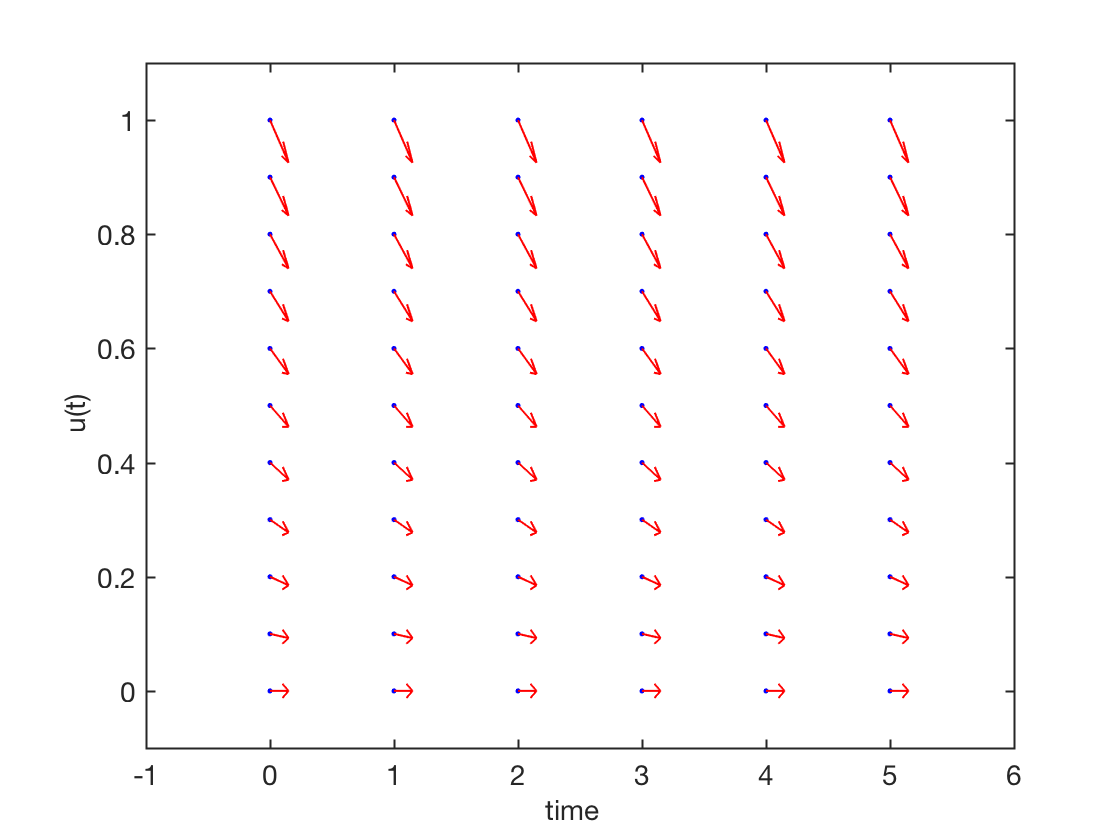
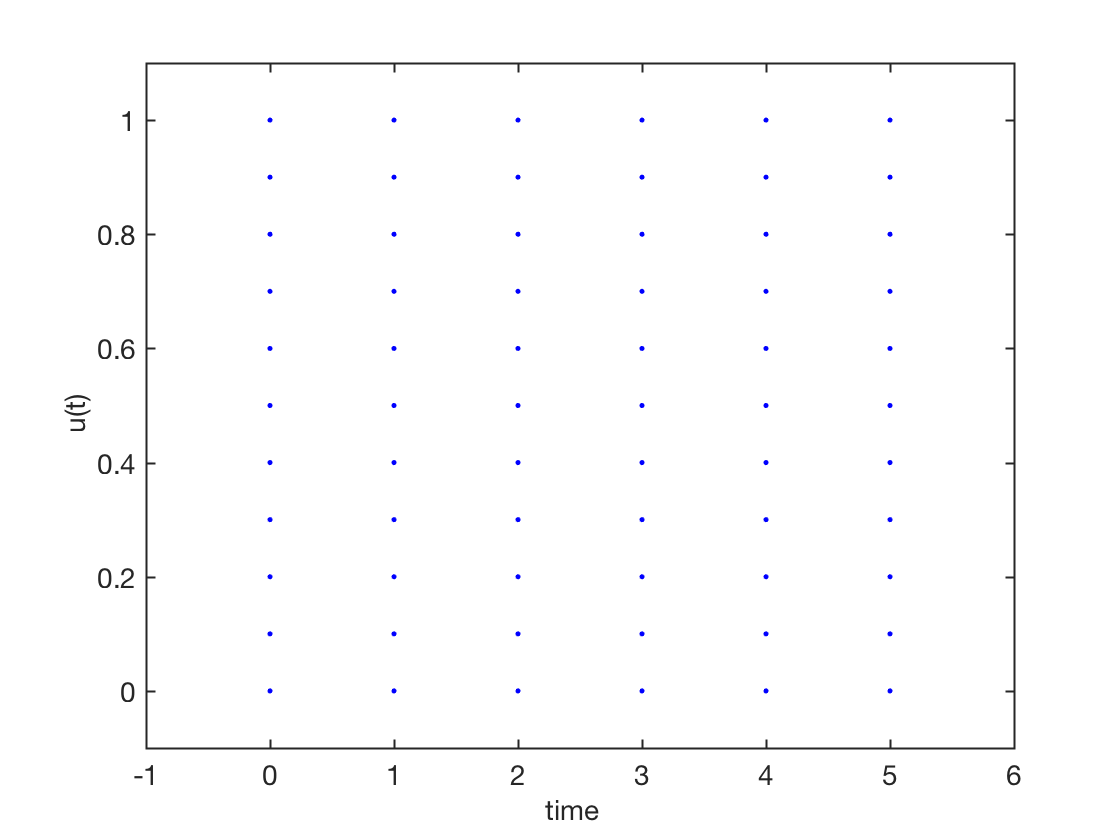
# **Simulation of chemical reactions.**

Consider a simple linear equation: . In this case, the solution for the unknown is a number: . Consider instead the equation:

The equation is a *1st order* *Ordinary Differential Equation* (ODE) because it involves an *ordinary 1st order* *derivative* (as opposed to a *partial derivative*) of some unknown function , which may or may notbe present in the right-hand side of the equation as a functionof two variables, (dependent) and *t* (independent)*.* In the particular case in which, the equation:

is also called *autonomous* because the independent variable *t* does not appear explicitly in the right-hand side. Thus, the solution to this ODE is *not a number* but a *function* that satisfies the equation for all values of *t* over some specified range. In general, the task is to find all the possible functions that satisfy the equation.



Before attempting to solve the equation analytically we can get a feel for the solution by plotting its *slope field* (also called *direction field*). To do so we choose a range of values for *t* (i.e. from 0 to 5, the *abscissa* axis), and given some *u*(*t*) values at *t =* 0 we repeat those values for all times. We can display the range of *t* and *u* values as a two-dimensional plot. This plot shows that, for each time *t*, different values of *u* are observed, depending on the underlying function *u*(*t*):

[t,u] = meshgrid(0:5,0:0.1:1)

figure;plot(t,u,'b.')

xlabel('time  ');ylabel('u(t)  ')

xlim([-1,6]);ylim([-0.1,1.1]);hold on

Next, we choose a value for *k* (i.e. -0.5), and for each value of *t*and *u*(*t*) in the chosen range we draw a little arrow with a slope equal to:

k = -0.5;

du = k.\*u

dt = ones(11,6);

quiver(t,u,dt,du,0.2,'r')

Now, it is also clear that for each time *t*, depending on the underlying functions , there exist different values of .

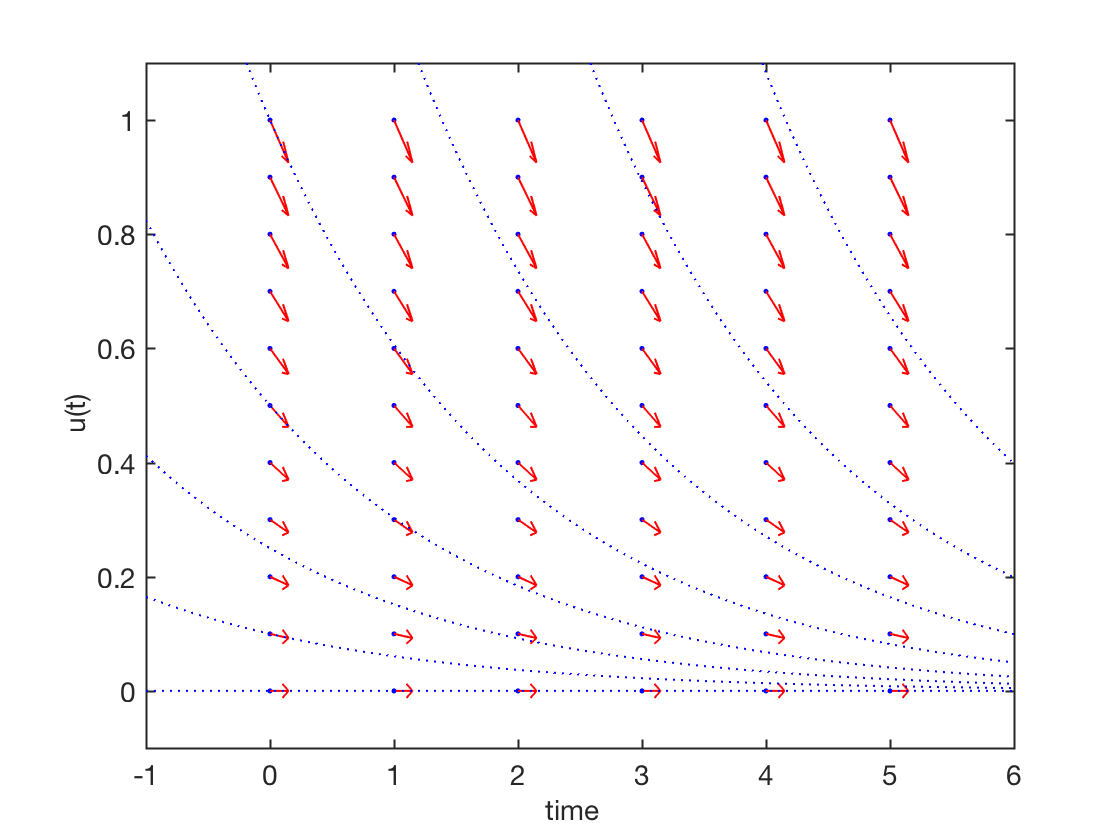
We can understand in what way these various functions differ by deriving an analytical solution for . To this end, we first *separate the dependent* (*u*) *and independent* (*t*) *variables* on opposite sides of the equation, and then *integrate* both sides:

in which we arbitrarily, but purposefully, chose as the constant of integration. Exponentiating both sides we finally obtain:

It's worth noting that the solution:

reduces to just *C* at *t = 0* because .

Thus, by chosing :



is the solution that starts from *u(0)* at *t = 0*.

For example, if we chose values of we will obtain 8 different functions .

hold on

t = [-1:0.1:6];

u0 = [0 0.1 0.25 0.5 1 2 4 8];

[t,u0] = meshgrid(t,u0)

u = u0.\*exp(k\*t)

plot(t(1,:),u',':b')

For each choice of *initial value* *u*(0) the solution curve is different, and there is no way to move one curve around to get another one. The initial value *u*(0) is the value for which the infinite solutions of the differential equation collapse into just one. In general, any *nth* order differential equation requires satisfying *n boundary conditions* in order to make the solution unique: for a 1st order ODE there is only *n*=1 *boundary condition*, and since the various functions *u*(t) are distinguished by the choice of the initial value *u*(0) as the boundary condition, the corresponding ODE is said to represent an *initial value problem.*

Now notice the similarity between our general *1st order, constant coefficient k* ordinary differential equation (ODE):

and the familiar differential equation describing the time evolution of a chemical reaction with *rate constant* *k* starting from the concentration of the chemical species u at time t = 0:

equation solution

This is a pair equation/solution that describes a chemical system with only 1 chemical species, but of course we could have a chemical system with *n* different species. In that case, we have a *system of differential equations*, which is known as the *Reaction Rate Equation* (RRE):

...

[u] now becomes a vector [**u**] of concentrations and *k* becomes a matrix ***K*** of coefficients:

for which we expect to find a solution vector ***ut*** of *n* elements.

In the very simple case in which all the chemical species in the system are acting independently the matrix ***K*** will be diagonal, but in the most common situation the chemical species will be connected by an equilibrium relationship and there will be more than one unknown in each equation. For example, consider the case of the simple chemical equilibrium:

The two differential equations describing this equilibrium are:

which we can write as a matrix equation:

In other words, at any point in time [u1] and [u2] are *coupled*, and instead of having a single equation:

now we have a matrix equation:

for which intuitively we expect the solution to be an exponential of the form:

although, at this point we do not know what is the meaning of the exponential .

Our approach to solving the matrix equation will be to use *similarity transformation* to place ***u*** in a space with basis ***S*** in which a matrix ***Λ*** *similar* to ***K*** is diagonal:

The rational for this solution is obviously that we are going to use as basis ***S*** the *eigenvectors* of ***K***. Thus, the *eigen decomposition* of ***K***:

gives us directly the central elements of the matrix equation above. First, we rearrange:

Then taking as the representation of [***u***] in the *eigenvector* basis ***S***, we can write:

This is now an *uncoupled* system of differential equations with uncoupled solutions:

which we can write in its own matrix form as:

This is the solution of our system of differential equations in ***S*** space, which now we can bring back in our standard Cartesian space:

and recalling that we finally obtain:

Comparing this result with the expected exponential solution of the form:

We now understand the meaning of, and how to calculate the exponential of the matrix . Given:

we derive that:

and likewise:

Again, it is important to stress here how the solution we found for the system of differential equations belongs to the general solutions for problems of the type:

***uk*** = ***Au0***

in which the *coupling* between the elements in vector ***u0*** exerted by a matrix ***A*** to produce vector ***uk*** is resolved by following a 3-step general strategy:

1. find the coordinates of ***u0*** in *eigenvector* space:

2. write ***u0*** as a linear combination of the *eigenvectors* of***A***:

3. multiply each *eigenvector* component by its *eigenvalue*:

in which case the solution is a combination of the '*pure*' solutions , which are the independent *normal modes* of the system.

Thus, if we represent the initial vector ***u****(0)* at time 0 as a linear combination of the *eigenvectors* of ***K***:

the ***u****(t)* vector at time *t* will be the same linear combination of the individually evolved *eigenvectors* at time *t* based on .

