

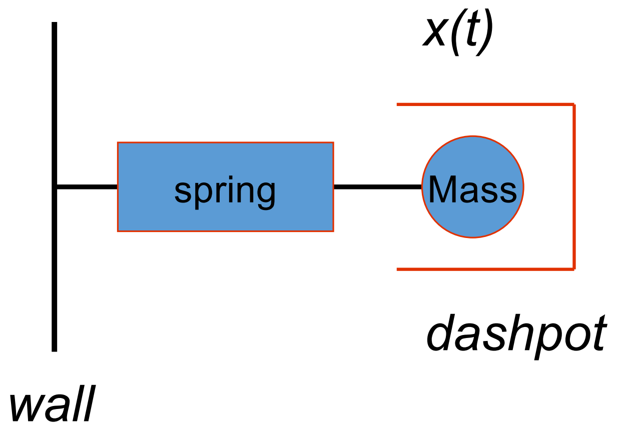

Distance ball would keep traveling after collision to reach total time step $dt = dt_1 + dt_2$
Distance ball would have traveled in time dt at velocity v_x/v_y
Distance ball actually traveled until collision (time dt_1)

Spring Mass Dashpot System

- **Consider a mass connected to a spring and a dashpot**
  - A classic example of a second order mechanical system used to model car suspension systems, pogo sticks, biomechanical systems etc.

SpringMassDashpot.m

  - The force acting on the mass is related to its displacement, *x*(*t*), and velocity, *v*(*t*), as follows:



$$F(t) = m\frac{d^2x}{dt^2} = -kx(t) - cv(t)$$

Other Examples

- **Planetary simulation**

  PlanetarySimulation.m

- **Planets in 2D**

  planets.m

- **String Simulation**

  StringSimulation.m

Stochastic Systems

- **There are many situations where we are interested in simulating the behavior of a system driven by random processes.**

- **A classic example is Brownian motion named after Robert Brown who studied the motion of pollen particles suspended in a liquid**
  http://en.wikipedia.org/wiki/Brownian_motion

  - We can simulate the motion of these particles by updating the position of the particles by a random step at each time instant
  - One can imagine other systems, like the stock market, where the state of the system at one instant depends probabilistically on its state at the previous instant

brownianMotion.m

Chaotic Systems

- Most of the systems that we are interested in studying are ‘well behaved’ in the sense that the motions either die off over time or the system settles into a regular pattern or oscillation. However, they don’t all do that.

- Systems that display ‘weird’ or unpredictable behavior over time are referred to as chaotic and they occur fairly often.

The Lorenz Attractor

- **A classic example of chaos theory was described by Edward Lorenz - a meteorologist at MIT**

  http://en.wikipedia.org/wiki/Lorenz_attractor

He was interested in simulating the behavior of a simple looking set of differential equations

$$\frac{dx}{dt} = \sigma(y - x) \qquad (\sigma, \rho, \beta)$$

$$\frac{dy}{dt} = x(\rho - z) - y$$

$$\frac{dz}{dt} = xy - \beta z$$

Lorenz Attractor

- **Prof. Lorenz observed that his simulation behaved ‘weirdly’ and never seemed to settle into a coherent pattern**

- **He also observed that small changes in initial conditions would lead to large changes in the final result of the system**

  This led him to coin the term ‘The Butterfly Effect’ which helps to explain why it is difficult to predict the weather

Lorenz.m, LorenzAttractor.m

N-body systems

- Another classic example of chaos in nature is due to Poincaré who pointed out that planetary systems with more than 3 bodies can demonstrate chaotic behavior.

  http://en.wikipedia.org/wiki/Three-body_problem#Three-body_problem

Planets.m

ODE

Solving Single First Order ODE

$$\boxed{\begin{array}{l} \dfrac{dy}{dt} = f(y,t) \ \text{ for } t_0 \le t \le t_f \\ \text{with } y = y_0 \ \text{ at } t = t_0 \end{array}}$$

- Single independent variable `t` (time)
- Single dependent variable `y`
- Solution is a function `y=y(t)` that satisfies the ODE

- Steps for solving with MATLAB
  1. Write in standard form
  2. Define function in MATLAB
  3. Select method of solution
  4. Solve

Numerical solutions to differential equations (Euler)

