Differential Equations Basics

Differential Equations

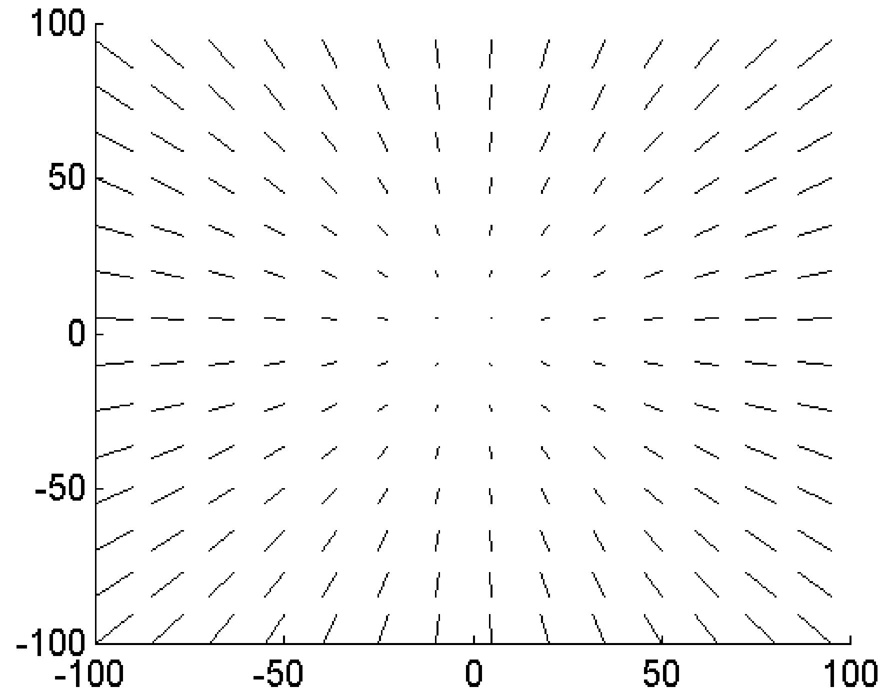
- Equations with derivative symbols :-)
- Ordinary differential equations
  - initial value problems
- Partial differential equations
  - boundary value problems, initial-boundary value problems
- Exact solutions
  - guess work
- Numerical solutions
  - guess work (but computer does it!)
- *Warning: This is not a course on numerical analysis! Take an appropriate course if interested*