Viewed this way, the 3-step strategy to solve the system of differential equations

is:

1. find the coordinates of [**u**]**0** in the *eigenvector* space:

2. write [**u**]**0** as a linear combination of the *eigenvectors* of***K***:

3. multiply each *eigenvector* component by the exponent of the product of each *eigenvalue* by time:

Also in this case, the solution is a linear combination of the '*pure*' solutions : these are the independent *normal modes* of the evolving chemical system.

Let's apply what we have learned to a slightly more complicated equilibrium:

k1 = .8;

k2 = .4;

k\_1 = .3;

k\_2 = .5;

The system of differential equations describing this equilibrium is:

The *rate matrix* is:

row1 = [-k1 k\_1 0 ];

row2 = [k1 -(k\_1+k2) k\_2 ];

row3 = [0 k2 -(k\_2) ];

K = [row1;row2;row3]

the matrix equation is:

and the exact solution is:

For example, we can follow this reaction every 0.3 seconds (this is our stepsize) for 10 seconds starting from an initial value of the [***u***] vector:

Initial values

u\_vec\_0 = [100;0;0];

Stepsize

dt = 0.3;

Last time point

end\_time = 10;

Time vector

t\_vec = [0:dt:end\_time];

Eigen decomposition of the rate matrix

[S,D] = eig(K);

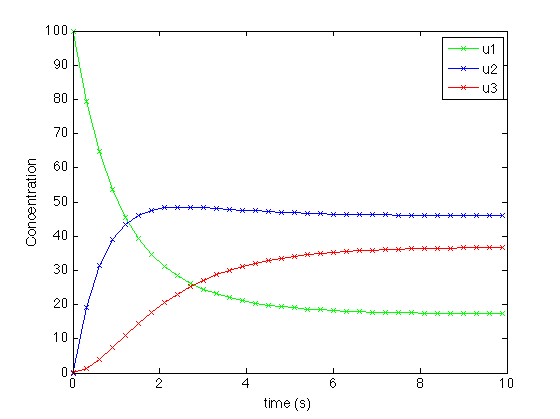
D = diag(D);

Here we set up a loop:

nsteps = length(t\_vec);

u\_vec = zeros(nsteps,3);

Coordinate of u\_vec\_0 in ***S*** space;

C = S\u\_vec\_0;

figure

xlim([0 10]);ylim([0 100]);

xlabel('time (s)');

ylabel('Concentration');

box on

hold on

for i = 1:nsteps

t = t\_vec(i);

u\_vec(i,:) = S\*diag(exp(D\*t))\*C;

Here we plot progressively

plot(t\_vec(1:i),u\_vec(1:i,1),'-xg',t\_vec(1:i),u\_vec(1:i,2),'-xb',t\_vec(1:i),u\_vec(1:i,3),'-xr')

drawnow expose; % pause(.05)

end

legend('u1','u2','u3')

We obtain the result shown on the side. Alternatively, we could represent directly the initial values as a linear combination of the eigenvectors:

Snm = S\*diag(C);

Notice how u\_vec\_0 is the sum of the vectors in ***Snm***:

sum(Snm,2)

figure;xlim([0 10]);ylim([0 100]);

xlabel('time (s)');ylabel('Concentration');box on;hold on

for i = 1:nsteps

t = t\_vec(i);

Here we sum up the normal modes:

u\_vec(i,:) = Snm\*exp(D\*t);

plot(t\_vec(1:i),u\_vec(1:i,1),'-og',t\_vec(1:i),u\_vec(1:i,2),'-ob',...

t\_vec(1:i),u\_vec(1:i,3),'-or')

drawnow expose; % pause(.1)

end

legend('u1','u2','u3')

Of course, we can also solve directly for with ***u***t = ***u***0e***K****t*, but in this case the eigen decomposition of ***K*** is carried out at each cycle of the loop in order to calculate the exponent (function *expm*), and thus the operation is much slower.

figure;xlim([0 10]);ylim([0 100]);

xlabel('time (s)');ylabel('Concentration');

box on;hold on

for i = 1:nsteps

t = t\_vec(i);

u\_vec(i,:) = expm(K\*t)\*u\_vec\_0;

plot(t\_vec(1:i),u\_vec(1:i,1),'-sg',t\_vec(1:i),u\_vec(1:i,2),'-sb',...

t\_vec(1:i),u\_vec(1:i,3),'-sr')

drawnow expose; % pause(.1)

end

legend('u1','u2','u3')

Clearly the chemical system reaches an equilibrium very rapidly. However, in other cases the equilibrium may be reached very slowly. Regardless of how fast the reaction is, we can calculate the exact value of the equilibrium concentration of all the chemical species in the system at infinite time from the *eigenvalues* of the *rate matrix*.

Since at any time *t* the solution is a linear combination of the *eigenvectors* with coefficients that contain the *eigenvalues* (multiplied by *t*)as exponents:

at infinite time the terms that contain negative *eigenvalues* tend to 0, and the term that contains the *eigenvalue* = 0 remains unchanged as *.* Therefore, provides the concentrations of the three chemical species at equilibrium.

Equilibrium at infinite time (steady state):

ss = C(3)\*S(:,3)

We would obtain the same steady state also if we started from a different initial concentration of the chemical species as long as the sum of all the concentrations is the same.

u\_vec\_0 = [30;20;50];

Coordinate of u\_vec\_0 in ***S*** space;

C = S\u\_vec\_0;

Steady\_state at infinite time:

ss = C(3)\*S(:,3)

We can check that the equilibrium is what we expected from the equilibrium constants:

k1/k\_1

ss(2)/ss(1)

k2/k\_2

ss(3)/ss(2)

A corollary of this result, valid for any chemical reaction, is that if one of the *eigenvalues* of the rate matrix is 0 it means the reaction leads to a chemical equilibrium.

In another example, we consider a more complex equilibrium:

k1 k2 k3

U1⇔U2⇔U3⇔U4

k-1 k-2 k-3

k1 = 3; k2 = 7; k3 = 3; k\_1 = 5; k\_2 = 1; k\_3 = 4;

u\_vec\_0 = [0.5 0.2 0.6 0.7]';

The system of differential equations is:

dU1/dt = -k1U1 + (k-1)U2 + 0\*U3 + 0\*U4

dU2/dt = k1U1 [-(k-1) -(k2)]U2 + (k-2)U3 +0\*U4

dU3/dt = 0\*U1 + k2U2 [-(k-2) -k3]U3 +(k-3)U4

dU4/dt = 0\*U1 + 0\*U2 + k3U3 -(k-3)U4

The rate matrix ***K*** is:

row1 = [-k1 k\_1 0 0 ];

row2 = [k1 -(k\_1+k2) k\_2 0 ];

row3 = [0 k2 -(k\_2+k3) k\_3];

row4 = [0 0 k3 -k\_3];

K = [row1;row2;row3;row4]

[S,D] = eig(K);

We solve the system of linear equations using the 'backslash' operator. Again, C is the representation of the initial value vector in the eigenvector basis. We can check the 4 components sum up to the ***u***(0) vector. At infinite time the components corresponding to the 3 negative eigenvalues vanish and the remaining component determines the steady-state:

C = S\u\_vec\_0

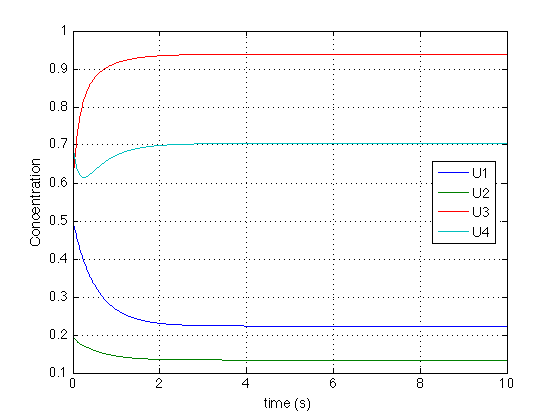
S\*C

steady\_state = S(:,4)\*C(4)

The reaction time course is derived as usual.

dt = 0.05; end\_time = 5;

D = diag(D);

time\_vec = (0:dt:end\_time);

nsteps = length(time\_vec);

u\_vec = zeros(nsteps,4);

for i = 1:nsteps

t = time\_vec(i);

u\_vec(i,:) = S\*diag(exp(D\*t))\*C;

end

figure;plot(time\_vec,u\_vec)

legend('U1','U2','U3','U4','Location','best')

xlabel('time (s)');

ylabel('Concentration');

grid on

Since the differential equation gives us directly the rate of change of the vector ***u*** with respect to time, we can also find a numerical solution for the function ***u***(t) by calculating directly the Δ***u*** that we need to add to the valueof ***u***at time *t* to obtain the value of ***u*** at time *t*+Δ*t*. The numerical evaluation of the function ***u***(t) is therefore carried out in sequential steps rather than individual independent steps. In this method, we replace:

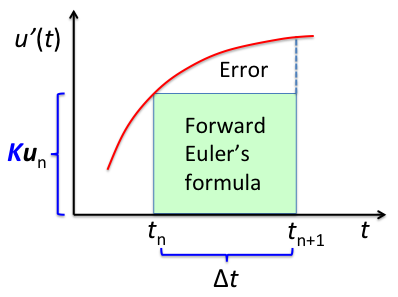
the differential equation:

with a difference equation:

Since changes in the Δ*t* interval as goes from to , the accuracy of the difference equation:

depends on the time (within the interval) chosen to evaluate the *slope* in the product .

We first describe the formulation known as the *Forward Euler Method*. In this method, the slope is evaluated at the beginning of the Δ*t* interval: thus *slope* = ***Ku***(n), where *n* is the iteration number. The value of the *integral* is the area contained in the light green rectangle and the *error* is the area between the rectangle and the curve:



***u***(n+1) - ***u***(n) = Δ*t****Ku***(n)

***u***(n+1) = Δ*t****Ku***(n) + ***u***(n)

***u***(n+1) = (Δ*t****K*** + ***I***) ***u***(n)

***G*** = Δ*t****K*** + ***I***

from which we derive the general expression:

***u***(n+1) = ***G u***(n)

***u***(n) = ***Gn u***(0)

This formulation is usually defined as an 'explicit method' because ***u***(n+1) appears *only on the left side*. First, we build the growth matrix ***G***:

G = dt\*K + eye(4);

Then we 'grow' the initial ***u*** vector by repeated applications of the 'growth' matrix:

u\_vec = zeros(4,nsteps);

u\_vec(:,1) = u\_vec\_0;

for i = 1:nsteps-1

u\_vec(:,i+1) = G\*u\_vec(:,i);

end

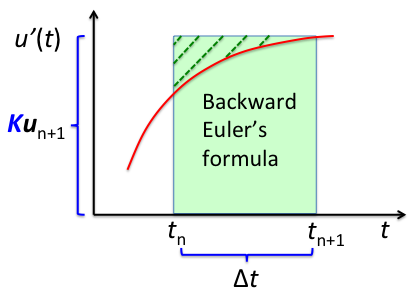
figure;plot(time\_vec,u\_vec)

legend('U1','U2','U3','U4','Location','best')

grid on

The *Forward Euler* method provides 1st order accuracy . This means the anticipated error is the same order of magnitude as the chosen stepsize Δ*t*, and therefore a very small Δ*t* must be taken to ensure the result is as correct as possible.

Another formulation is the *Backward Euler Method*. In this method, the slope is evaluated at the end of the Δ*t* interval: thus *slope* = ***Ku***(n+1), where *n* is the iteration number. The value of the *integral* is the area contained in the light green rectangle and the *error* is the hatched area of the rectangle above the curve:

***u***(n+1) - ***u***(n) = Δ*t****K*** ***u***(n+1)

***u***(n+1) - (Δ*t****K***) ***u***(n+1) = ***u***(n)

(***I*** - Δ*t****K***) ***u***(n+1) = ***u***(n)

***u***(n+1) = inv(***I*** - Δ*t****K***) ***u***(n)

***G*** = inv(***I*** - Δ*t****K***)

Thus, only the growth matrix ***G*** changes:

G = inv(eye(4) - dt\*K);

while the general expression remains unchanged:

***u***(n+1) = ***Gu***(n)

***u***(n) = ***Gnu***(0)

*Backward Euler* is defined as an 'implicit method' because ***u***(n+1) appears *on the both sides* of the difference equation. Like *forward Euler* it provides only 1st order accuracy , and therefore a very small Δ*t* must be taken to ensure that the discrete solution is a good approximation of the continuous solution of .

It is important to consider the conceptual difference between the *explicit* and the *implicit* Euler methods. Let's look at the two *growth* matrices:

***Gexpl*** = Δ*t****K*** + ***I Gimpl*** = (***I*** - Δ*t****K***)-1

We recall here that adding or subtracting the identity matrix ***I*** to the matrix Δ*t****K*** increases or decreases its *eigenvalues* by 1 without changing its *eigenvectors*. Thus, for the purpose of understanding what happens to the solution when the value chosen for Δ*t* increases we can just consider any of the *decoupled* components ***Gd\_*** obtained by diagonalizing ***K*** to ***D***.

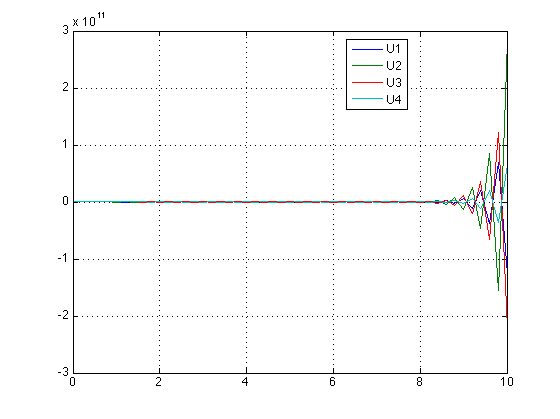
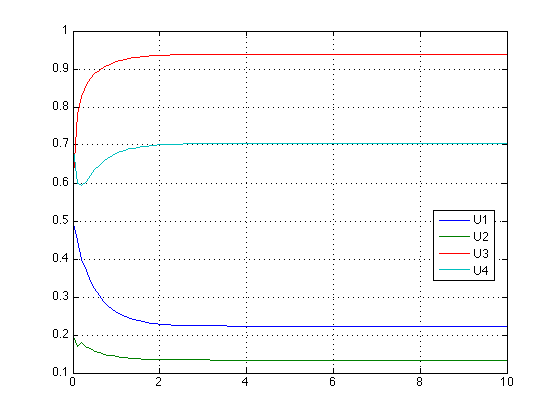
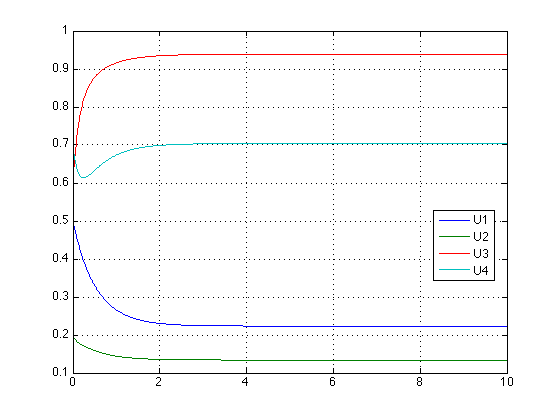
***Gdi\_expl*** = Δ*t****Di*** + 1 ***Gdi\_impl*** = (1 - Δ*t****Di***)-1

Since the *eigenvalues* ***D*** *of* ***K*** are all negative with the exception of one (which is 0, and thus provides the steady state solution), as the stepsize Δ*t* becomes larger the value of Δ*t****Di*** becomes progressively more negative until, when Δ*t****Di*** < -2, ***Gdi\_expl*** = (Δ*t****Di*** + 1) < -1 and its powers, required for the solution ***u***(n) = ***Gdi\_expln u***(0), grow exponentially while alternating in sign: as a result the numerical solution of the *difference equation* explodes. In contrast, as Δ*t* grows and Δ*t****Di*** becomes more negative, 0 < (1 - Δ*t****Di***)-1 < 1, and the powers of ***Gdi\_impl***, required for the solution ***u***(n) = ***Gdi\_impln u***(0), become smaller.