- Consider

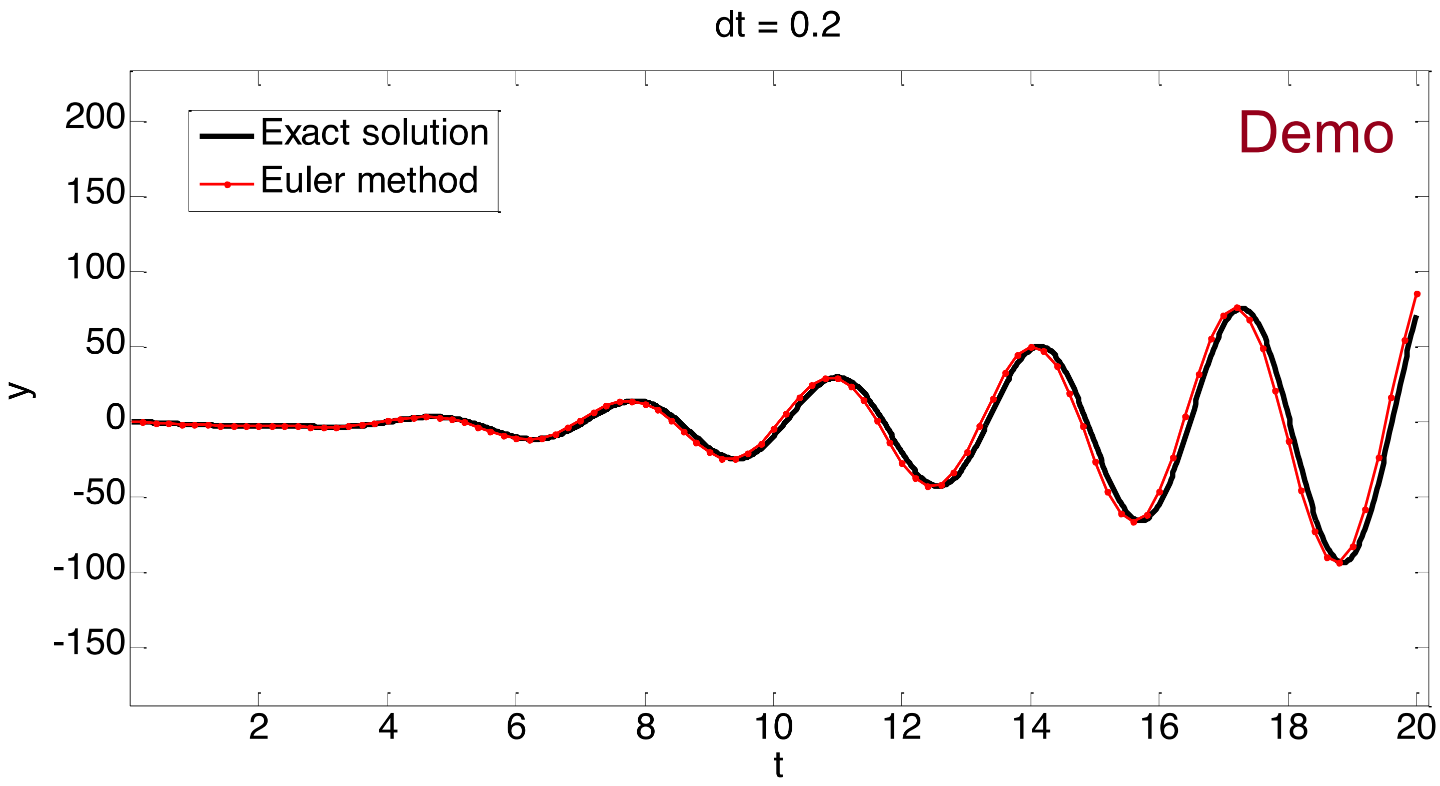
$$\frac{dx(t)}{dt} = f\left(t, x(t)\right)$$

- A first order approximation for the next state of the system can be constructed using the trajectory of the system (Euler's method)

$$x(t+\Delta t) \approx x(t) + f\left(t, x(t)\right)\Delta t$$

- However, this simple approximation is often very inaccurate

Visual interpretation of the Euler method



Numerical solutions to differential equations (predictor-corrector)

- **Consider**

$$\frac{dx(t)}{dt} = f\left(t, x(t)\right)$$

- A better approximation can be constructed by considering the effective derivative over the entire interval

