LECTURE 7: Numerical Solvers

Modeling and Simulation 2

Daniel Georgiev

Winter 2014

OUTLINE

- Review
- Algebraic equations
- Numerical integration
- Modelling
- Options

DAE System

expressed as fully implicit system

```
%MODULES:
%   - module 1: core network w voltage sensor and power sensor
%   - module 2: timer for tap changer
%   - module 3: timer for capacitor bank
%   - module 4: load
%   - module 5: capacitor bank
%   - module 6: wind turbine
```

DAE System

expressed as fully implicit system

$$
\begin{aligned}
r_{1,1-2} &= nT\ \left(V0 - nT\ V2\right)/(i\ X1) - Is - (Ic + (V2 - V3)/(i\ X2)) \\
r_{1,3-4} &= Sd - conj(V3)(V2 - V3)/(i/X2) \\
r_{1,5} &= \dot{V0} \\
r_{2,1} &= x\dot{T}T - onT \\
r_{2,2} &= yTT - xTT + tresT \\
r_{3,1} &= x\dot{T}C - onC \\
r_{3,2} &= yTC - xTC + tresC \\
r_{4,1} &= \dot{xd} + (Pd - Ps)/Tp \\
r_{4,2} &= Pd - xd - Pt\left|V3\right|^2 \\
r_{4,3} &= Qd \\
r_{5,1-2} &= Ic - i\ V2\ C\ nC \\
r_{6,1-2} &= -V2 + Is(Rs + i\ Xs) + i\ Ir\ Xm \\
r_{6,3-4} &= i\ Is\ Xm + i\ Ir\ Xr - Ir\ Rr/(wr/ws - 1) \\
r_{6,5} &= -Jg\ \dot{wr} + Tm + imag\left(Is\ conj(Is\ Xs/ws + Ir\ Xm/ws)\right)
\end{aligned}
$$

DAE System

expressed as fully implicit system

```
%DESCRIPTION OF CONTINUOUS STATE VECTOR
% x0{1} = [V0R V0I V2R V2I V3R V3I Pd Qd IsR IsI IcR IcI];
% x0{2} = [xTT yTT];
% x0{3} = [xTC yTC];
% x0{4} = [xd Pd Qd V3R V3I];
% x0{5} = [IcR IcI V2R V2I];
% x0{6} = [IsR IsI IrR IrI wr V2R V2I];
```

```
%DESCRIPTION OF PARAMETER VECTOR:
% p{1} = [X1 X2 Vm Vp ns Qm Qp];
% p{2} = [tauT];
% p{3} = [tauC];
% p{4} = [Tp Ps Pt];
% p{5} = [C nCT];
% p{6} = [Xs Xm Xr Rs Rr ws Jg Tm];
```

```
%DESCRIPTION OF PARAMETER VECTOR:
% p{1} = [X1 X2 Vm Vp ns Qm Qp];
% p{2} = [tauT];
% p{3} = [tauC];
% p{4} = [Tp Ps Pt];
% p{5} = [C nCT];
% p{6} = [Xs Xm Xr Rs Rr ws Jg Tm];
```