For this reason, *backward Euler* (implicit) is *unconditionally stable*, while *forward Euler* (explicit) is *conditionally stable* with the stability condition being that Δ*t****Di*** be> -2 for any of the *eigenvalues* of ***K***. For example, in the equilibrium considered above the threshold for stability is at a Δ*t* ~ 0.15 s. This effect can be seen clearly by calculating the evolution of the reaction with both *forward and backward Euler* for different Δ*t* from 0.01 to 0.2.

*forward Euler*

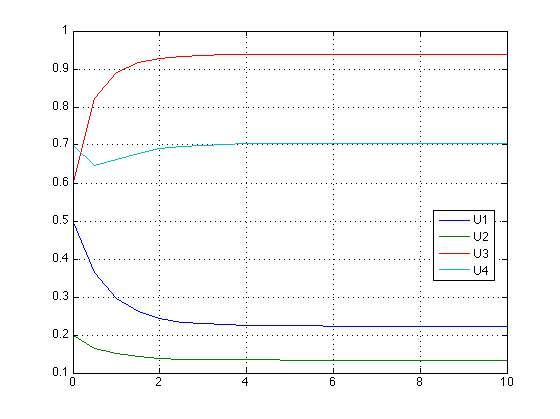
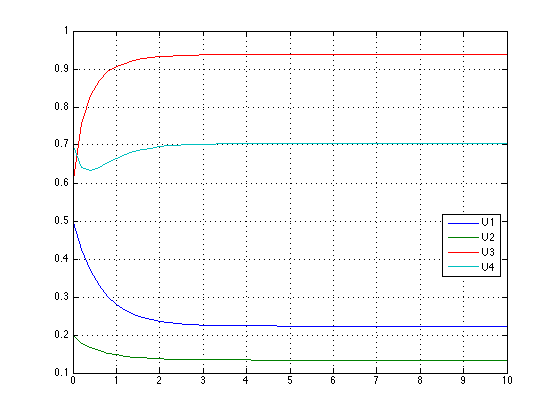
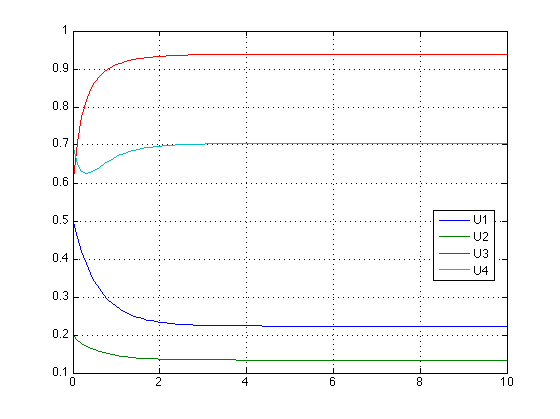
Δ*t* = 0.01s Δ*t* = 0.1s Δ*t* = 0.2s



Notice the exponent in the ordinate axis when Δ*t* = 0.2s: the solution explodes with oscillating positive and negative values. In contrast, the solution remains *stable*, although progressively *less accurate*, when we use an implicit calculation even if we use even larger stepsizes:

*backward Euler*

Δ*t* = 0.1s Δ*t* = 0.2s Δ*t* = 0.5s



Before proceeding further, it is important to understand why Euler methods have accuracy. Euler forward difference equation is:

***u***(n+1) = ***u***(n) + Δ*t****Ku***(n) = ***u***(n) + Δ*t****u***(n)'

The true solution from *Taylor expansion* is:

***u***(n+1) = ***u***(n) + Δ*t****u***(n)' + 1/2 Δ*t*2 ***u***(n)'' + On(Δ*t* n>2)

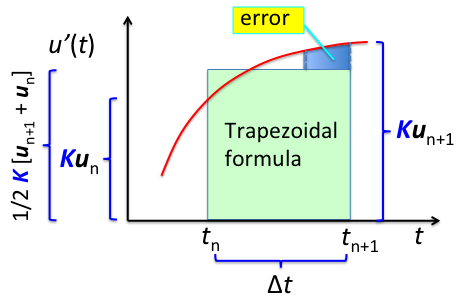
Thus, at each step Euler misses the 2nd + higher order terms: altogether these terms represent the local *discretization error* DE of each step. For Euler this error is ~1/2 Δ*t*2 ***u***(n)'', which can be considered ≤ some constant 0 < *c* < 1times Δ*t*2, or equivalently . At each step this error accumulates and at the end of *N* steps the total error |eN| contains *N* steps, each ≤ *c*Δ*t*2. Since *N* is the total time *T/*Δ*t* we derive:

DE ≈ *c*Δ*t***p+1**

|eN| ≤ *Nc*Δ*t***p+1** = *T/*Δ*t c*Δ*t***p+1** = *cT*Δ*t* **p**

Every method has its own values of *p and c* (1 and 1/2 respectively in Euler).

The local discretization error DE tells us about the accuracy of a method. Euler is only 1st order accuracy because the 1st term missed in the Taylor series is 1/2 Δ*t*2 ***u***''. Better methods capture this term, but may miss other higher order terms.

The first of these methods is the *Trapezoidal Method*, known as Heun or Crank-Nicolson, which provides 2nd order accuracy . In this method, the slope ***Ku*** is evaluated as the mean of the slope at the beginning and the slope at the end of the Δ*t* interval: the value of the *integral* is the mean of the areas calculated by the forward and backward Euler methods. The new difference equation is the following:

***u***(n+1) - ***u***(n) = Δ*t*1/2 [***Ku***(n+1) + ***Ku***(n)]

Therefore, this is an *implicit* method because ***u***(n+1) appears on both the left and the right side:

***u***(n+1) - ***u***(n) = Δ*t*/2 [***Ku***(n+1) + ***Ku***(n)]

***u***(n+1) - ***u***(n) = Δ*t*/2 ***Ku***(n+1) + Δ*t*/2 ***Ku***(n)

***u***(n+1) - Δ*t*/2 ***Ku***(n+1) = Δ*t*/2 ***Ku***(n) + ***u***(n)

(***I*** - ***K***Δ*t*/2) ***u***(n+1) = (***I*** + ***K***Δ*t*/2) ***u***(n)

***u*** (n+1) = inv(***I*** - ***K***Δ*t*/2) (***I*** + ***K***Δ*t*/2) ***u***(n)

***G*** = inv(***I*** - ***K***Δ*t*/2) (***I*** + ***K***Δ*t*/2)

***u***(n+1) = ***Gu***(n)

then the growth matrix has the form:

dt = 0.05;

end\_time = 5; % last time point in seconds

time\_vec = [0:dt:end\_time];

nsteps = length(time\_vec);

I = eye(4);

G = (I - K\*dt/2)\(I + K\*dt/2);

u\_vec = zeros(4,nsteps);

u\_vec(:,1) = u\_vec\_0;

for i = 1:nsteps-1

u\_vec(:,i+1) = G\*u\_vec(:,i);

end

figure;plot(time\_vec,u\_vec)

legend('U1','U2','U3','U4','Location','best')

grid on

Notice how, with any particular choice of Δ*t* at infinite time the ***G*** matrix is associated with only one eigenvector, as the power of all eigenvalues <1 progressively vanishes:

[S,D] = eig(G)

[~,l\_max] = max(diag(D))

l\_ind = true(4,1);

l\_ind(l\_max) = false;

D(l\_ind,l\_ind) = 0;

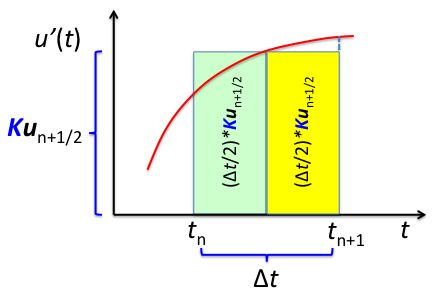
Ginf = S\*D/S

u\_vec\_inf = Ginf\*u\_vec\_0

For the *Trapezoidal Method* we have:

DE ≈ *c*Δ*t***2+1**

|eN| ≤ *N c*Δ*t***3** = *cT*Δt**2**

It is also possible to increase the accuracy by keeping the method *explicit*. Conceptually, the trapezoidal method uses Euler to evaluates the slope at the beginning and end of the entire interval between *t*(n) and *t*(n+1), and then averages the two slopes to take the actual step. Another possibility, corresponding to the *Midpoint Method*, would be to use Euler to step only halfway across the interval, evaluate the slope at this intermediate point, and then use that slope to take the actual step.

The following are the algebraic steps leading to the formulation of the growth matrix:

***u***(n+1) - ***u***(n) = Δ*t****Ku***(n+1/2)

***u***(n+1) = ***u***(n) + Δ*t****Ku***(n+1/2)

***u***(n+1) = ***u***(n) + Δ*t****K***[***u***(n) + Δ*t/*2 ***Ku***(n)]

***u***(n+1) = ***u***(n) + Δ*t****Ku***(n) + Δ*t****K***(Δ*t/*2)***Ku***(n)

***u***(n+1) = ***u***(n) + Δ*t****Ku***(n) + Δ*t2/*2***KKu***(n))

***u***(n+1) = (***I*** *+*Δ*t****K*** *+*Δ*t2/*2***KK***)***u***(n)

***G*** = (***I*** *+*Δ*t****K*** *+*Δ*t2/*2***KK***)

***u***(n+1) = ***Gu***(n)

dt = 0.05;

end\_time = 5;

time\_vec = [0:dt:end\_time];

nsteps = length(time\_vec);

I = eye(4);

G = (I + K\*dt + K\*K\*dt\*dt/2);

u\_vec = zeros(4,nsteps);

u\_vec(:,1) = u\_vec\_0;

for i = 1:nsteps-1

u\_vec(:,i+1) = G\*u\_vec(:,i);

end

figure;plot(time\_vec,u\_vec)

legend('U1','U2','U3','U4','Location','best')

ylim([0 1]); grid on

Like the trapezoidal method the midpoint method is .

If we were to use both the *Midpoint* and the *Trapezoidal* methods simultaneously, they would produce two slightly different values for ***u***(n+1). The difference between the two values can be used to provide an error estimate and a basis for picking an optimal step size. Furthermore, a combination of the two values would be more accurate than either one individually.

This is the idea behind *single-step methods* for integrating ordinary differential equations (ODEs). The function is evaluated *i* times for values of *ti* between *t*(n)and *t*(*n*+1) and intermediate values of ***u***are obtained by adding linear combinations of the values of Δ*tfi* to ***u***(n). Single-step methods are often called *Runge–Kutta methods*, after the two German applied mathematicians who first wrote about them around 1905. The classical Runge–Kutta method uses four function evaluations per step:

***u***(n+1) = Δ*t*/6 [***K1*** *+**2****K2*** *+*2***K3*** + ***K4***]***u***(n) + ***u***(n)

where is :

With some rearrangements, the 4th order accuracy *Runge-Kutta* (*RK4*) method can be coded in the standard form employing a growth matrix ***G***:

***u***(n+1) = (***I*** + Δ*t*/6 [***K1*** *+**2****K2*** *+*2***K3*** + ***K4***])***u***(n)

***G*** = (***I*** + Δ*t*/6 [***K1*** *+**2****K2*** *+*2***K3*** + ***K4***])

***u***(n+1) = ***Gu***(n)

K1 = K;

P = I + K\*dt/2;

K2 = K\*P;

Q = I + K\*P\*dt/2;

K3 = K\*Q;

R = I + K\*Q\*dt;

K4 = K\*R;

G = I + dt\*(K1 + 2\*K2 + 2\*K3 + K4)/6;

u\_vec = zeros(4,nsteps);

u\_vec(:,1) = u\_vec\_0;

for i = 1:nsteps-1

u\_vec(:,i+1) = G\*u\_vec(:,i);

end

figure;plot(time\_vec,u\_vec)

legend('U1','U2','U3','U4','Location','best')

grid on

Fourth order *Runge–Kutta* (*RK4*) is the basis of ODE45, MATLAB workhorse method for solving ODE's with 4th-5th order accuracy, . ODE45 is properly an *adaptive RK method* using both 4th and 5th order integration formulas to estimate the discretization error in the formula of 4th order at each integration step as , and automatically adjusting the stepsize to keep the error within prescribed limits.

However, although optimized, a constant stepsize may not be appropriate for the entire range of integration: for example, chemical reactions typically start off with rapid changes before becoming smooth. This behaviour is usually described as '*stiffness*'. Dictionary definitions of the word “stiff” involve terms like “not easily bent,” “rigid,” and “stubborn.” We are concerned with a computational version of these properties.

*A problem is stiff if some parts of the solution being sought vary slowly, but other parts vary rapidly, so that the numerical method must take* *small steps to obtain satisfactory results.*

Stiffness is an efficiency issue. If we weren’t concerned with how much time a computation takes, we wouldn’t be concerned about stiffness. Non-stiff methods can solve stiff problems; they just take a long time to do it. This means that using a *non-stiff* solver to compute a chemical reaction that is very fast at the beginning and very slow at the end would force us to use a very small stepsize, which would provide the required accuracy for the initial part of the reaction, but would slow down very much the computation of the final part.