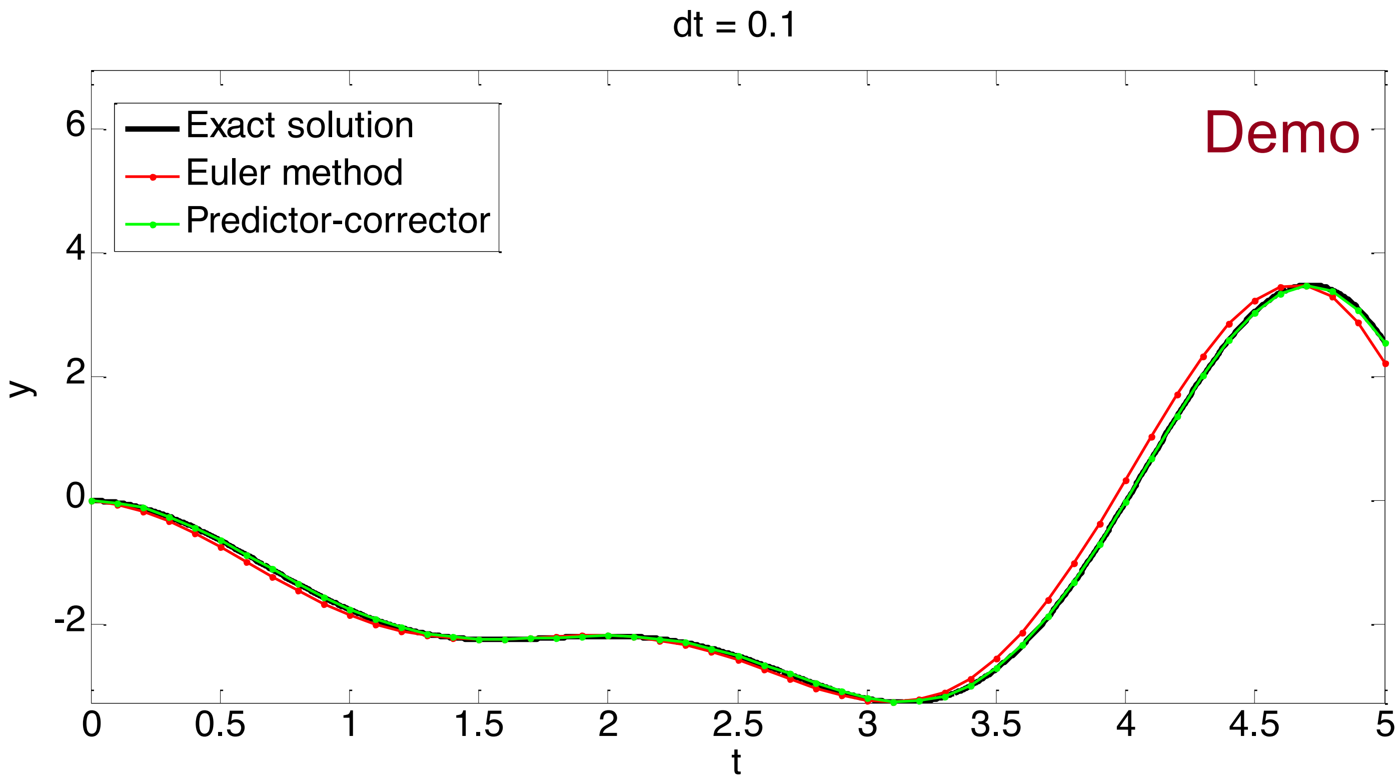
$$x(t+\Delta t) \approx x(t) + 1/2 \times (k_1 + k_2)\Delta t$$

$$k_1 = f\left(t, x(t)\right)$$

$$k_2 = f\left(t+\Delta t, x(t) + k_1 \Delta t\right)$$

- Note that this method uses an estimate for the derivative at the beginning, k_1, and ending, k_2, of the interval and increases accuracy of the solution

Visual comparison of the Euler and predictor-corrector methods



Numerical solutions to differential equations (Runge-Kutta)

- Following the logic of the predictor-corrector method, the accuracy of the estimate can be improved by evaluating the derivative at its midpoint