INITIAL STATE AND DISCRETE RESETS

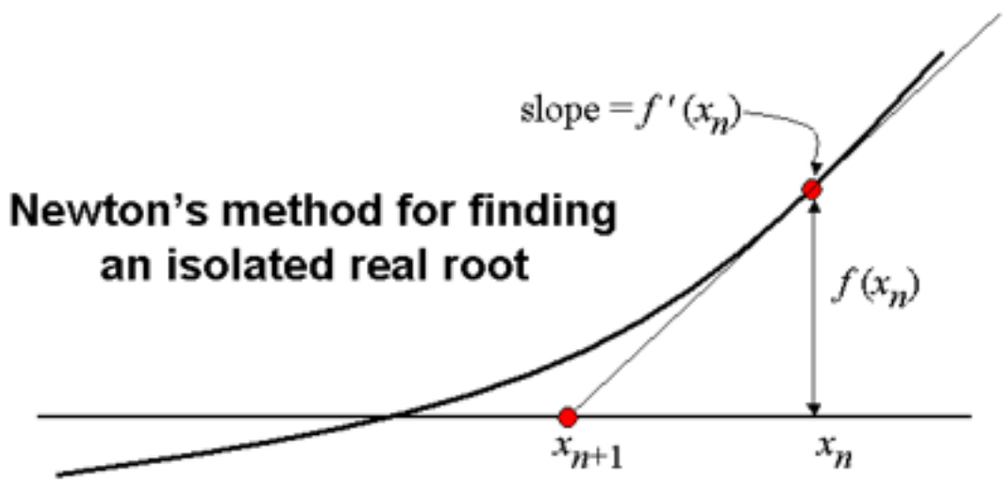
- PROBLEM: not all initial states are defined in DAE systems
- At t=0 and at t = te (discrete event times), must ensure state is consistent with DAEs and with previous state values
- MATLAB provides a function decic

```
[x0,d0] = decic(@(t,x,d)odefun(t,x,d,pars),t0,x0,x0f,d0,d0f);
```

- decic is based on a simple Newton Method implementation

NEWTON METHOD

- Implicit solvers must iteratively solve for state value at next time step
- The newton's method is used for this purpose
- FACT: if the the function is continuous up to its second derivative, then there exists a nonempty neighbourhood around the root where Newton's method converges to the root
- If the root is simple, the Newton's method converges quadratically with a asymptotic error constant $|\frac{f''(x^*)}{2f'(x^*)}|$



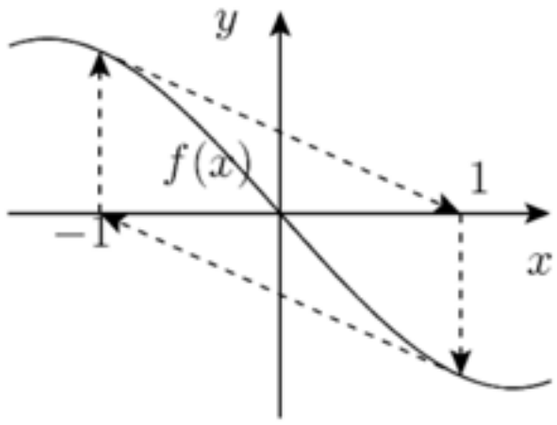
$$x_{n+1} = x_n - J^{-1} f(x_n)$$

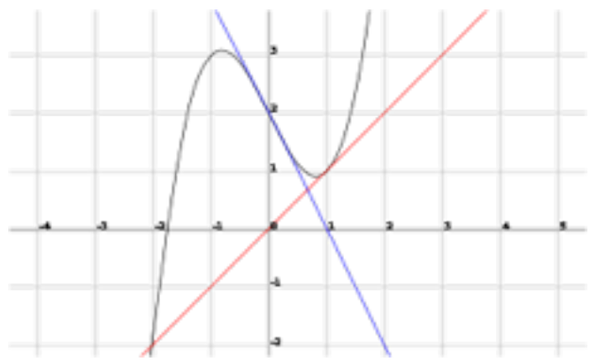
NEWTON METHOD FOR NON-SQUARE SYSTEMS

- Some system may be under determined, i.e., number of equations is less than the number of unknows
- In this case, the state that minimises the distance from the current state (or the initial guess) is sought
- For this the pseudo-inverse is used in solving for the Newton step

$$x_{n+1} = x_n - J^{\#} f\left(x_n\right)$$

NEWTON METHOD WITH TRUST REGIONS





- Newton method may not always converge
- A more robust set of root finding methods are known as the trust region methods

$$M(d) = f(x_n) + J(x_n)(d - x_n)$$

$$s_{n+1} = \underset{||d|| \leq \lambda_{n+1}}{\operatorname{argmin}} ||M(d)||^2$$

$$x_{n+1} = x_{n+1}, \text{ if } r_{n+1} = \frac{||f(x_n)||^2 - ||f(d)||^2}{||f(x_n)||^2 - ||M(d)||^2} \geq \tau_0$$