Several ODE solvers are available in MATLAB for both non-stiff and stiff problems. For example, notice how the stepsize would be handled quite differently by ODE45 (high accuracy, non-stiff solver), ODE23s (medium accuracy, stiff solver), and ODE15s (high accuracy, stiff solver):

Fp = @(t,u) K\*u

tspan = [0 10]

[time\_vec,u\_vec] = ode45(Fp,tspan,u\_vec\_0);

figure;plot(time\_vec,u\_vec,'-x');legend('U1','U2','U3','U4','Location','best');

title('ODE45');grid on

[time\_vec,u\_vec] = ode15s(Fp,tspan,u\_vec\_0);

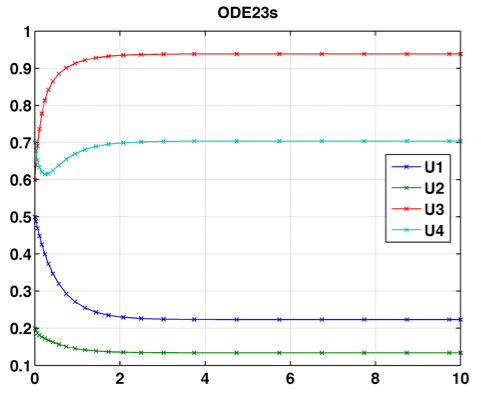
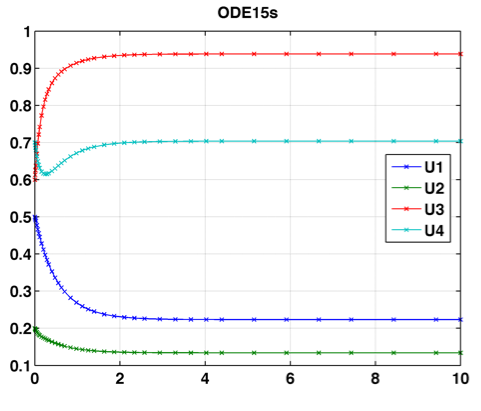
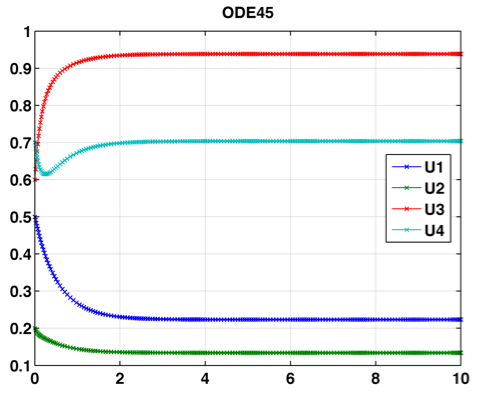
figure;plot(time\_vec,u\_vec,'-x');legend('U1','U2','U3','U4','Location','best');

title('ODE15s');grid on

[time\_vec,u\_vec] = ode23s(Fp,tspan,u\_vec\_0);

figure;plot(time\_vec,u\_vec,'-x');legend('U1','U2','U3','U4','Location','best');

title('ODE23s');grid on



The chemical ensembles we have considered so far is represented by a system of differential equations that are 1st order (only 1st derivatives are present) and constant coefficients (the coefficient matrix ***K***, the rate matrix, does not change in time). However, in the more general case, the system of differential equations does not have constant coefficients because some of the reactions are bimolecular. In this case, the rate constants are 2nd order (1/(time\*concentration)), which means they are the product of a 1st order rate constant (1/time) and the reciprocal of a concentration (1/M). As an example, let's consider an equilibrium involving a bimolecular reaction:

k1 k2

U1 + U2 ⇔ U3 ⇔ U4

k\_1 k\_2

This is 'almost' what an enzymatic reaction would look like, with U1 being the substrate, U2 the enzyme, U3 the ES complex and U4 the product. However, notice that this is not really an enzymatic reaction because when U3 is converted to U4 we are not regenerating U2.

k1 = 5;

k\_1 = 2;

k2 = 7;

k\_2 = 1;

The following are the differential equations:

dU1/dt = -k1\*U1\*U2 + 0\*U2 + (k\_1)\*U3 + 0\*U4

dU2/dt = 0\*U1 -k1\*U1\*U2 + (k\_1)\*U3 + 0\*U4

dU3/dt = k1\*U1\*U2 + 0\*U2 -(k\_1+k2)\*U3 + k\_2\*U4

dU4/dt = 0\*U1 + 0\*U2 + k2\*U3 -k\_2\*U4

We initialize the matrix assigning some initial concentrations to the species.

Uo = [2.0 2.0 0.0 0.0]';

U1 = Uo(1);

U2 = Uo(2);

row1 = [-k1\*U2 0 k\_1 0 ];

row2 = [0 -k1\*U1 k\_1 0 ];

row3 = [k1\*U2 0 -(k\_1+k2) k\_2];

row4 = [0 0 k2 -k\_2];

K = [row1;row2;row3;row4];

This system of differential equations does not have constant coefficients! We see that now the rate matrix ***K*** contains terms (U1 and U2) that will be changing as the reaction goes on. This system of differential equations is called *non-autonomous* (the independent variable *t* is present also on the right side of each equation:

as opposed to the systems of differential equations with constant coefficients or *autonomous*, in which the independent variable *t* is present only on the left side of each equation as:

In an autonomous system, once the initial condition is set, the state of the system depends only on the elapsed time *t−t0* since starting and not directly on the current time *t*. In contrast, the state of the non-autonomous system depends separately on the actual time *t* and the starting time *t0*.

Therefore, we cannot use a single rate matrix, but we have to update the matrix at reasonably small intervals along the reaction. The smaller our choice of intervals (the 'time step') the more accurate will be the reconstruction of the reaction progress, but also the more slowly our simulation will go.

time\_step = 0.01; time\_vec = [0:time\_step:1];

nsteps = length(time\_vec);

U = zeros(4,nsteps); U(:,1) = Uo;

for i = 2:nsteps

K = [row1;row2;row3;row4];

[S,D] = eig(K);

D = diag(D);

Dt = D\*time\_step;

eDt = diag(exp(Dt));

U(:,i) = (S\*eDt/S)\*U(:,i-1);

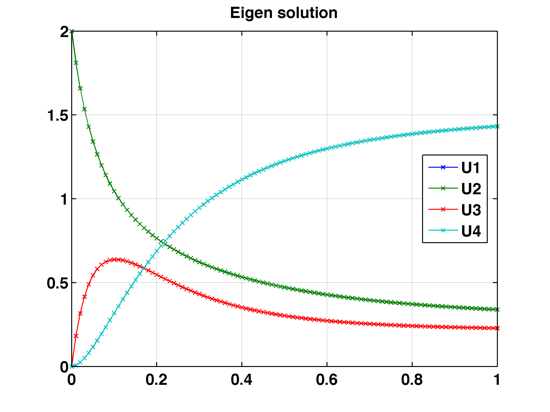
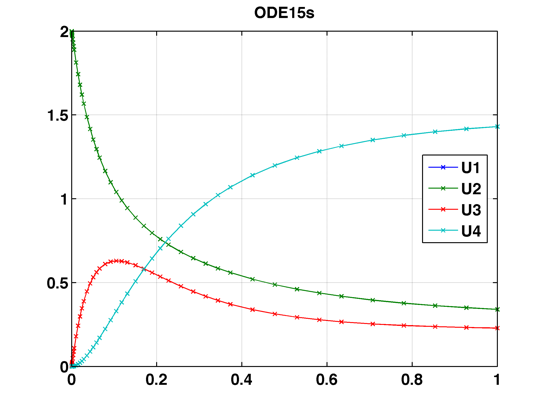
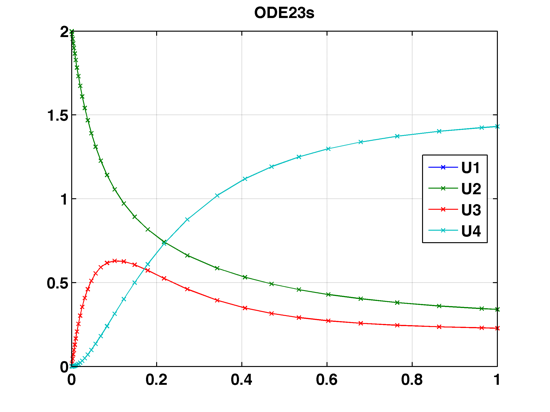
U1 = U(1,i);

U2 = U(2,i);

row1 = [-k1\*U2 0 k\_1 0 ];

row2 = [0 -k1\*U1 k\_1 0 ];

row3 = [k1\*U2 0 -(k\_1+k2) k\_2];



end

figure;plot(time\_vec,U,'-x')

legend('U1','U2','U3','U4','Location','best')

title('Eigen solution');grid on

We can see how this small loop implements a very simple solver of chemical reactions. Much more complex reactions can be represented just as well, the only thing requiring a change is the rate matrix ***K***, which is different in each case.

In order to use MATLAB solvers with equilibria involving 2nd order reactions we need to create an *ad hoc* function for the derivative calculation. Notice that in order to be used by the solver this must be defined as a function of *t* despite the fact the time is not used inside the function:

function du = Fp(t,Uo)

k1 = 5;

k\_1 = 2;

k2 = 7;

k\_2 = 1;

U1 = Uo(1);

U2 = Uo(2);

row1 = [-k1\*U2 0 k\_1 0 ];

row2 = [0 -k1\*U1 k\_1 0 ];

row3 = [k1\*U2 0 -(k\_1+k2) k\_2];

row4 = [0 0 k2 -k\_2];

K = [row1;row2;row3;row4];

du = K\*Uo;

end

After saving the function as 'Fp.m' we can use the ODE solvers and compare the solutions:

tspan = [0 1];

[time\_vec,u\_vec] = ode23s(@Fp,tspan,Uo);

figure;plot(time\_vec,u\_vec,'-x');legend('U1','U2','U3','U4','Location','best');

title('ODE23s');grid on

[time\_vec,u\_vec] = ode15s(@Fp,tspan,Uo);

figure;plot(time\_vec,u\_vec,'-x');legend('U1','U2','U3','U4','Location','best');

title('ODE15s');grid on

Finally, we can simulate a real enzymatic reaction with a *Briggs-Haldane* model:

k1 k2

U1 + U2 ⇔ U3 ⇔ U4 + U2

k\_1 k\_2

where U1 is the substrate S, U2 the enzyme E, U3 the complex ES, and U4 the product P.

k1 = 10;

k\_1 = 1;

k2 = 100;

k\_2 = 10;

dU1/dt = -k1\*U1\*U2 + 0\*U2 + (k\_1)\*U3 + 0\*U4

dU2/dt = 0\*U1 -k1\*U1\*U2 + (k\_1+k2)\*U3 + -k\_2\*U2\*U4

dU3/dt = k1\*U1\*U2 + 0\*U2 -(k\_1+k2)\*U3 + k\_2\*U2\*U4

dU4/dt = 0\*U1 + 0\*U2 + k2\*U3 -k\_2\*U2\*U4

We initialize the matrix assigning some initial concentrations to the species:

Uo = [2.0 0.2 0.0 0.0]';

U1 = Uo(1);

U2 = Uo(2);

row1 = [ -k1\*U2 0 k\_1 0 ];

row2 = [ 0 -k1\*U1 (k\_1+k2) -k\_2\*U2 ];

row3 = [ k1\*U2 0 -(k\_1+k2) k\_2\*U2 ];

row4 = [ 0 0 k2 -k\_2\*U2 ];

K = [row1;row2;row3;row4];

end\_time = 3.0;

time\_step = 0.01;

time\_vec = [0:time\_step:3];

nsteps = length(time\_vec);

U = zeros(4,nsteps); U(:,1) = Uo;

for i = 2:nsteps

K = [row1;row2;row3;row4];

[S,D] = eig(K);

D = diag(D);

Dt = D\*time\_step;

eDt = diag(exp(Dt));

U(:,i) = (S\*eDt/S)\*U(:,i-1);

U1 = U(1,i);

U2 = U(2,i);

row1 = [ -k1\*U2 0 k\_1 0 ];

row2 = [ 0 -k1\*U1 (k\_1+k2) -k\_2\*U2 ];

row3 = [ k1\*U2 0 -(k\_1+k2) k\_2\*U2 ];

row4 = [ 0 0 k2 -k\_2\*U2 ];

end

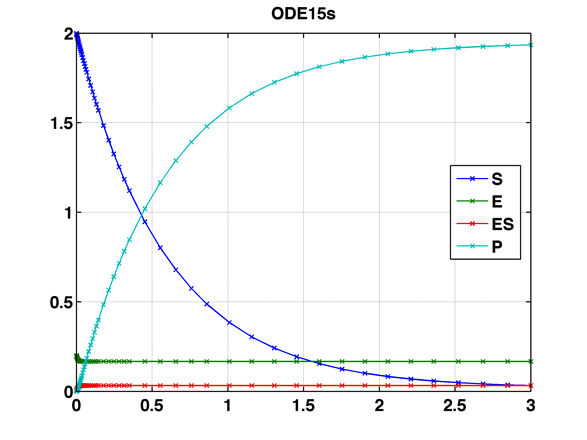
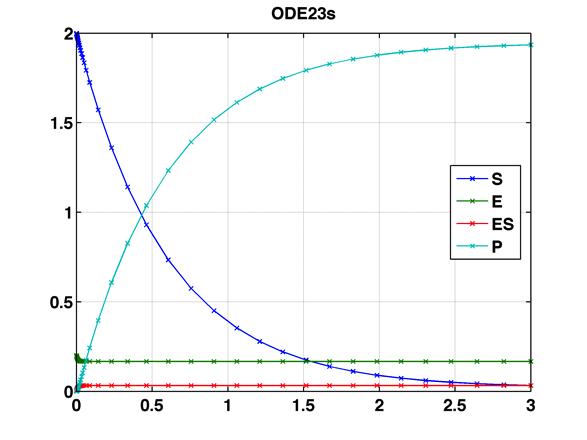
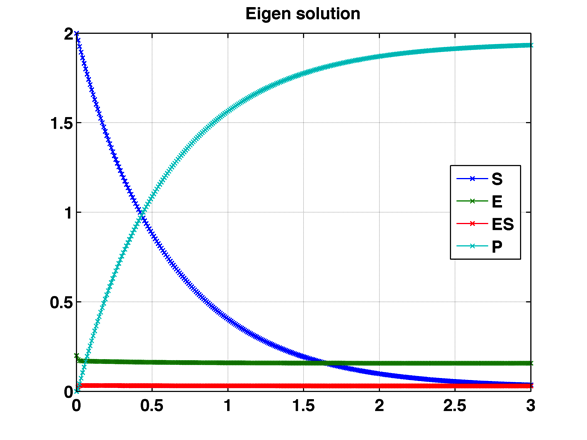
figure;plot(time\_vec(1:4:end),U(:,1:4:end),'.-')

legend('S','E','ES','P','Location','best')

grid on

To use the MATLAB ODE solvers we generate a new function 'Fp\_enz' for the derivatives:

function du = Fp\_enz(t,Uo)



k1 = 10;

k\_1 = 1;

k2 = 100;

k\_2 = 10;

U1 = Uo(1);

U2 = Uo(2);

row1 = [ -k1\*U2 0 k\_1 0 ];

row2 = [ 0 -k1\*U1 (k\_1+k2) -k\_2\*U2 ];

row3 = [ k1\*U2 0 -(k\_1+k2) k\_2\*U2 ];

row4 = [ 0 0 k2 -k\_2\*U2 ];

K = [row1;row2;row3;row4];

du = K\*Uo;

After which we can compare the results:

tspan = [0 3];

[time\_vec,u\_vec] = ode23s(@Fp\_enz,tspan,Uo);

figure;plot(time\_vec,u\_vec,'-x');legend('S','E','ES','P','Location','best');

title('ODE23s');grid on

tic

[time\_vec,u\_vec] = ode15s(@Fp\_enz,tspan,Uo);

toc

figure;plot(time\_vec,u\_vec,'-x');legend('S','E','ES','P','Location','best');

title('ODE15s');grid on

**SPECIAL TOPICS**

**Conversion of a single higher order differential equation into a system of 1st order differential equations.**

Most ODE solvers (including the entire MATLAB suite) only solve 1st order differential equations. What do we do if we have a higher order differential equation? For example, consider the 2nd order differential equation:

This is one of the most important equations in mechanics. The 1st term is *m* times the acceleration *a*. This term balances the force *F* (Newton's law), which includes the dampingand the restoring force proportional to the distance moved. Although 2nd order, this equation is still linear with constant coefficients *m*, *b*, and *k*.