Ordinary Differential Equations

- Usually the independent variable is treated as time (t) $\dot{x} = f(x,t)$

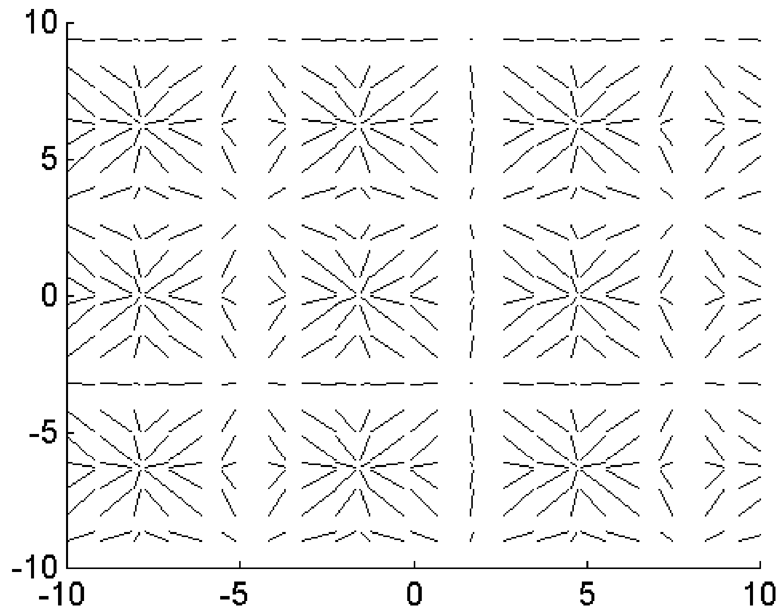
Example

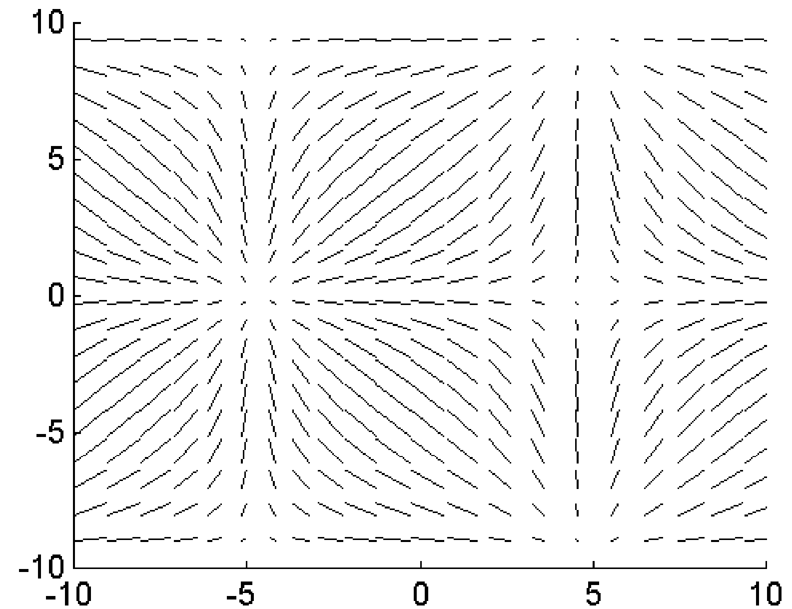
$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = -k \begin{bmatrix} |x| \\ |y| \end{bmatrix}, k > 0$$



More Examples

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = -k \begin{bmatrix} \cos(xt) \\ \sin(yt) \end{bmatrix}$$