$$x(t+\Delta t) \approx x(t) + 1/6 \times (k_1 + 2k_2 + 2k_3 + k_4)\Delta t$$

$$k_1 = f\left(t, x(t)\right)$$

$$k_2 = f\left(t + \frac{\Delta t}{2}, x(t) + k_1 \frac{\Delta t}{2}\right)$$

$$k_3 = f\left(t + \frac{\Delta t}{2}, x(t) + k_2 \frac{\Delta t}{2}\right)$$

$$k_4 = f\left(t + \Delta t, x(t) + k_3 \Delta t\right)$$

where

$$\frac{dx(t)}{dt} = f\left(t, x(t)\right)$$

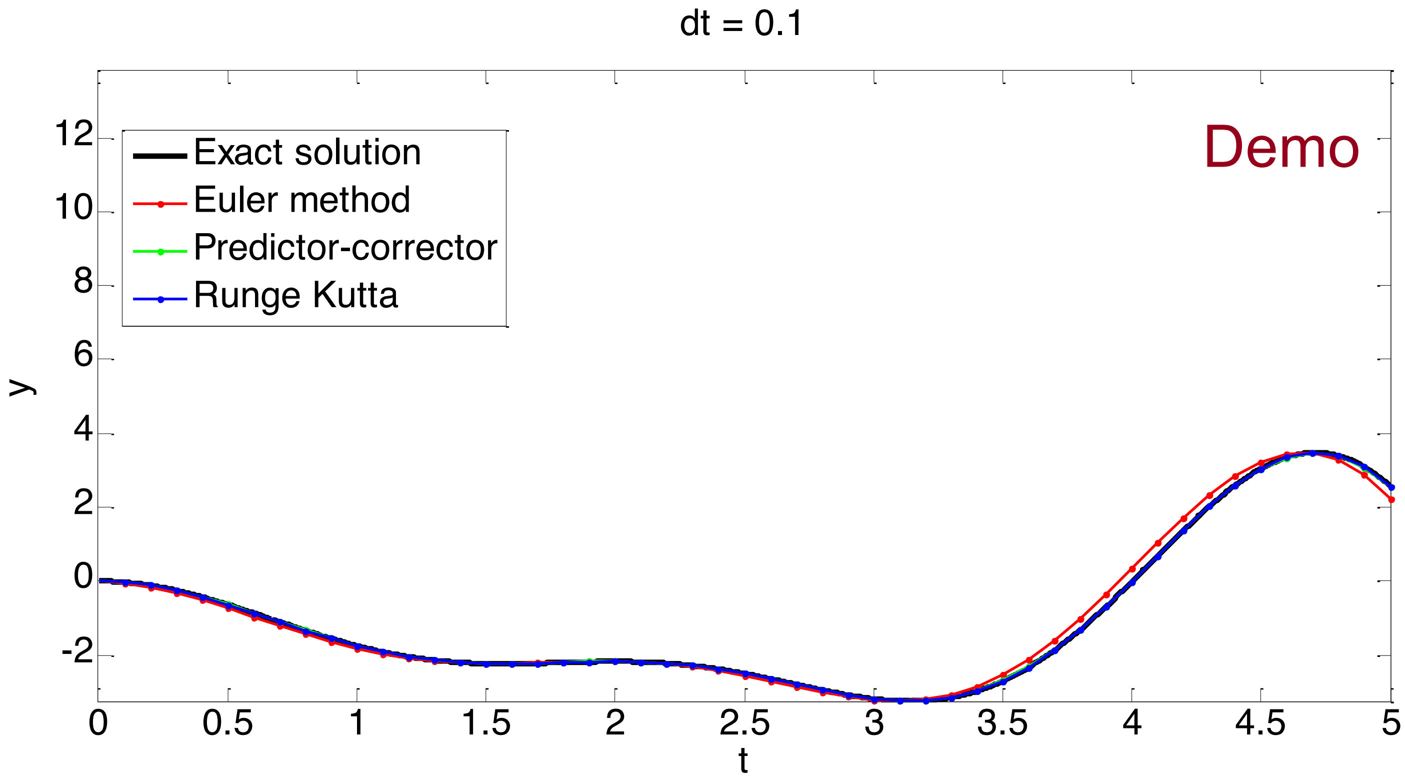
Numerical solutions to differential equations (Runge-Kutta)

- Note that in the last slide k_1 corresponds to the derivative at the beginning of the interval, k_2 and k_3 are two estimates for the derivative at the midpoint of the interval, and k_4 is an estimate of the derivative at the end of the interval

- The example of the last slide would be a Runge-Kutta approximation of rank 4 (RK4)

- Runge-Kutta ODE solvers of RK23 and RK45 can be accessed with the MATLAB commands `ode23` and `ode45`

Visual comparison of the Euler, predictor-corrector, and Runge Kutta methods



Demo

Numerical accuracy

- **Essential Matlab states:**

  “How do we ever know when we have the ‘right’ numerical solution?

  Well, we don’t - the best we can do is increase the accuracy of the numerical method until no further wild changes occur over the interval of interest.”

- **DEMO of this convergence for cantilever dynamics.**

Example: Solving an ODE

- **Write in standard form**

$$\frac{dy}{dt} = \frac{t^3 - 2y}{t} \quad \text{for } 1 \le t \le 3 \quad \text{with } y = 4.3 \quad \text{at } t = 1$$

- Define the function in MATLAB

```
dydt
```

- Select method of solution

```
ode45, ode23, ode113, ode15s, ode23s, ode23t, ode23tb
```

- Solve

```
[t y] = ODE_solver_name(ODEfun, tspan, y0)
```

A simple example

- **Lets do this in MATLAB**

$$\frac{dy}{dt} = 2y \xrightarrow{\text{Generalize the variables}} \dot{x}_1 = 2x_1$$