We can convert this 2nd order *differential equation* into a *vector equation*, that is a *system* of 1st order differential equations. First, we notice that:

Then, we introduce the new variables and :

The system of 1st order equations is now easily set in the form :

which can be solved the usual way with a growth matrix as ***u***(n+1) = ***Gu***(n) once the stepsize Δ*t* and the initial vector ***u****(0)* are chosen*.* For example, using *RK4*:

First we set up the parameters and the initial values for , which includes and , consistent with the fact that the differential equation is 2nd order, and thus it requires two boundary conditions:

m = 3;b = 1; k = 3; u\_vec\_0 = [0.1;1]

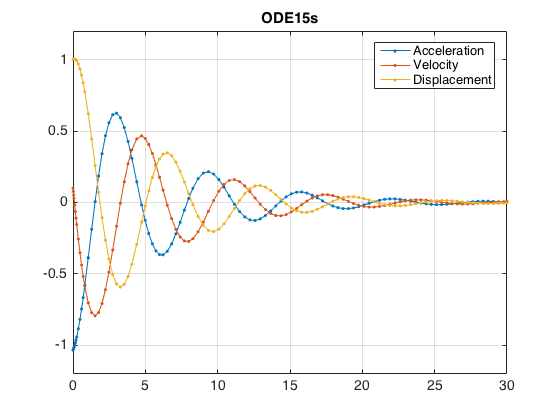
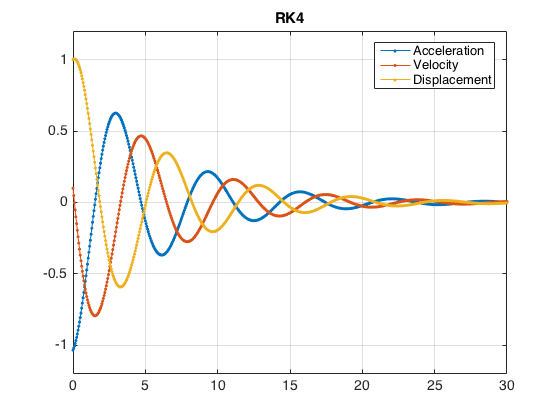
Here we set up the ***K*** matrix and the time vector:

K = [-b/m -k/m;1 0];

dt = 0.05; end\_time = 30; time\_vec = [0:dt:end\_time];

nsteps = length(time\_vec);

Here we set up the ***G*** matrix and the arrays where we will store the result:



I = eye(2);

K1 = K;

P = I + K\*dt/2;

K2 = K\*P;

Q = I + K\*P\*dt/2;

K3 = K\*Q;

R = I + K\*Q\*dt;

K4 = K\*R;

G = I + dt\*(K1 + 2\*K2 + 2\*K3 + K4)/6;

u\_vec = zeros(2,nsteps);

u\_vec(:,1) = u\_vec\_0;

Here we follow the spring dumped oscillation:

for i = 1:nsteps-1

u\_vec(:,i+1) = G\*u\_vec(:,i);

end

udot\_vec = (K\*u\_vec);

u\_vec2 = [udot\_vec(1,:) ; u\_vec];

Finally, we plot:

Dumped\_oscillation\_RK4 = figure;

plot(time\_vec,u\_vec2,'.-')

legend('Acceleration','Velocity','Displacement','Location','best')

title('RK4'); xlim([0 end\_time]);ylim([-1.2 1.2]);grid on

We obtain a similar result using MATLAB *stiff* solver ODE15s, but with far fewer points:

Fp = @(t,u) K\*u;

tspan = [0 end\_time];

[time\_vec,u\_vec] = ode15s(Fp,tspan,u\_vec\_0);

udot\_vec = (K\*u\_vec')';

u\_vec2 = [udot\_vec(:,1) u\_vec];

Dumped\_oscillation\_ODE15s = figure;

plot(time\_vec,u\_vec2,'.-');legend('Acceleration','Velocity','Displacement','Location','best');

title('ODE15s');

grid on;xlim([0 end\_time]);ylim([-1.2 1.2]);

This strategy applies to differential equations of any order. For example, the equations:

in which we use the alternative *prime* ()notation for derivatives, becomes:

Thus, as a general rule, the first row of the ***K*** matrix contains always the original equation coefficients, while the remaining rows contain only 1's along the diagonal. Notice that since in this case we have a 3rd order differential equation, we define three boundary conditions as the vector:

.

**Differential Algebraic Equations**

Differential algebraic equations (DAEs) are a type of differential equation where one or more derivatives of the dependent variables are not present in the equations. Variables that appear in the equations without their derivative are called algebraic, and the presence of algebraic variables means that it is not possible to write down the equations in the *explicit form*:

Instead, it is possible to write DAEs in one the following forms:

*Fully implicit:*

In this form, the distinction between DAEs and ODEs becomes apparent if the vector of dependent variables is written as a pair and the system of differential equations of the DAE appears in the form:

where: *y*, a vector in , are dependent variables for which derivatives are present (*differential variables*),

*x*, a vector in , are dependent variables for which no derivatives are present (*algebraic variables*),

*t*, a scalar (usually time) is an independent variable.

, a vector of *n* + *m* functions that involve *n* derivatives.

*Implicit:*

where is a *mass matrix.* In this form, the presence of algebraic variables leads to a singular mass matrix, since there are one or more zeros on the main diagonal.

*Semi-explicit:*

as it shows explicitly the constraint in the 2nd equation.

The *Index* of a DAE is a notion used to measure the distance from a DAE to its related ODE. Since a DAE involves a mixture of differentiations and integrations, differentiating the constraints (in a semi-explicit DAE system) and substituting, repeatedly if necessary, from the differential equations can yield an explicit ODE system for all unknowns. The number of repetitions needed for this transformation is called the *differential index* of the DAE. Thus, ODEs have *index 0*. For example, let *q*(*t*) be a smooth function, the scalar equation:

is an index-1 DAE, since it takes one differentiation to obtain the ODE . Instead, for the system:

differentiating we get:

The index is 2 since two differentiations are needed.

As an example of DAE applied to a common chemical reaction, we can reformulate the enzymatic reaction:

k1 k2

U1 + U2 ⇔ U3 ⇔ U4 + U2

k\_1 k\_2

(where U1 is the substrate S, U2 the enzyme E, U3 the complex ES, and U4 the product P), to contain a *constraint* forcing the sum of U2 and U3 to be always equal to the initial concentration of U2. We define the constraint in the function Fp\_enz\_DAE.m, in which we add an extra variable u(5) to represent the total concentration of the enzyme at time 0.

function du = Fp\_enz\_DAE(t,u)

k1=10; k\_1=1; k2=100; k\_2=10;te = u(5);

du=zeros(5,1);

du(1) = -k1\*u(2)\*u(1) + k\_1\*u(3);

du(2) = -k1\*u(1)\*u(2) + (k\_1+k2)\*u(3) - k\_2\*u(2)\*u(4);

differential equations

du(3) = k1\*u(2)\*u(1) - (k\_1+k2)\*u(3) + k\_2\*u(2)\*u(4);

du(4) = k2\*u(3) - k\_2\*u(2)\*u(4);

du(5) = u(2) + u(3) - te;

algebraic equation

end

We use a *mass matrix* to define the 5th equation (*mass balance* constraint) as *algebraic*:

M = eye(5); M(5,5) = 0;

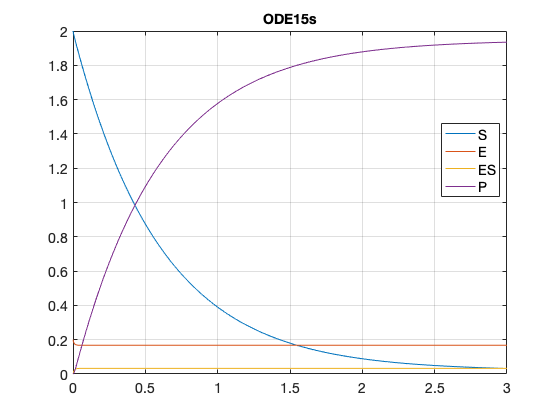
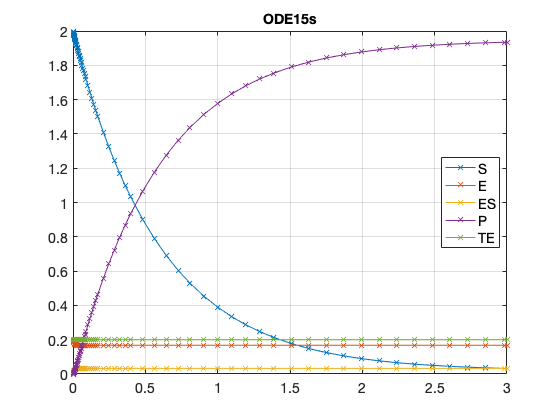
Uo = [2.0 0.2 0.0 0.0 0.2];

options=odeset('Mass',M,'RelTol',1e-4,'AbsTol',[1e-6 1e-10 1e-6 1e-10 1e-6])

tspan = [0 3]; [time\_vec,u\_vec] = ode15s(@Fp\_enz\_DAE,tspan,Uo,options);

figure(1);plot(time\_vec,u\_vec,'-x');

legend('S','E','ES','P','TE','Location','best'); title('ODE15s');grid on



Alternatively:

sol = ode15s(@Fp\_enz\_DAE,tspan,Uo,options);

x1=linspace(0,3,1000);

[y,z] = deval(sol,x1); [y1,z1] = deval(sol,x1,1); % Following just the substrate

figure(2); plot(x1,y(1:4,:),'-'); legend('S','E','ES','P','Location','best'); title('ODE15s');grid on

**Molecular dynamics and the leapfrog/velocity Verlet algorithm.**

Determination of the position and the velocity of individual particles in a chemical system requires integrating Newton equation of motion. Assuming for all particles *mass*, *m =* 1, and velocity independent forces, the equations of motions are:

Forward Euler would approximate the 1st equation as:

where *h* is the time stepsize. As we have seen in the 1st practice problem of CHAPTER 12 a better approximation would be to replace *v* with its value at the midpoint of the interval:

Then, since we now know we can apply the midpoint method to the 2nd equation:

and step forward with . Thus, once we start with and we can continue with and *leapfrogging* each other as shown on the side:

The basic integration formula for the *leapfrog algorithm* is therefore:

in which and are vectors containing the coordinates and velocities of all the particles in the system. Clearly, at each step the forces ***F*** are recomputed based on the new coordinates. Since it is based on the *midpoint method* the *leapfrog method* is .

However, to start the leapfrog we need , but we only have the initial velocity (and the initial position ). An approximation of is obtained by doing a single half-step of forward Euler:

Although this is not a midpoint method, and so has a *discretization* *error* *DE* of , we only do this once so it does not lower the order of the method, which remains 2nd order.

A second problem with the leapfrog method is how to get velocity and position at the same time to produce *phase space* plots and to compute energy and angular momentum. The simplest solution to this problem is to consider the equation as composed of two equal half steps, which successively relate to and to .

The *leapfrog algorithm* with a provision to start the algorithm and determine and at the same times, is called the *velocity Verlet*. Verlet integration was first used by Carl Størmer to compute the trajectories of particles moving in a magnetic field (hence it is also called Störmer's method) and was popularized in molecular dynamics by French physicist Loup Verlet in 1967.

A single time step can be written as:

where *n* = 0, 1, 2, ... . Although it looks like in the *velocity Verlet* we do two force calculations per time step as opposed to the standard *leapfrog*, this is not true because the force in the third line of the algorithm is the same as the force in the first line of the next step, and thus it can be stored and reused. Since *velocity Verlet* is the same as *leapfrog*, it is also a second order method. Notice that eliminating the half-step velocity calculation the *velocity Verlet* algorithm can be shortened to:

which is the form commonly used in software implementations.