λ_{n+1} is increased if $r_{n+1} \geq \tau_1$, and decreased otherwise

MATLAB TRUST REGION METHODS

- The matlab root finding function fsolve implements various trust region methods

```
options = optimoptions('fsolve','Jacobian','on');

x = fsolve(@(x)myfun(x,par),x0,options);
```

```
function [r,J] = myfun(x,par)
Q1 = par(1);
Q2 = par(2);
r = zeros(2,1);
r(1) = x'*Q1*x;
r(2) = x'*Q2*x;

J=zeros(2);
J(1,:) = 2*x'*Q1;
J(2,:) = 2*x'*Q2;
```

DIFFERENTIAL EQUATIONS

- PROBLEM: obtain the continuous time trajectory between discrete states
- MATLAB has several solvers
    - ode45 - first try
    - ode15s - stiff systems
    - ode15i - implicit systems

```
sol = ode15i(@(t,x,d)odefun(t,x,d,pars),[t0 tf],x0,d0);
[xi,di] = deval(sol,ti);

function r = odefun(t,x,d,pars)
r = zeros(20,1);
r(1) = …
```

NUMERICAL INTEGRATION: FORWARD EULER METHOD

- Consider a simple ODE

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

- Approximate both sides as

$$\frac{x(t+\Delta t)-x(t)}{\Delta t} = f(t, x(t))$$

- Forward Euler method is an explicit solver
- Solution is found by marching forward in time

ORDER: FORWARD EULER METHOD

- Forward Euler method represents a discrete time system

$$x_{k+1} = x_k + \Delta t f(k, x_k)$$

- Linearize to obtain

$$x_{k+1} = (\Delta t A + I) x_k$$

- Spectrum

$$\lambda(\Delta t A + I) = \Delta t \lambda(A) + 1$$

- Consider a difference between a perturbed solution and the actual solution

$$x_{k+1} - \tilde{x}_{k+1} = (\Delta t A + I)(x_k - \tilde{x}_k)$$

$$x_{k+N} - \tilde{x}_{k+N} = (\Delta t A + I)^{N-1}(x_k - \tilde{x}_k)$$

- Error goes to zero if eigenvalues have norm less than 1
- For real eigenvalues

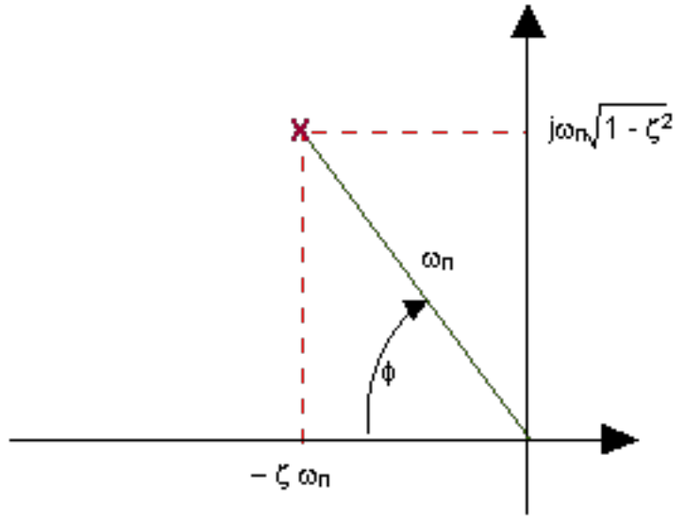
$$-1 < 1 + \Delta t \lambda(A) < 1$$

FORWARD EULER METHOD CONVERGENCE

- Consider again the forward Euler method where the spectrum is

$$\lambda(\Delta t A + I) = \Delta t \lambda(A) + 1$$

- Consider the second order system with system poles parametrised as



- Question: how does the value of wn affect stability of the Euler method?