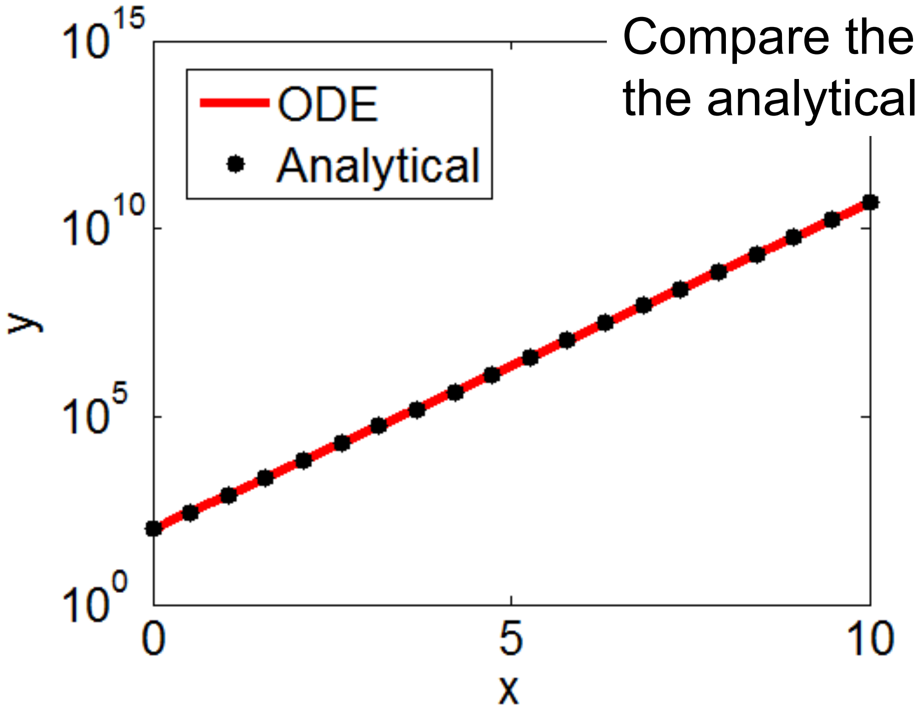
Define the ODE

```
function dx = firstOrderODE(t,x)
dx = 2*x;
```

Solve the ODE

```
% Solve the ODE
[t,x] = ode45(@firstOrderODE,[0 10],[100]);

% Generate values for the analytical solution
tTest = linspace(0,10,20);
xTest = 100*exp(2*tTest);
```

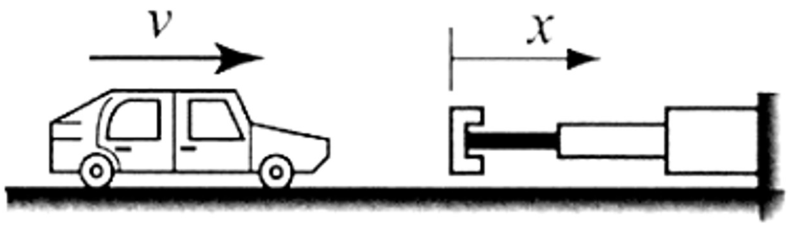
A simple example



Compare the ODE and the analytical solution

Example - Car Crash & Safety Bumper

- **Force applied by bumper is a function of the velocity *v* and displacement *x* of the front edge according to**



$$F = Kv^3(x+1)^3$$

where K = 30 s-kg/m^5.

Car has mass m=1500kg and hits bumper at 90 km/hr (25 m/s).

Plot velocity of the car as a function of position.

Acceleration: $$a = -Kv^3(x+1)^3 / m$$

Kinematic relationship $$a = \frac{dv}{dt} = \frac{dv}{dx}\frac{dx}{dt} = \frac{dv}{dx}v \quad \text{or} \quad \frac{dv}{dx} = -\frac{Kv^2(x+1)^3}{m}$$

ODEs of higher orders - state space representation

What is a state space representation?

We take this...

$$\frac{d^n y(t)}{dt^n} + a_1 \frac{d^{n-1} y(t)}{dt^{n-1}} + \ldots + a_{n-1} \frac{dy(t)}{dt} + a_n y(t) = F$$

$$\downarrow$$

...and turn it into this

$$\underline{\dot{x}}(t) = \underline{A}\underline{x}(t) + \underline{B}u(t)$$

Note: this representation is only true for linear ODEs

ODEs of higher orders - state space representation

In other words - say we have an equation of the form

$$\dddot{y} + p\ddot{y} + q\dot{y} + ry = 0$$

$$\dot{y} = \text{something}$$

This would be input into MATLAB as a series of differential equations of first order

ODEs of higher orders - state space representation

We start by assigning generalized variables (say x) to derivatives of y of increasing order as such

$$x_1 = y$$
$$x_2 = \dot{y}$$
$$x_3 = \ddot{y}$$

ODEs of higher orders - state space representation

Next, we rearrange the ODE to get the variable of highest order on the left side of the equation and divide out any coefficients on this variable (if they exist)

$$\dddot{y} = -p\ddot{y} - q\dot{y} - ry$$

ODEs of higher orders - state space representation

Next, we rewrite the derivatives of the generalized variables with respect to each other and the ODE

$$\begin{aligned}\dot{x}_1 &= x_2 \\ \dot{x}_2 &= x_3 \\ \dot{x}_3 &= -px_3 - qx_2 - rx_1\end{aligned}$$

Note that the number of linear ODEs corresponds to the order of the original equation

ODEs of higher orders - state space representation

This equation can then be input into a function definition of the ODE for use in MATLAB

```
function xout = thirdOrderODE(t,x)

p = 2;
q = 1.5;
r = 0.1;

dx1 = x(2);
dx2 = x(3);
dx3 = -p*x(3)-q*x(2)-r*x(1);

xout = [dx1; dx2; dx3];
```

Note: time and initial condition serve as inputs

Note: the derivatives of the generalized variables are returned as a column vector