It is often useful to show the trajectory of a particle as a *phase space* plot (a path in the *p-x* plane, where *p = mv* is the *momentum*). For example, consider a mass attached to a spring with force constant *k* (thereby spring forces are given by *kx*), and apply to it the Newton equation of motion:

In the presence of external forces *f* we have an equilibrium between *f* and *kx* and the mass does not move:

However, in the absence of damping or external forces all the forces come from the spring and the equation becomes:

Under these conditions the spring will oscillate forever. This system represents a *simple harmonic oscillator* for which the total energy:

*E = T* (kinetic energy) + *V* (potential energy) = *mv2/2 + kx2/2*

is constant. If we set *m = 1* and *k = 1* we obtain:

Hence the *phase space* plot is a circle with radius equal to . In order to see this clearly, we can split the 2nd order differential equation into two equations:

All the methods that solve these equations by *finite differences* (that is converting *differential* into *difference* equations) replace:

The crucial difference between methods (i.e. *explicit* or *implicit*) stems from how they evaluate the right end sides and . The following are the *growth matrices* ***G*** for four different methods we have discussed:

***Forward Euler*** (explicit):

***Backward Euler*** (implicit):

***Trapezoidal*** (implicit):

***Velocity Verlet*** (explicit):

Harmonic oscillator

close all

clear

x0 = 1;

v0 = 0;

xv0 = [x0; v0];

h = 2\*pi/32

nsteps = 33

Circle\_step = [0:pi/16:2\*pi]

Circle\_x = cos(Circle\_step);

Circle\_y = sin(Circle\_step);

Forward Euler

xv = zeros(2,32);

xv(:,1) = xv0;

G = [1 h;-h 1];

for i = 2:nsteps

xv(:,i) = G\*xv(:,i-1);

end

xv = xv';

figure;plot(Circle\_x,Circle\_y,':b')

hold on

plot(xv(:,1),xv(:,2),'-or')

axis equal

xlim([-2 2]);ylim([-1.4 1.8]);

hold off

vline([0],'y');hline([0],'y')

xlabel('x');ylabel('v');

title('Forward Euler')

Backward Euler

xv = zeros(2,32);

xv(:,1) = xv0;

G = inv([1 -h;h 1]);

for i = 2:nsteps

xv(:,i) = G\*xv(:,i-1);

end

xv = xv';

figure;plot(Circle\_x,Circle\_y,':b')

hold on

plot(xv(:,1),xv(:,2),'-or')

axis equal

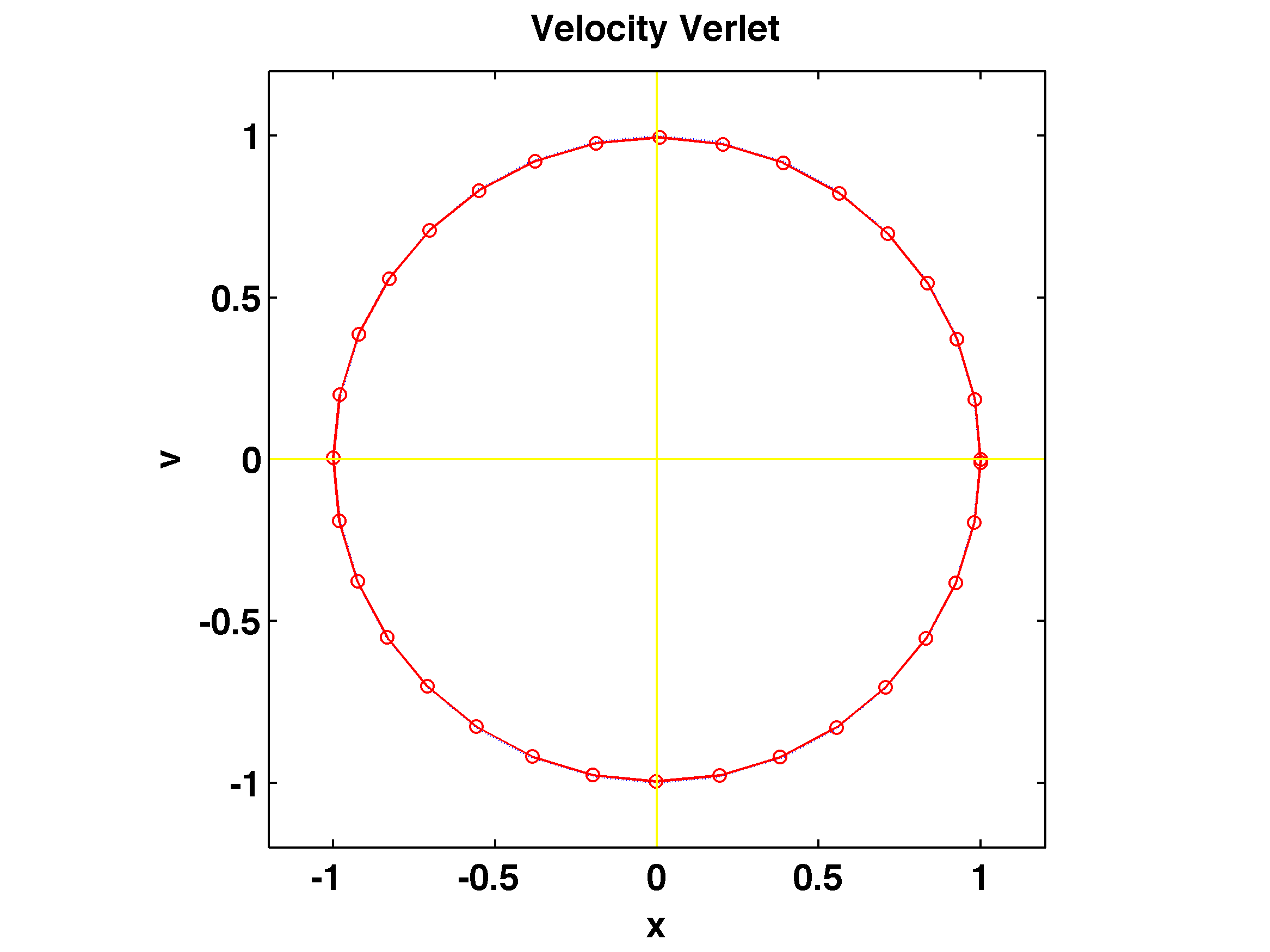
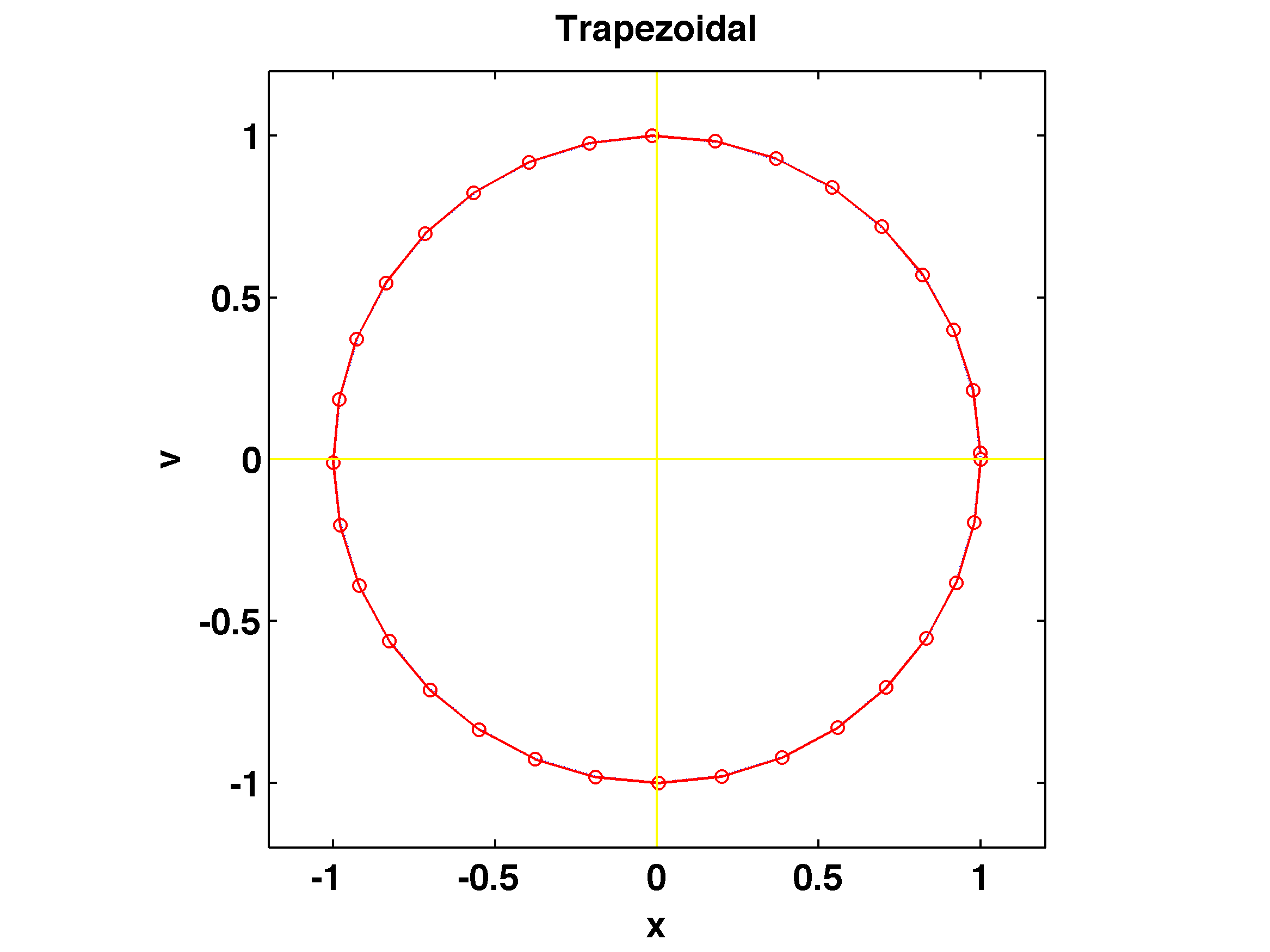
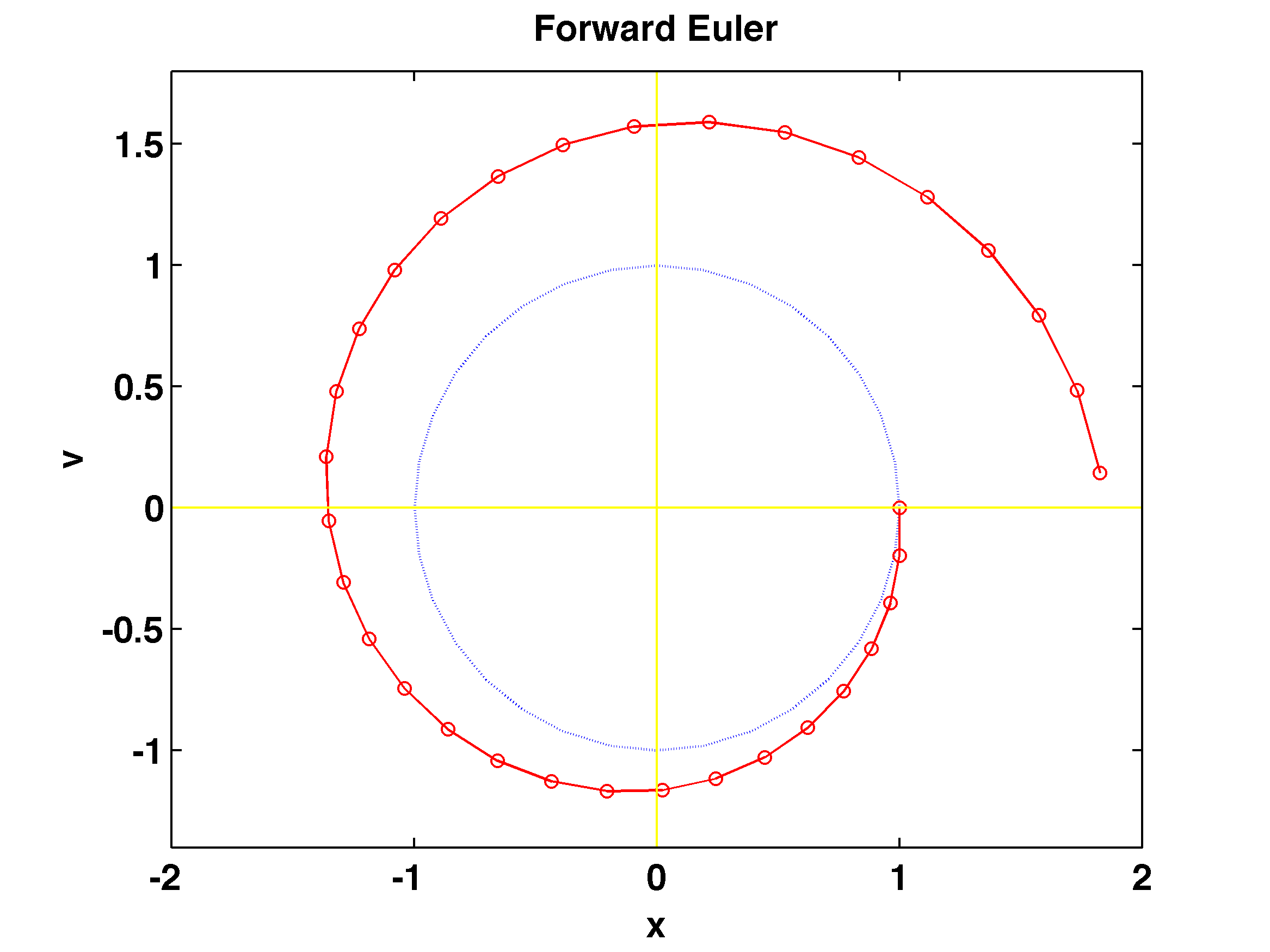
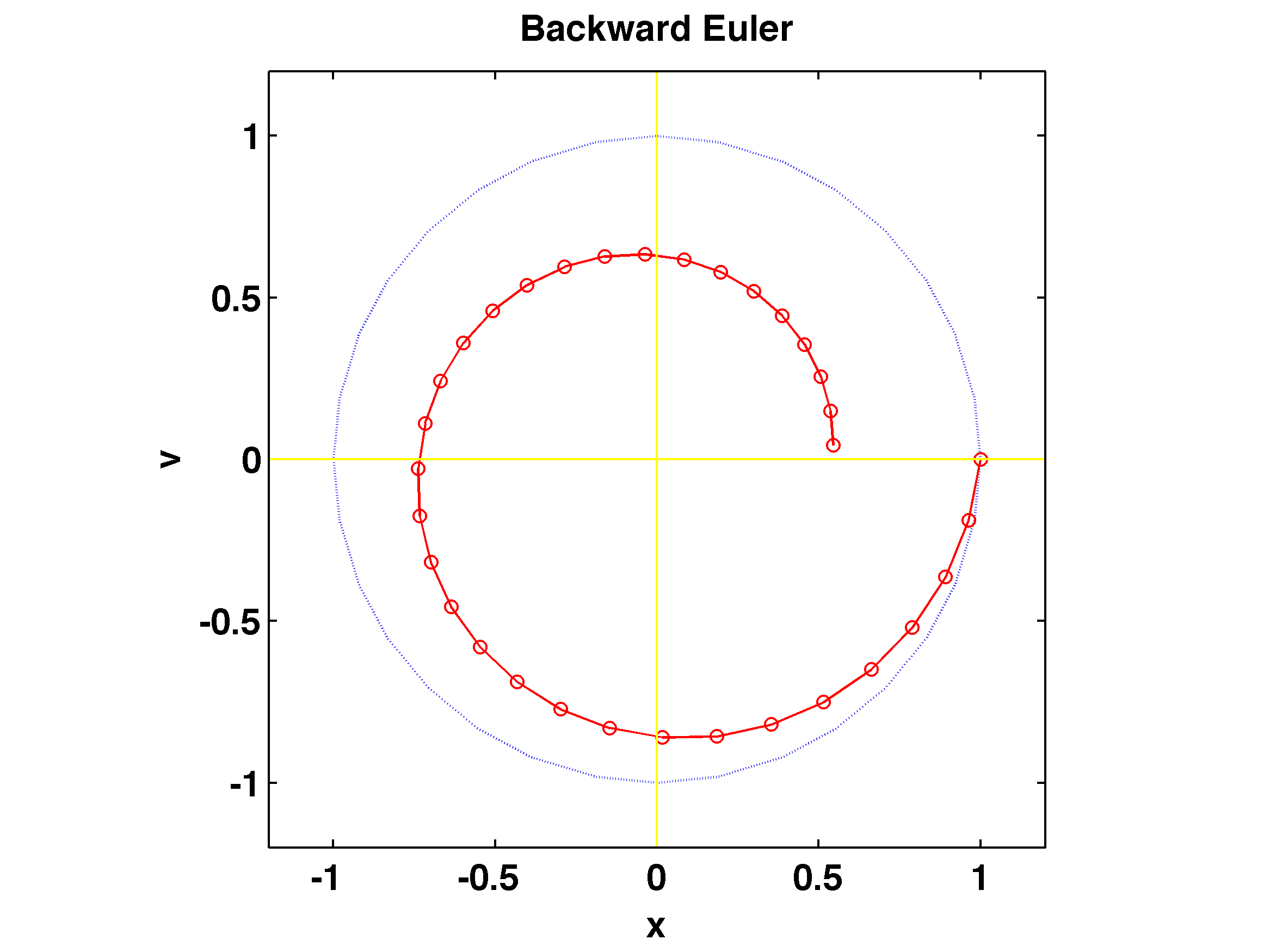
xlim([-1.2 1.2]);ylim([-1.2 1.2]);

hold off

vline([0],'y');hline([0],'y')

xlabel('x');ylabel('v');

title('Backward Euler')