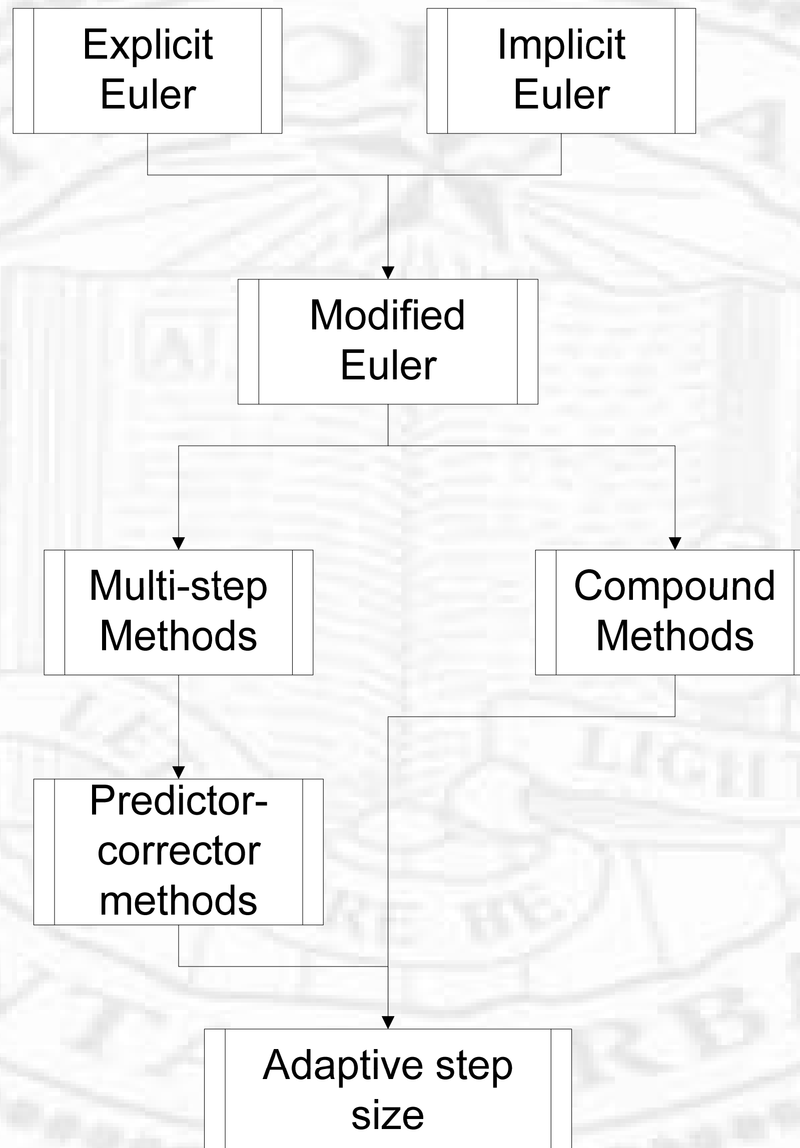
Intuition

- Imagine a vector field
  - field (flow) direction specified by
    - location, and
    - time
  - a particle in the vector field will drift according to the current flow direction where the particle is
  - time independent - dropping something in a river and watching it flows
  - time dependent - dropping something in the ocean and watching ocean waves carry it

Numerical Solutions

- Advance a particle from its initial location through (discrete) time steps
- Many issues
  - accuracy
  - stability
  - efficiency
- Many solutions
  - Euler (explicit, implicit, modified)
  - Midpoint and Runge-Kutta
  - Multistep, predictor-corrector
  - Adaptive step size

Roadmap



Issues

- Accuracy
  - How close is the discrete solution going to be to the continuous solution?
- Stability
  - Numerical error unavoidable
  - Error accumulation to grow or decay?
- Efficiency
  - How fast for each time step?
  - How large can each time step be?

(Explicit) Euler Method

- Go in the tangent direction with distance controlled by time step

$$\dot{x} = f(x,t) \qquad \dot{x} = -kx$$

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

$$\frac{x_{n+1} - x_n}{\Delta t} = f(x_n, t_n) \qquad \frac{x_{n+1} - x_n}{\Delta t} = -kx_n$$

$$x_{n+1} = x_n + \Delta t \cdot f(x_n, t_n) \qquad x_{n+1} = x_n - k\Delta t x_n = (1 - k\Delta t)x_n$$

Questions

- Is Euler method accurate?
- Is Euler method stable?
- How close are the the exact and Euler solutions?

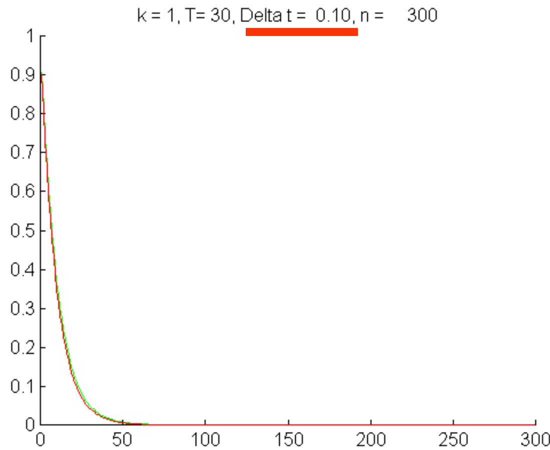
Exact solution

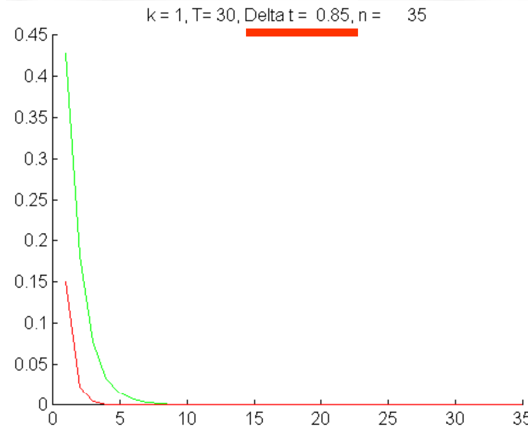
$$\dot{x} = -kx$$
$$\Rightarrow x = x_o e^{-kt}$$
$$\Rightarrow x(n\Delta t) = x_o e^{-kn\Delta t}$$