NUMERICAL INTEGRATION: BACKWARD EULER METHOD

- Consider a simple ODE

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

- Approximate both sides as

$$\frac{x(t+\Delta t)-x(t)}{\Delta t} = f(t + \Delta t, x(t + \Delta t))$$

- Forward Euler method is an implicit solver
- Solution is found by solving for the updated state by looking for values satisfying the above equality

ORDER: BACKWARD EULER METHOD

- Backward Euler method represents a discrete time system

$$x_{k+1} = x_k + \Delta t f(k, x_k)$$

- Linearize to obtain

$$(I - \Delta t A)x_{k+1} = x_k$$

- Right hand side matrix is invertible for stable systems

$$x_{k+1} = (I - \Delta t A)^{-1} x_k$$

- Spectrum

$$\lambda(I - \Delta t A)^{-1} = (1 - \Delta t \lambda(A))^{-1}$$

- Consider a difference between a perturbed solution and the actual solution
- Error goes to zero if eigenvalues have norm less than 1
- For real eigenvalues

$$-1 < (1 - \Delta t \lambda(A))^{-1} < 1$$

NUMERICAL INTEGRATION: BDF (backward differential formula)

- Discretize in time
- Extrapolate using Legendre polynomials the last value using the previous m values

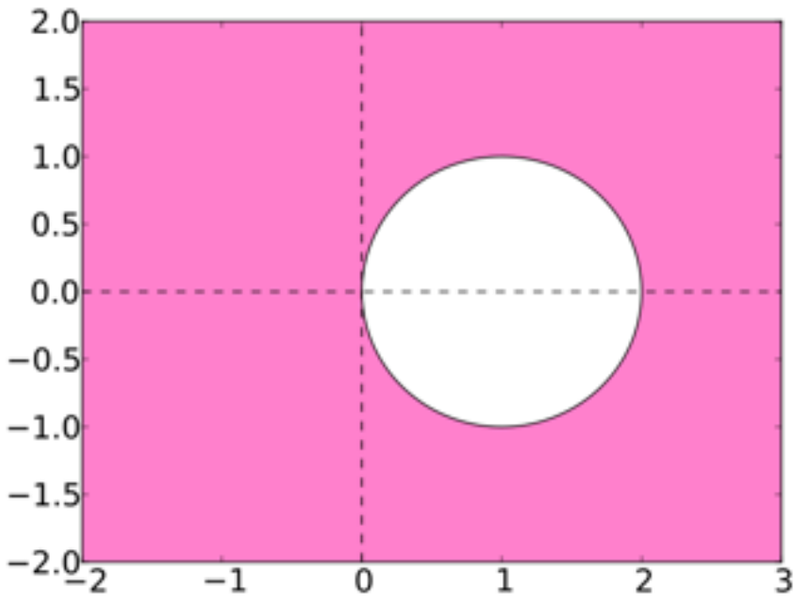
$$P(t) = \sum_{j=0}^{m} x_{i+1-j} L_j(t)$$

- Determine the missing polynomial coefficient by solving the nonlinear equation system

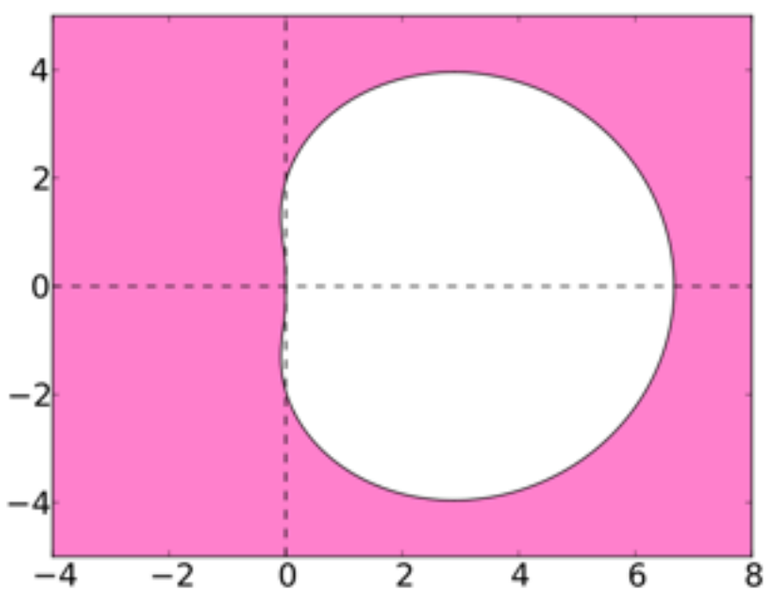
$$x_{i+1} = -\sum_{j=1}^{m} a_j x_{i+1-j} + b_m h f(t_{i+1}, x_{i+1}, z_{i+1})$$
$$0 = g(t_{i+1}, x_{i+1}, z_{i+1})$$