Trapezoidal

xv = zeros(2,32);

xv(:,1) = xv0;

G = [1 -h/2;h/2 1]\[1 h/2;-h/2 1];

for i = 2:nsteps

xv(:,i) = G\*xv(:,i-1);

end

xv = xv';

figure;plot(Circle\_x,Circle\_y,':b')

hold on

plot(xv(:,1),xv(:,2),'-or')

axis equal

xlim([-1.2 1.2]);ylim([-1.2 1.2]);

hold off

vline([0],'y');hline([0],'y')

xlabel('x');ylabel('v');

title('Trapezoidal')

E = 0.5\*(xv(:,1).^2+xv(:,2).^2);

figure; plot([0:32],E)

xlabel('step');ylabel('E');

title('Energy')

Velocity Verlet

xv = zeros(2,32);

xv(:,1) = xv0;

h2 = h\*h;

h3 = h\*h\*h;

G = [1-0.5\*h2 h;-h+0.25\*h3 1-0.5\*h2];

for i = 2:nsteps

xv(:,i) = G\*xv(:,i-1);

end

xv = xv';

figure;plot(Circle\_x,Circle\_y,':b')

hold on

plot(xv(:,1),xv(:,2),'-or')

axis equal

xlim([-1.2 1.2]);ylim([-1.2 1.2]);

hold off

vline([0],'y');hline([0],'y')

xlabel('x');ylabel('v');

title('Velocity Verlet')

E = 0.5\*(xv(:,1).^2+xv(:,2).^2);

figure; plot([0:32],E)

xlabel('step');ylabel('E');

title('Energy')

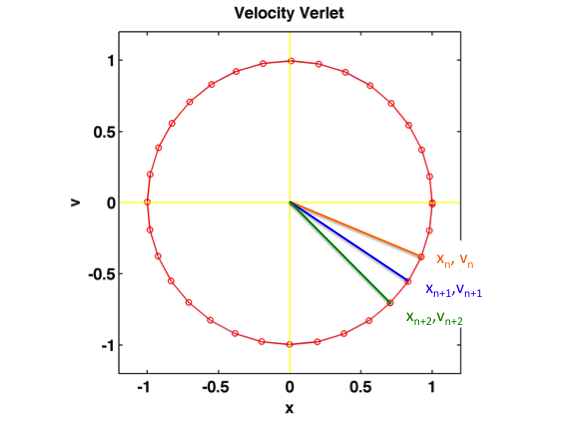
Both *Euler methods* leave the circle very rapidly because of low accuracy. Furthermore, there is a small phase error because after 32 steps = 2π the trace on the phase plane does not return eactly to the x axis. The *Trapezoidal method* stays exactly on the circle, but still has a small phase error: however, *energy* is conserved because the ***G*** matrix is orthogonal and therefore does not change the length of the *xv* vector. The *Velocity Verlet method* has the advantage of being a completely *explicit* method and stays well on the circle with an even smaller phase error. However, the ***G*** matrix is not exactly orthogonal, and for this reason *energy* is not perfectly conserved and oscillates.

Notice that in the more general case, if *m ≠* 1 and *k ≠* 1, and masses and elastic constants are stored as diagonal matrices ***M*** and ***K***, energy calculation takes into account these matrices:

*E = mv2/2* (kinetic energy, T) *+ kx2/2* (potential energy, V)

Because the *determinant* of the ***G*** matrix in the *Trapezoidal* and *velocity Verlet* methods is 1, these methods preserve the *area of each step* inside the circle on the phase plane, and for this reason are called ***symplectic*** (*area preserving*). A famous theorem (*Liouville’s* theorem, whose derivation can be found in advanced textbooks of classical mechanics) proves that integration of the Newton equation for a *finite time* can be regarded as a map in phase space (i.e. the v-x space), and that Newtonian dynamics give rise to an *area-preserving* map for each step. Therefore, it is paramount that this property, which corresponds to Kepler's 2nd law for orbiting planets(*equal areas in equal times*) be preserved for the success of long time numerical integration by any method.

To understand the meaning of the determinant in this context, consider a vector with coordinates X1, X2. Ifwe want to stretch it in some direction we multiply the coordinates by a matrix called the *deformation gradient* (usually represented in *mechanics* with the symbol ***F***) because its elements are the partial derivatives of the final coordinates (expressed with lower case letters x1, x2) with respect to the initial coordinates (expressed with upper case letters X1, X2):

Geometrically, the columns of the *deformation gradient* ***F*** are vectors pointing in the direction of the deformation. Now consider the triangles in the *phase plane* formed by the vectors *xn*,*vn* and *xn+1*,*vn+1*, and *xn+1*,*vn+1* and *xn+2*,*vn+2*. The areas A1 and A2 of the triangles are:

We also know that:

in which the *growth* matrix ***G*** corresponds to the *deformation gradient* matrix ***F***.Since the determinant of the product of two matrices is the product of the determinants of the individual matrices we have:

is often called the *Jacobian* of the *deformation gradient*. It follows that if , then:

which proves that the area of the phase plane inside the circle swept at each time step is constant.

**Topology and dynamics of a network of reactions.**

When several chemical or enzymatic reactions act in concert sharing several reactants and products they are best described as a *network* of reactions. A *metabolic pathway* inside a cell is a typical example of a network of reactions. Analysis and simulation of large networks of metabolic reactions is a key area of research in *Systems Biology*. It is customary in Systems Biology to split the analysis of a network in two components:

1. the network *structure* or *topology*.

2. the network *dynamics*.

The network topology is provided by the *stoichiometric matrix* ***S***. Every row of this matrix represents a compound and every column represents a reaction. For example, consider the coupled reactions:

syms x1 x2 x3 x4 k\_1 k\_m1 k\_2 k\_m2 t real

x = [x1 x2 x3 x4]'

The stoichiometric matrix of this small network of reactions is:

reaction 1 2 3 4

S = [-1 1 0 0;

-1 1 0 0;

1 -1 -1 1;

0 0 1 -1]

We can also represent the reaction in term of elementary reactions and velocities:

v1 = k\_1\*x1\*x2

v2 = k\_m1\*x3

v3 = k\_2\*x3

v4 = k\_m2\*x4

v = [v1 v2 v3 v4]'

in which case the time derivatives of the four compounds are given by the product of the stoichiometric matrix times the vector of elementary velocities (see also CHAPTER 19):

dxdt = S\*v

We can also define a *gradient matrix* ***G*** whose elements are:

G = [diff(v1,x1) diff(v1,x2) diff(v1,x3) diff(v1,x4);

diff(v2,x1) diff(v2,x2) diff(v2,x3) diff(v2,x4);

diff(v3,x1) diff(v3,x1) diff(v3,x3) diff(v3,x4);

diff(v4,x1) diff(v4,x1) diff(v4,x3) diff(v4,x4)]

If we modify the gradient matrix in order to avoid a double counting of the 2nd order reactions we obtain:

G = [(1/2)\*diff(v1,x1) (1/2)\*diff(v1,x2) diff(v1,x3) diff(v1,x4);

diff(v2,x1) diff(v2,x2) diff(v2,x3) diff(v2,x4);

diff(v3,x1) diff(v3,x1) diff(v3,x3) diff(v3,x4);

diff(v4,x1) diff(v4,x1) diff(v4,x3) diff(v4,x4)]

then the product of the *stoichiometrix matrix* ***S*** times the *gradient matrix* ***G*** is the *rate matrix*: in the language of *Systems Biology* this matrix is often called the *Jacobian* ***J*** of the reaction network:

The time derivatives of the four compounds can now be obtained as the usual product of the rate (or Jacobian) matrix ***J*** times the vector of concentrations:

J = S\*G

dxdt = J\*x

It is important to realize that in large networks of biochemical reactions events typically occur on multiple *time scales*. These time scales can be better appreciated by looking at the *time constants* ***τ***, which are the reciprocals of the reaction rate constants. For this purpose, it is very useful to split the reactions in their *independent normal modes* as we have learned in a previous section. We recall here the 3-step strategy to solve a system of differential equations:

is:

1. find the coordinates of [**x**]**0** in the *eigenvector* space:

2. write [**x**]**0** as a linear combination of the *eigenvectors* of***J***:

3. multiply each *eigenvector* component by the exponent of the product of each *eigenvalue* by time:

where ***V*** is here the matrix of *eigenvectors* ***v***, and ***Λ*** is the diagonal matrix of *eigenvalues λ* of the Jacobian ***J***. Using this strategy, the chemical network can be decomposed into independent components that evolve with different *relaxation time constants* (the reciprocals of the *eigenvalues*).

We can clearly see this decomposition with a numerical case:

k\_1 = 1; k\_m1 = .6; k\_2 = .2; k\_m2 = .02;

Xo = [2.0 2.0 2.0 2.0]';

X1 = Xo(1);X2 = Xo(2);

row1 = [-k\_1\*X2 0 k\_m1 0 ];

row2 = [0 -k\_1\*X1 k\_m1 0 ];

row3 = [(1/2)\*k\_1\*X2 (1/2)\*k\_1\*X1 -(k\_m1+k\_2) k\_m2];

row4 = [0 0 k\_2 -k\_m2];

J = [row1;row2;row3;row4];

time\_step = 0.1;

time\_vec = [0:time\_step:40];

nstep = length(time\_vec);

X = zeros(4,nstep);

W1 = zeros(4,nstep);W2 = zeros(4,nstep);

W3 = zeros(4,nstep);W4 = zeros(4,nstep);

X(:,1) = Xo;

[V,D] = eig(J)

tc = inv(D(1:3,1:3))

w = S\X(:,1);

W1(:,1) = w(1)\*V(:,1);

W2(:,1) = w(2)\*V(:,2);

W3(:,1) = w(3)\*V(:,3);

W4(:,1) = w(4)\*V(:,4);

for i = 2:nstep

J = [row1;row2;row3;row4];

[V,D] = eig(J);

D = diag(D);

Dt = D\*time\_step;

eDt = diag(exp(Dt));

w = V\X(:,i-1);

W1(:,i) = w(1)\*V(:,1)\*eDt(1,1);

W2(:,i) = w(2)\*V(:,2)\*eDt(2,2);

W3(:,i) = w(3)\*V(:,3)\*eDt(3,3);

W4(:,i) = w(4)\*V(:,4)\*eDt(4,4);

X(:,i) = W1(:,i) + W2(:,i) + W3(:,i) + W4(:,i);