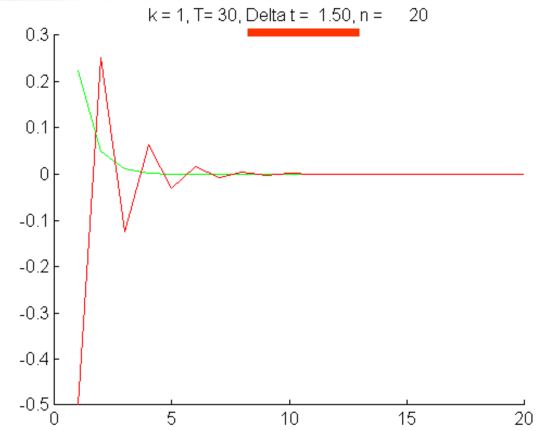
Euler solution

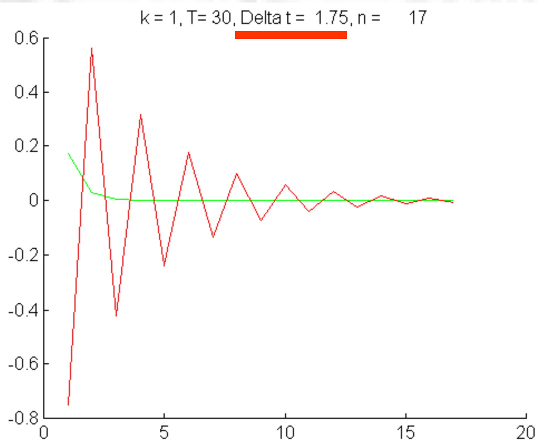
$$\dot{x} = -kx$$
$$x_n = x_{n-1} - k\Delta t x_{n-1} = (1 - k\Delta t)x_{n-1}$$
$$= \ldots = (1 - k\Delta t)^n x_o$$

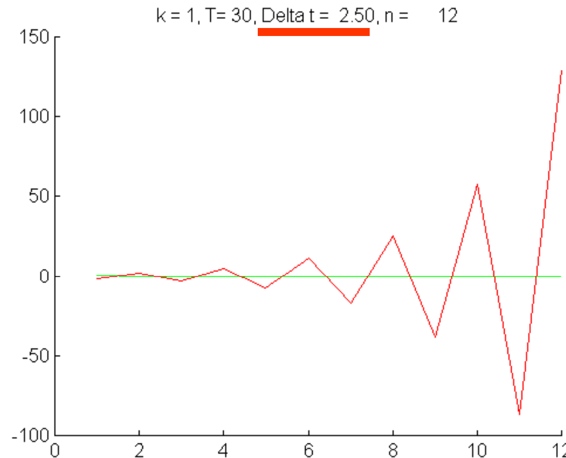
$\dot{x} = -x \qquad x_n = (1-\Delta t)^n x_o$ 30 - second simulation