- mth order BDF is order m and can solve an m order DAE
- only m < 7 are stable!

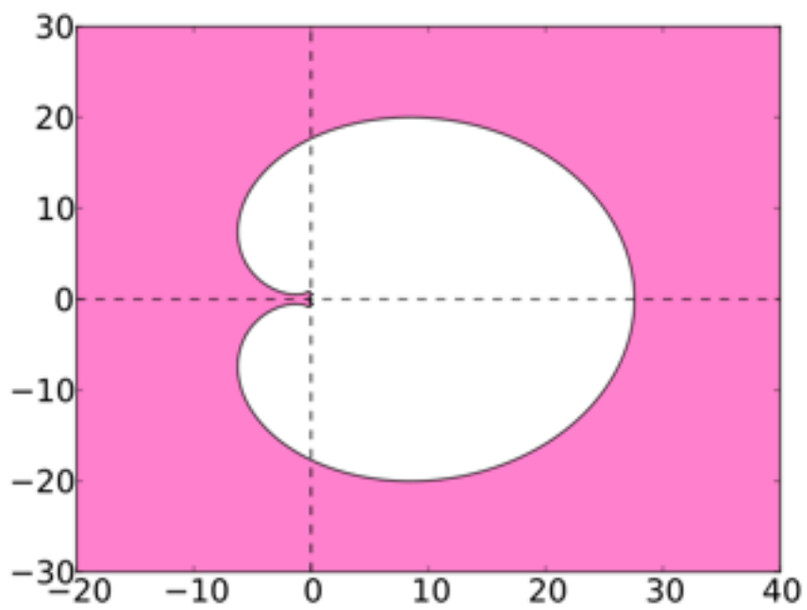
STABILITY: BDF



BDF: 1



BDF: 3



BDF: 6

SINGULAR PERTURBATION THEORY

- Also referred to as time scale separation, for the following reasons:
- Theory applies to systems that look like

$$\frac{dx}{dt} = f(x, y)$$

$$\epsilon \frac{dy}{dt} = g(x, y)$$

where epsilon is some small positive constant.