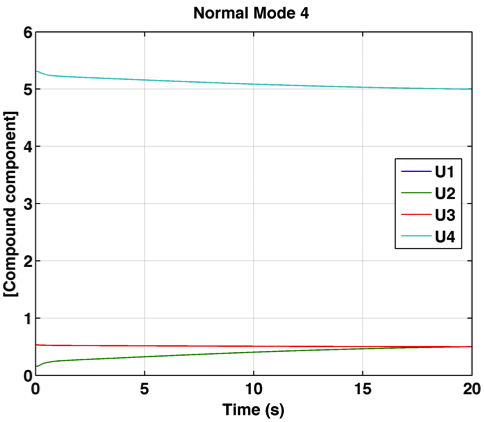
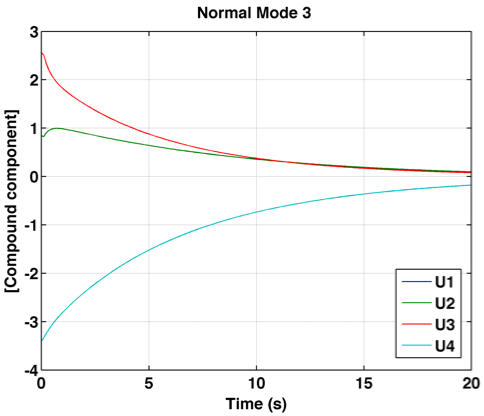
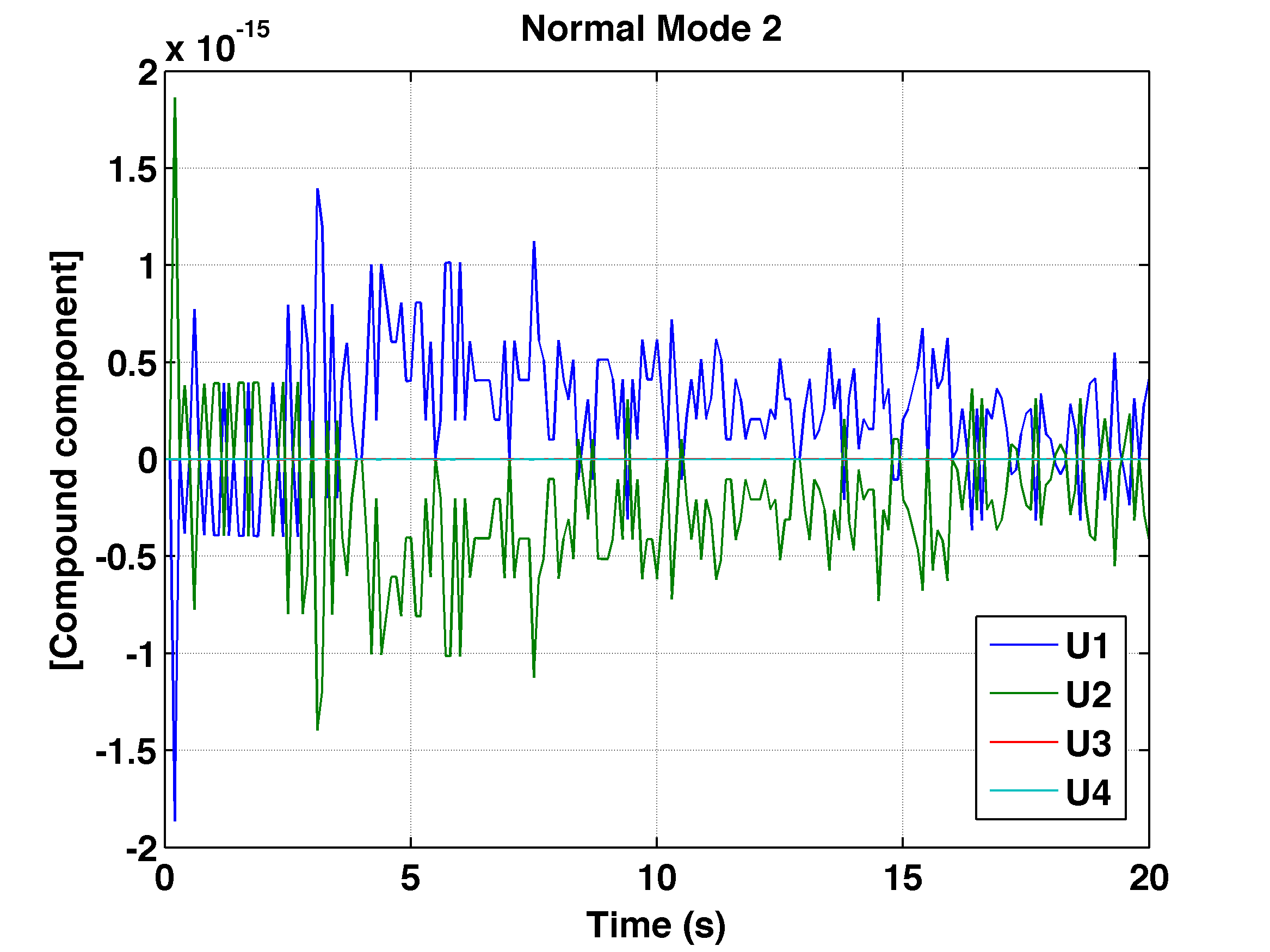
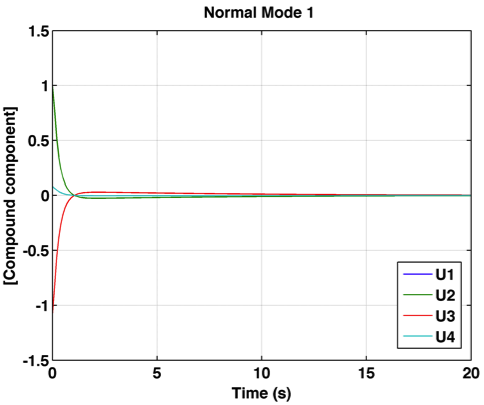
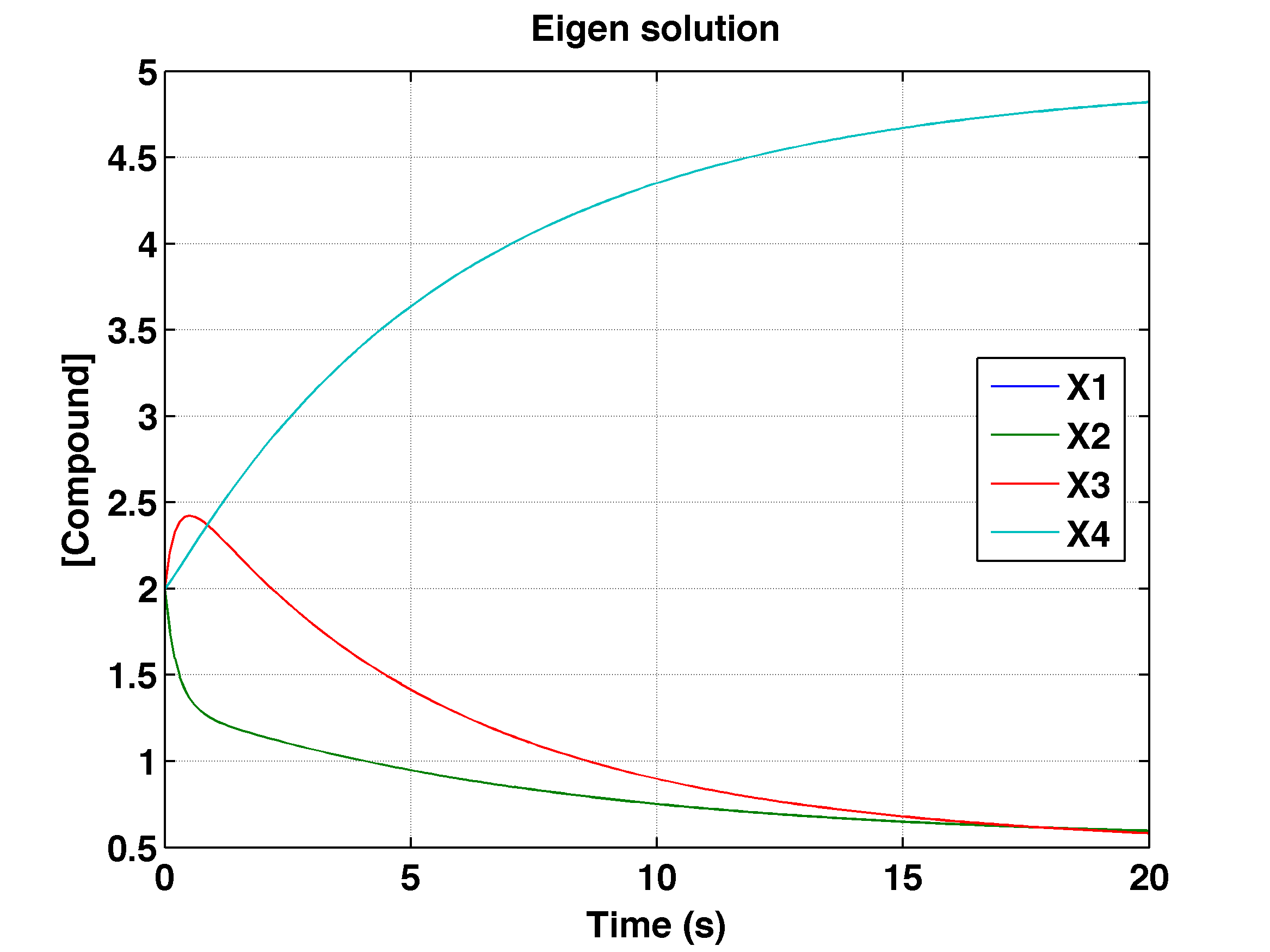
X1 = X(1,i);X2 = X(2,i);

row1 = [-k\_1\*X2 0 k\_m1 0 ];

row2 = [0 -k\_1\*X1 k\_m1 0 ];

row3 = [(1/2)\*k\_1\*X2 (1/2)\*k\_1\*X1 -(k\_m1+k\_2) k\_m2];

end



figure;plot(time\_vec,X,'-')

legend('X1','X2','X3','X4','Location','best')

title('Eigen solution');grid on

figure;plot(time\_vec(1:end),W1 ,'-')

legend('U1','U2','U3','U4','Location','best')

title('Eigen solution');grid on

figure;plot(time\_vec(1:end),W2 ,'-')

legend('U1','U2','U3','U4','Location','best')

title('Eigen solution');grid on

figure;plot(time\_vec(1:end),W3 ,'-')

legend('U1','U2','U3','U4','Location','best')

title('Eigen solution');grid on

figure;plot(time\_vec(1:end),W4 ,'-')

legend('U1','U2','U3','U4','Location','best')

title('Eigen solution');grid on

The *eigen* solution is the sum of the individual normal modes. The 1st mode has a time constant of ~0.4 s. The 2nd mode has a time constant of 0.5 s, but its amplitude is very small. The 3rd mode has a time constant of ~6 s. The 4th mode reflects the equilibrium value of the reactants at infinite time constant (rate constant = 0). Physically, the time constant represents the time it takes a *1st order* *linear time invariant* (= constant coefficients) system to reach ~63% of its final value, *if it increases*,or ~37% of its initial value, *if it decreases*. We can see from the progression of the different modes how this definition is not strictly valid for systems including *2nd order time dependent* (= non-constant coefficients) reactions. Nonetheless, it is easy to appreciate that there are two clearly distinct *time scales*, with one component reaching equilibrium in less than two seconds, and the other in more than 20 seconds. We can represent the dynamic behavior of this small network as a 3-dimensional volume in which one axis (eigenvector) represents a very fast motion and a plane (defined by two eigenvectors) provides the main observational window from which we follow the system. A large network of connected reactions can be conceptualized as a series of planes containing slow motions connected by fast motions. The chemical species populating each plane very often move together as a *pool*.

In order to identify the components of a pool, instead of representing the *time dependence* of the species concentration, it is informative to represent the *time dependent* changes of one species against the changes of another species: the resulting plot is called a *phase portrait*. Species that belong to the same pool produce a *straight line* in a phase portrait. This type of analysis can be carried out for the entire solution or for each normal mode. In our specific example, all the species behave as a pool in the 1st (fast motion) and 4th normal mode (equilibrium), but only ***x3*** and ***x4*** behave (approximately) as a pool in the 3rd normal mode (obviously ***x1*** and ***x2*** always move together).

figure;plot(X(1,:),X(3,:),'-',X(1,:),X(4,:),'-',X(3,:),X(4,:),'-')

legend('X1 versus X3','X1 versus X4','X3 versus X4','Location','best')

title('Eigen solution ');grid on; xlabel('[Species 1]');ylabel('[Species 2]')

figure;plot(W1(1,:),W1(3,:),'-',W1(1,:),W1(4,:),'-',W1(3,:),W1(4,:),'-')

legend('U1 versus U3','U1 versus U4','U3 versus U4','Location','best')

title('Normal Mode 1 ');grid on; xlabel('[Species 1]');ylabel('[Species 2]')

figure;plot(W3(1,:),W3(3,:),'-',W3(1,:),W3(4,:),'-',W3(3,:),W3(4,:),'-')

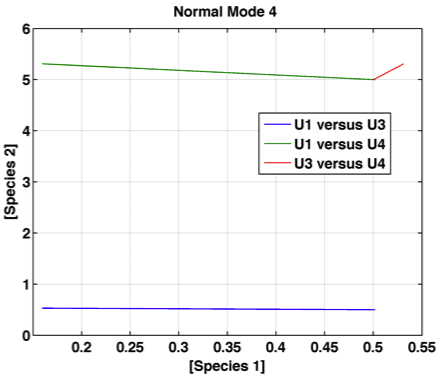
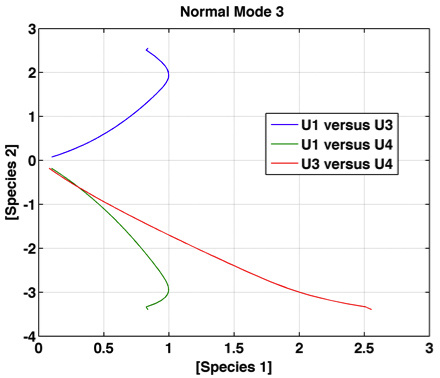
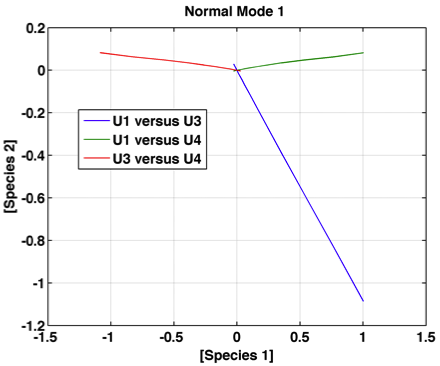
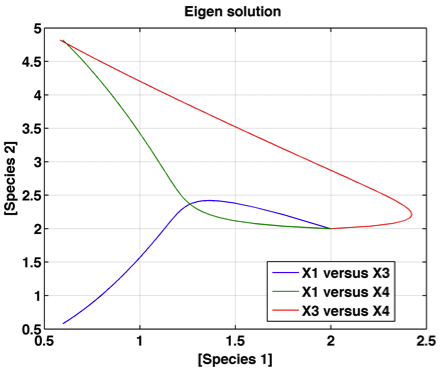
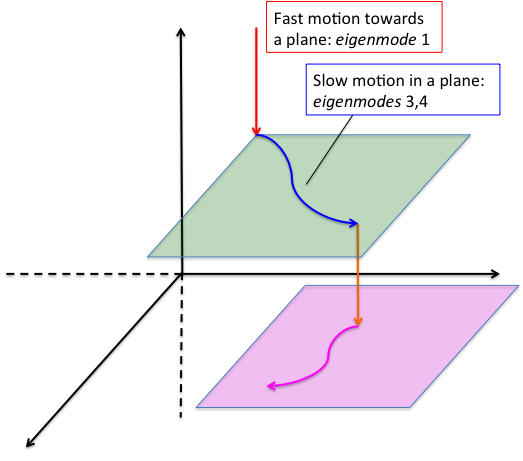
legend('U1 versus U3','U1 versus U4','U3 versus U4','Location','best')

title('Normal Mode 3 ');grid on; xlabel('[Species 1]');ylabel('[Species 2]')

figure;plot(W4(1,:),W4(3,:),'-',W4(1,:),W4(4,:),'-',W4(3,:),W4(4,:),'-')

legend('U1 versus U3','U1 versus U4','U3 versus U4','Location','best')

title('Normal Mode 4 ');grid on; xlabel('[Species 1]');ylabel('[Species 2]')



**PRACTICE**

1. Using as example/template the code for the forward Euler and trapezoidal methods write a program that uses the 'midpoint method' to simulate the equilibrium:

k1 k2 k3

U1⇔U2⇔U3⇔U4

k-1 k-2 k-3

# Since energy is not perfectly conserved in the *velocity Verlet* algorithm, when dealing with large systems of many particles it is necessary to rescale the energy periodically increasing or decreasing all the velocities. This is accomplished by keeping the system in equilibrium with a virtual *bath* at constant temperature. An example of this type of temperature control is shown in the function *simple\_md*, which was derived by simplifying and combining a group of functions originally written by Yanxiang Zhao at George Washington University (http://home.gwu.edu/~yxzhao/index.html).

a. Run a small molecular dynamics simulation with 50 particles at 300 K with temperature control using the function *simple\_md* with the following syntax:

trj = simple\_md(50,0.1,300,0.0005,1000,10);

and be ready to explain the different steps involved.

b. Display the ensemble evolution in time using *plot3*.

c. Can you modify the function in order to display a histogram of the velocities during the md simulation?