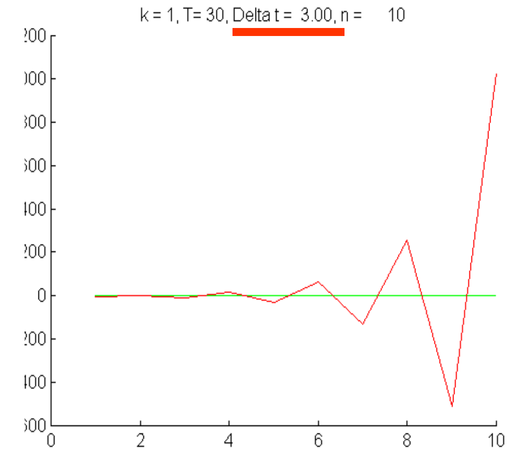








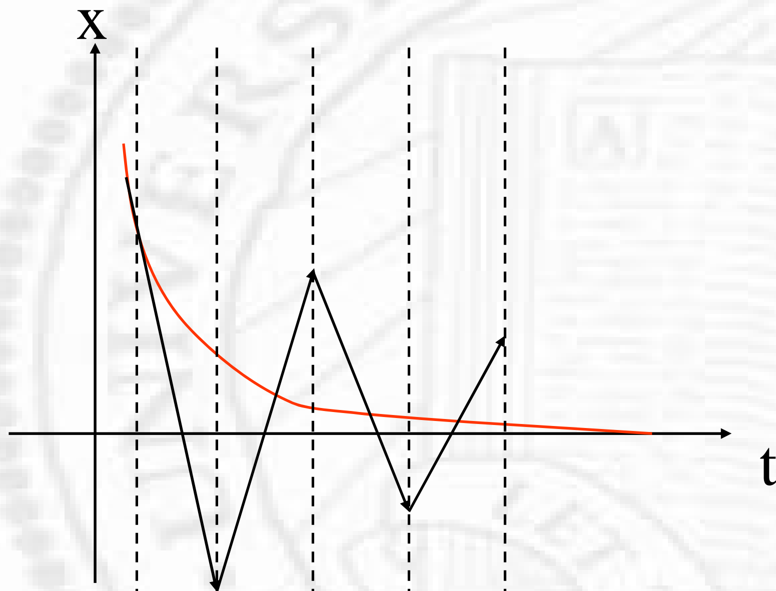




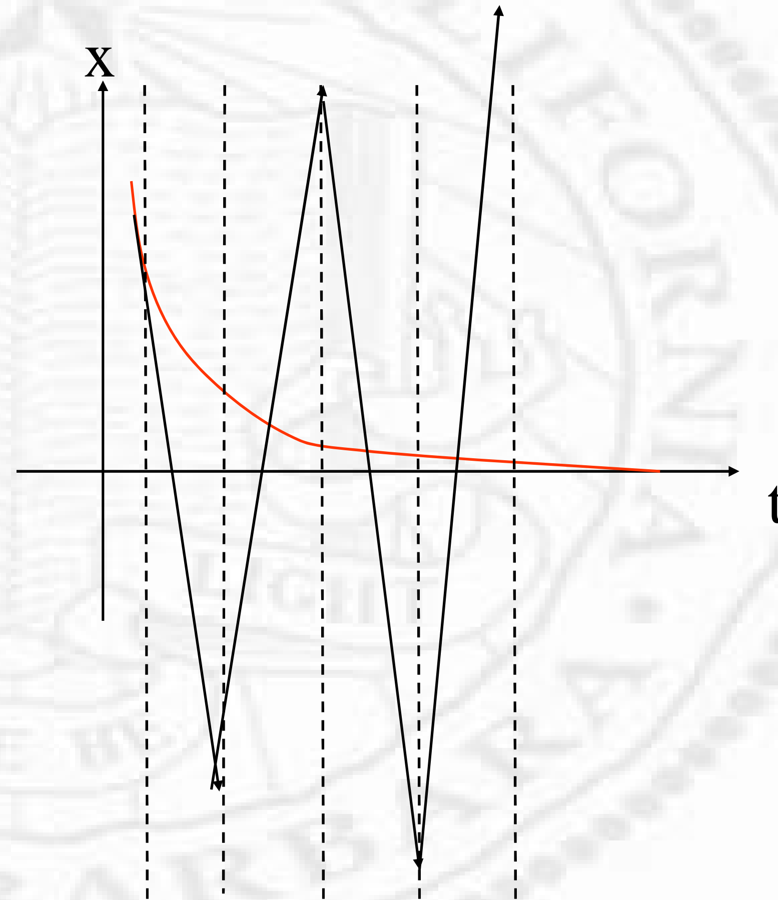
A graphical interpretation

$$\dot{x} = -kx$$

At large step sizes



At even large step sizes



Observations (Explicit Euler)

- Explicit Euler method can be made accurate - as long as you can tolerate (*very*) small step size
- It can be made stable (not blowing up!) with (*appropriate*) small step size
- Explicit Euler is easy to understand
- But it is seldom used in the real world (homework assignments are *not* real world, but should be treated as such ☺)

$$\dot{x} = -kx$$

$$x_{n+1} = (1 - k\Delta t)^n x_o$$

$$\Rightarrow 1 - k\Delta t \geq -1$$

$$\Rightarrow k\Delta t \leq 2$$

$$\Rightarrow \Delta t \leq \frac{2}{k}$$

How Small Should Step Size Be?