- Theory begins by setting epsilon to zero and solving for y

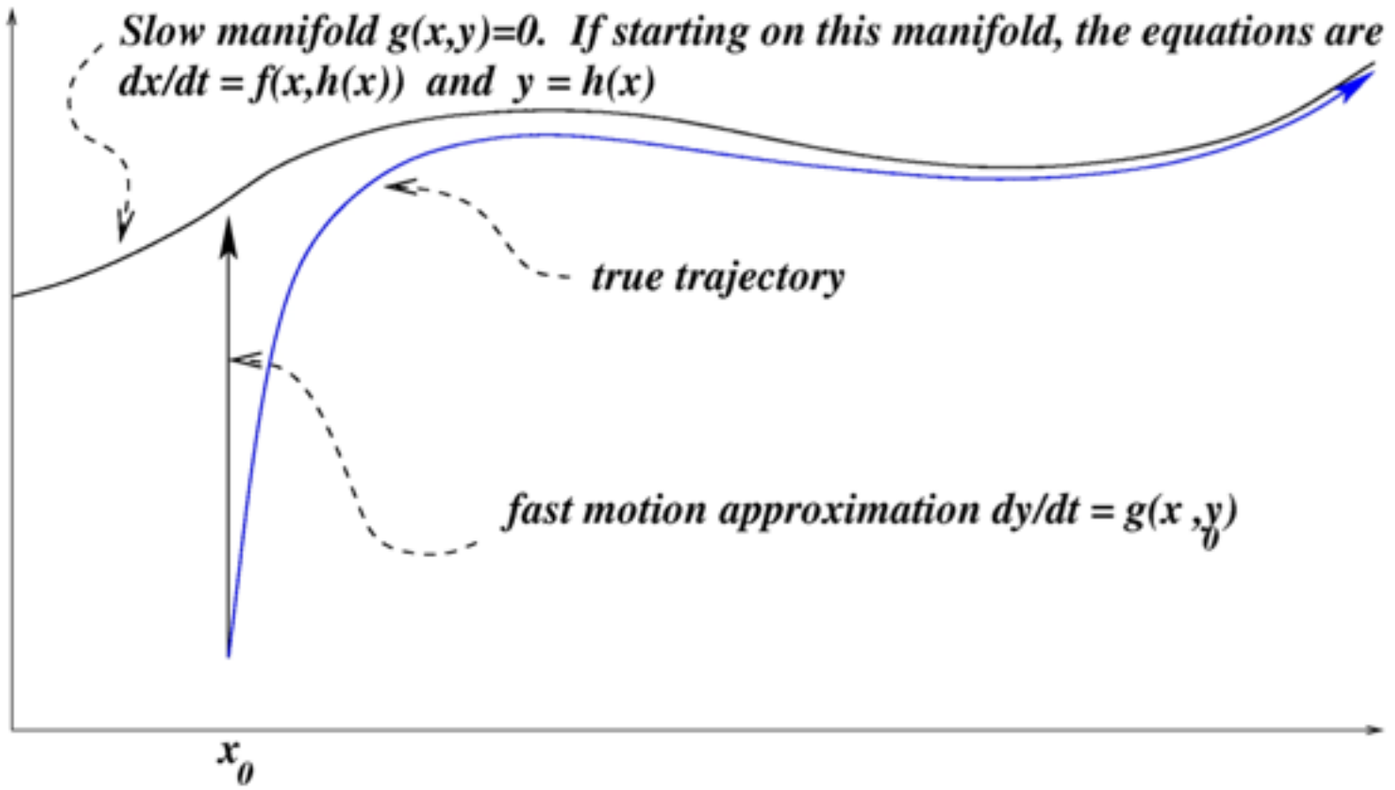
$$g(x, y) = 0 \rightarrow y = h(x)$$

- The reduced system has slow dynamics described by

$$\frac{dx}{dt} = f(x, h(x))$$

SINGULAR PERTURBATION THEORY

Eduardo D. Sontag, Lecture Notes on Mathematical Systems Biology



ODE SOLVER OPTIONS: TOLERANCES

- RelTol — This tolerance is a measure of the error relative to the size of each solution component. Roughly, it controls the number of correct digits in all solution components, except those smaller than thresholds AbsTol(i).The default, 1e-3, corresponds to 0.1% accuracy.

$$|1 - \frac{x_i}{x_{i-1}}| < relTol$$

- AbsTol — AbsTol(i) is a threshold below which the value of the ith solution component is unimportant. The absolute error tolerances determine the accuracy when the solution approaches zero.

$$|x(i) - x(i-1)| < absTol$$

```
options =
odeset('Jacobian',@(t,x,d)jfun(t,x,d,pars),'events',@(t,x,d)efun(t,x,d,pars));
```