ODEs of higher orders - state space representation

An alternate formulation of the ODE function

```
function xout = thirdOrderODE(t,x)

p = 2;
q = 1.5;
r = 0.1;

dx(1)  = x(2);
dx(2)  = x(3);
dx(3)  = -p*x(3)-q*x(2)-r*x(1);

xout = [dx(1);dx(2);dx(3)];
```

ODEs of higher orders - state space representation

Another alternate formulation of the ODE function

```
function dx = thirdOrderODE(t,x)

p = 2;
q = 1.5;
r = 0.1;

dx = zeros(3,1);
dx(1) = x(2);
dx(2) = x(3);
dx(3) = -p*x(3)-q*x(2)-r*x(1);
```

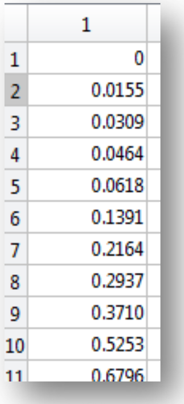
ODEs of higher orders - state space representation

We can then solve the function by calling the ode45 solver with respect to the ODE function

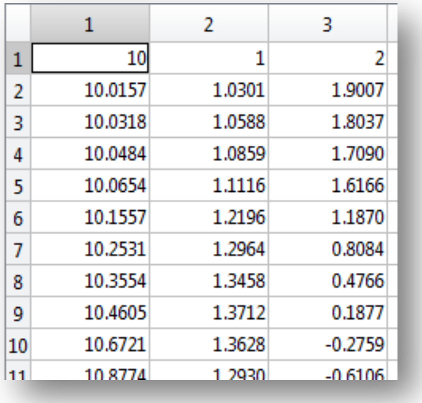
```
[t,x] = ode45(@thirdOrderODE,[0 10],[10 1 2]);
```

Row vector of initial conditions x(1) = 10, x(2) = 1, etc.; the number of initial conditions equals the order of the ODE

Row vector indicates the start and end time of the solution



	1
1	0
2	0.0155
3	0.0309
4	0.0464
5	0.0618
6	0.1391
7	0.2164
8	0.2937
9	0.3710
10	0.5253
11	0.6796



	1	2	3
1	10	1	2
2	10.0157	1.0301	1.9007
3	10.0318	1.0588	1.8037
4	10.0484	1.0859	1.7090
5	10.0654	1.1116	1.6166
6	10.1557	1.2196	1.1870
7	10.2531	1.2964	0.8084
8	10.3554	1.3458	0.4766
9	10.4605	1.3712	0.1877
10	10.6721	1.3628	-0.2759
11	10.8774	1.2930	-0.6106

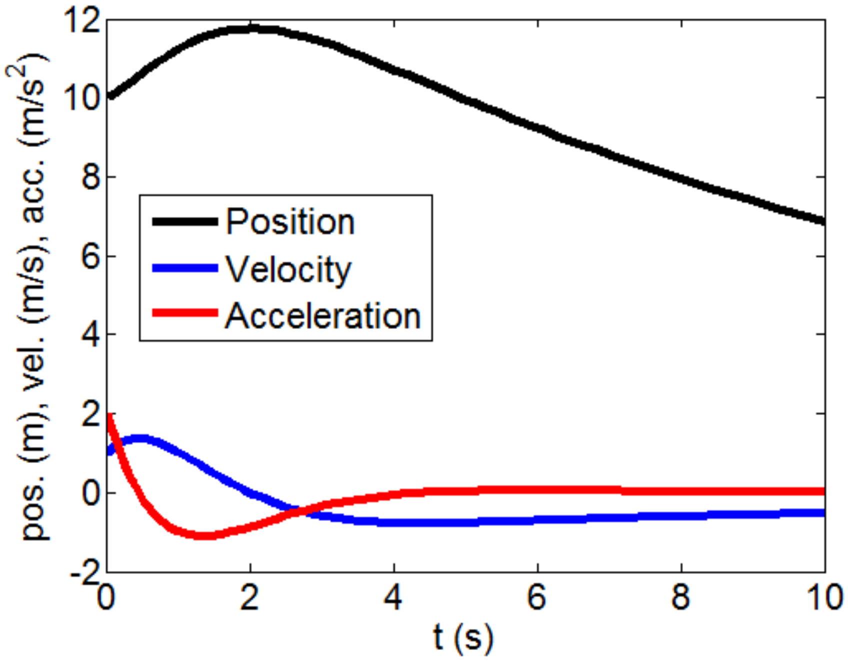
ODEs of higher orders - state space representation

We can then plot the position, velocity, and acceleration of the system

```
p = plot(t,x(:,1),'k-',t,x(:,2),'b-',t,x(:,3),'r-');
set(p,'LineWidth',4)
set(gca,'FontSize',16)
xlabel('t (s)')
ylabel('pos. (m), vel. (m/s), acc. (m/s^2)')
legend('Position','Velocity','Acceleration')
```