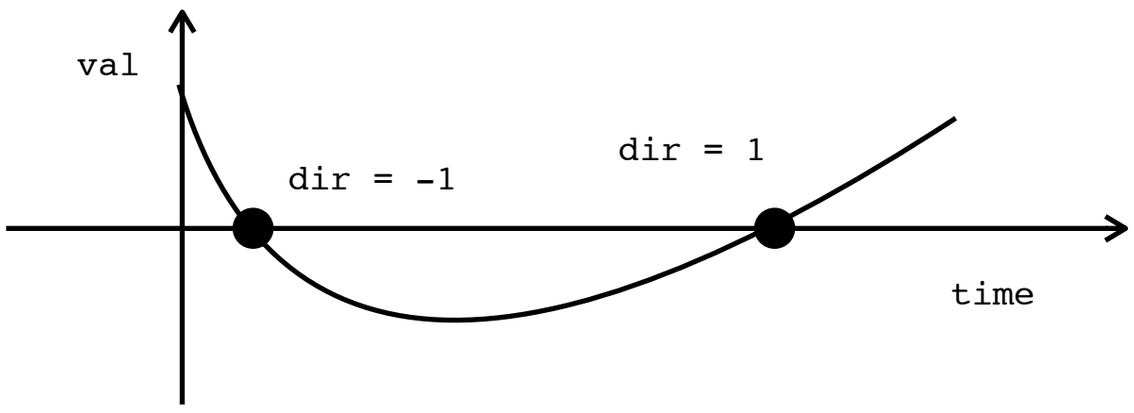
ODE SOLVER OPTIONS: JACOBIAN AND EVENTS

- By using numerical solver options, we can analytically define the Jacobian and define when discrete events occur

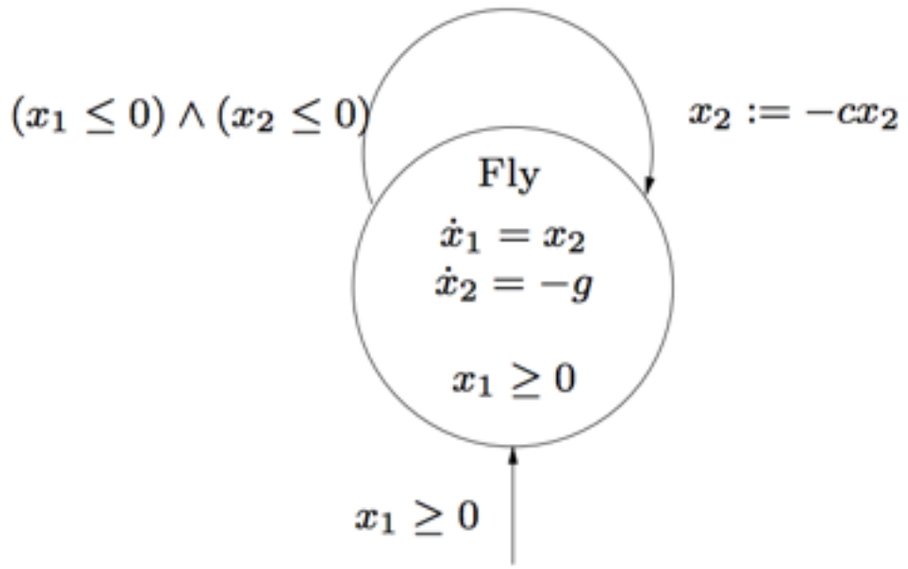
```
options = odeset('Jacobian',@(t,x,d)jfun(t,x,d,pars),'events',@(t,x,d)efun(t,x,d,pars));

function [Jx,Jd] = jfun(t,x,d,pars)
%function definition follows (will do this in practice)
```

$$Jx = \frac{\partial r}{\partial x} = \begin{pmatrix} (r_{1,1})_{x_1} & \cdots & (r_{1,1})_{x_{20}} \\ \vdots & \ddots & \\ (r_{6,5})_{x_1} & & \end{pmatrix} \qquad Jr = \frac{\partial r}{\partial d} = \begin{pmatrix} (r_{1,1})_{d_1} & \cdots & (r_{1,1})_{d_{20}} \\ \vdots & \ddots & \\ (r_{6,5})_{d_1} & & \end{pmatrix}$$

```
function [val,ter,dir] = efun(t,x,d,pars)
%function definition follows (will do this in practice)
```



BOUNCING BALL EXAMPLE




```
function [val,ter,dir] = efun(t,x,d,pars)

val = zeros(1); %initialize all variables, there is only a one event so not really required
ter = zeros(1);
dir = zeros(1);

val(1) = max([x(1),x(2)]);
dir(1) = -1;
ter(1) = 1; %stop simulation if val(1) is decreasing and val(1) = 0
```