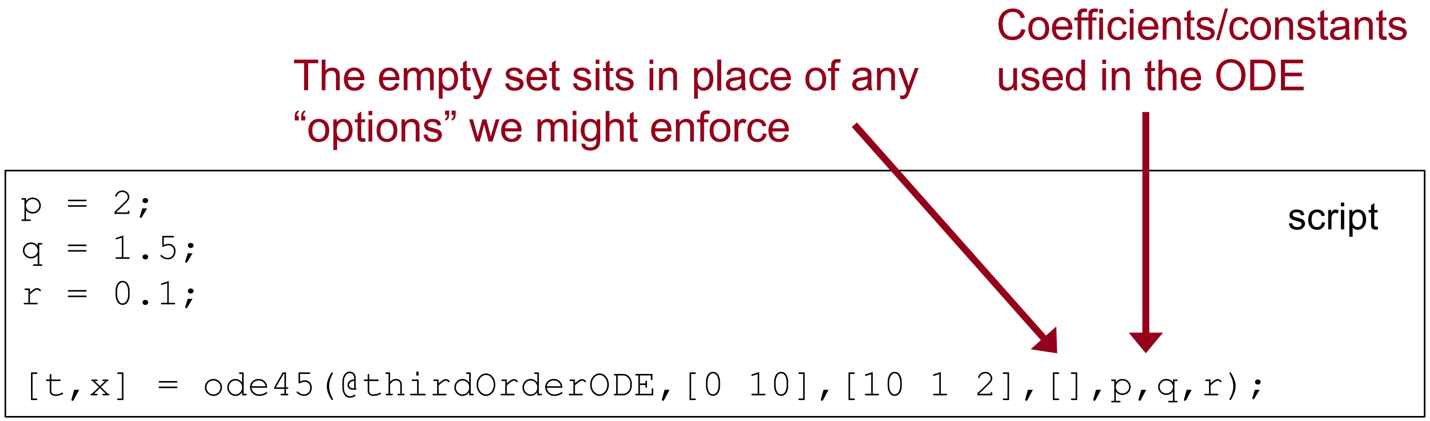
ODEs of higher orders - state space representation

The resulting plot



ODEs of higher orders - state space representation

The values of *p*, *q*, and *r* were defined in the ODE function - we can also supply these values when calling the solver



The empty set sits in place of any “options” we might enforce

Coefficients/constants used in the ODE

script

```
p = 2;
q = 1.5;
r = 0.1;

[t,x] = ode45(@thirdOrderODE,[0 10],[10 1 2],[],p,q,r);
```

ODE function

```
function dx = thirdOrderODE(t,x,p,q,r)
dx = zeros(3,1);
dx(1) = x(2);
dx(2) = x(3);
dx(3) = -p*x(3)-q*x(2)-r*x(1);
```

Systems of differential equations - Lorentz equations and chaos

Consider the following system of Lorenz equations

- This system is representative of chaotic systems in which minor changes to the input result in radically different result (butterfly effect)

- Chaos: When the present determines the future, but the approximate present does not approximately determine the future

$$\frac{dx}{dt} = -\sigma x + \sigma y$$
$$\frac{dy}{dt} = -\rho x - y - xz$$
$$\frac{dz}{dt} = -\beta z + xy$$

Systems of differential equations - Lorentz equations and chaos

How do we convert this system into a MATLAB "friendly" format

$$x_1 = x \qquad x_2 = y \qquad x_3 = z$$

$$\frac{dx}{dt} = -\sigma x + \sigma y$$

$$\frac{dy}{dt} = -\rho x - y - xz$$

$$\frac{dz}{dt} = -\beta z + xy$$

$$\longrightarrow$$

$$\dot{x}_1 = -\sigma x_1 + \sigma x_2$$

$$\dot{x}_2 = -\rho x_1 - x_2 - x_1 x_3$$

$$\dot{x}_3 = -\beta x_3 + x_1 x_2$$

Here, x_1, x_2, and x_3 refer to x(1), x(2), and x(3) in MATLAB

Numerical solution of the Lorentz system using MATLAB

script: lorenz_launcher

```
% Intitial conditions (x1, x2, x3)
x0 = [-8 8 27];

% Time span
tspan = [0 40];

% Using the ode45 solver to numerically evalute the ODEs
[t,x] = ode45(@lorenz,tspan,x0);
```

Note the function handle

function: lorenz

```
function dx = lorenz(t,x)

sigma = 10;
beta = 8/3;
rho = 28;

dx = [-sigma*x(1)+sigma*x(2);...
    rho*x(1)-x(2)-x(1)*x(3);...
    -beta*x(3)+x(1)*x(2)];
```

Note that dx(1), dx(2), and dx(3) are directly assigned to columns

Numerical solution of the Lorentz system using MATLAB

Alternate syntax for calling `ode45`

script: lorenz_launcher

```
% Intitial conditions (x1, x2, x3)
x0 = [-8 8 27];

% Time span
tspan = [0 40];

% Using the ode45 solver to numerically evalute the ODEs
[t,x] = ode45('lorenz',tspan,x0);
```

Function declaration as “string”

Numerical solution of the Lorentz system using MATLAB

Alternate syntax for calling for the function holding the differential equations

function: lorenz

```
function dx = lorenz(t,x)

sigma = 10;
beta = 8/3;
rho = 28;

dx = zeros(3,1);
dx(1) = -sigma*x(1)+sigma*x(2);
dx(2) = rho*x(1)-x(2)-x(1)*x(3);
dx(3) = -beta*x(3)+x(1)*x(2);
```

The outputs are first established as a 3x1 vector

Numerical solution of the Lorentz system using MATLAB

Alternate syntax for calling for the function holding the differential equations

function: lorenz

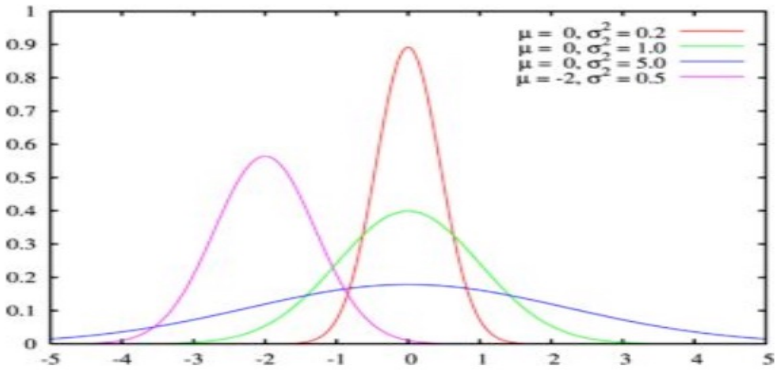
```
function dout = lorenz(t,x)

sigma = 10;
beta = 8/3;
rho = 28;

dx = -sigma*x(1)+sigma*x(2);
dy = rho*x(1)-x(2)-x(1)*x(3);
dz = -beta*x(3)+x(1)*x(2);

dout = [dx;dy;dz];
```

The outputs are combined at the output to a 3x1 vector

Random Numbers

Random Numbers

`rand`	Single random number between 0 and 1
`rand(1,n)`	Row vector of n random numbers between 0 and 1
`rand(n)`	n x n matrix of random numbers between 0 and 1
`rand(m,n)`	m x n matrix of random numbers between 0 and 1
`randperm(n)`	Row vector with random permutation of integers 1 through n
`randn(m,n)`	m x n matrix of normally distributed random numbers with mean 0 and standard deviation 1



Random Numbers (cont.)

- **Interval (a,b)**

  `(b-a)*rand + a`

- **Random integers**

  `round(rand)` **for random 0's and 1's**

  or use

  `round, floor, ceil` **on a multiple of** `rand`

- **Normal distribution mean M and standard deviation S**

  `S*randn(m,n) + M`

Random Number Generator

- **Write a function implementing a (pseudo-)random number generator**

- **Based on unpredictability of the modulus function** `mod(·,·)` **when applied to large numbers**
  - For instance, `ni+1 = mod(8121 ni + 28441, 134456)`
  - Generated numbers are uniformly distributed in range [0,134456)
  - For unknown parameters, predicting outcome `ni+1` is hard

- **Use above idea to produce random values in [0,1)**
  - Write function that generates one or more random numbers ran in range `0≤ran<1` each time is called, based on equation

    ```
    rani = ni / 134456
    ```

  - Generated numbers depend on initial value `n0` called `seed`

Random Number Generator

Problem statement

- write function rand0 returning an array ran of one or more numbers uniformly distributed in [0, 1)
- input arguments are (m, n) producing an m×n array or (m) producing an m×m array
- seed n_0 is chosen or reset by calling function seed

Input/output

- seed: integer input, no output
- rand0: input is one or two integers specifying size of the array, output is the array with random values in [0,1)

Random Number Generator

•Algorithm description

- Pseudocode for rand0:

```
function ran = random(m,n)
Set n ← m if second dimension is not supplied
Create output array with "zeros" function
iseed stores the current random-looking integer n_i
For every row i and every column j
iseed ← mod (8121 iseed + 28441, 134456)
ran(i,j) ← iseed/134456
```

- Pseudocode for seed:

```
function seed(new_seed)
new_seed ← round(new_seed)
iseed ← abs(new_seed)
```

Random Number Generator

- **Turn algorithm into MATLAB code**
  - Turning pseudocode to code is straightforward
  - Data management issue
    - Is there any data shared by the two functions?
    - They both need access to the current pseudorandom number `iseed`
    - Variable `iseed` will be placed in the global memory so that it may be accessed by both function `rand0` and `seed`
- **Test functions**
  - By resetting seed, produced numbers should repeat
  - Calculate the average and standard deviation of array and confirm uniform distribution