- The larger the k, the smaller the step size
- Equation of large k, or system of equations with large varying k, are stiff

$$\dot{x} = -kx$$

$$x_{n+1} = (1 - k\Delta t)^n x_o$$

$$\Rightarrow 1 - k\Delta t \geq -1$$

$$\Rightarrow k\Delta t \leq 2$$

$$\Rightarrow \Delta t \leq \frac{2}{k}$$

(Implicit) Euler Method

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

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

$$\frac{x_{n+1} - x_n}{\Delta t} = f(x_{n+1}, t_{n+1})$$

$$x_{n+1} - \Delta t \cdot f(x_{n+1}, t_{n+1}) = x_n$$

$$x_{n+1} = x_n + \Delta t \cdot f(x_{n+1}, t_{n+1})$$

$$\dot{x} = -kx$$

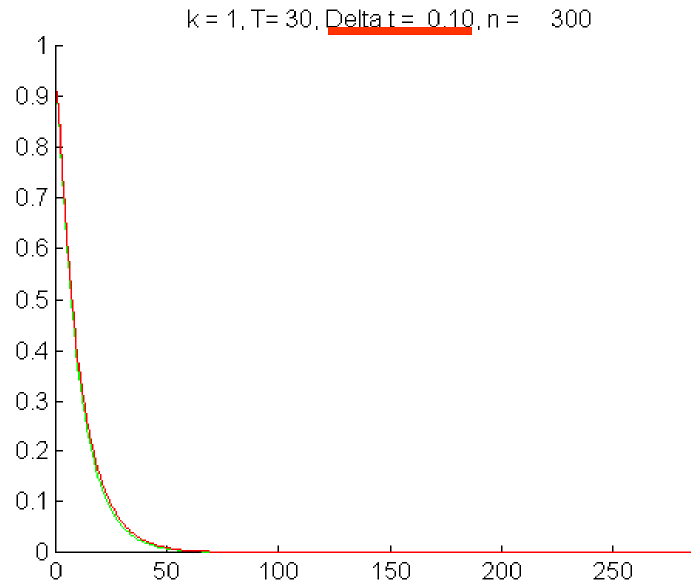
$$\frac{dx}{dt} = -kx$$

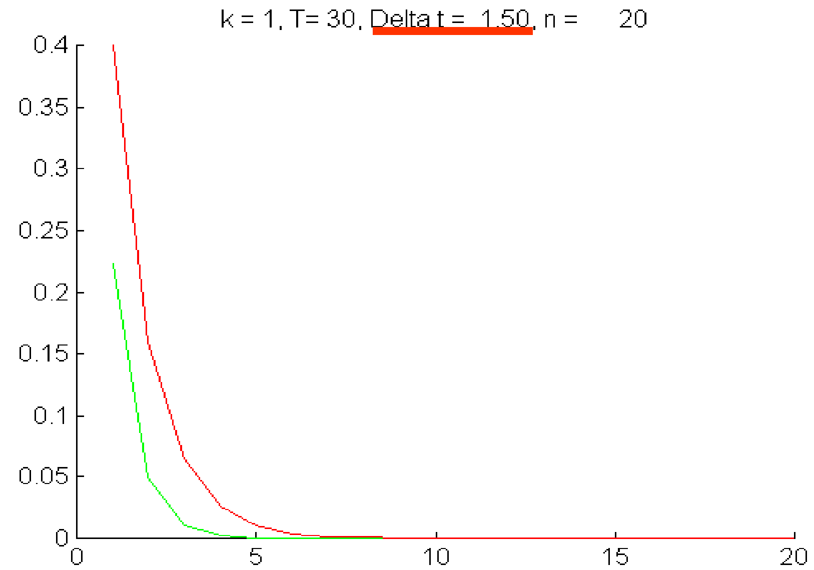
$$\frac{x_{n+1} - x_n}{\Delta t} = -kx_{n+1}$$

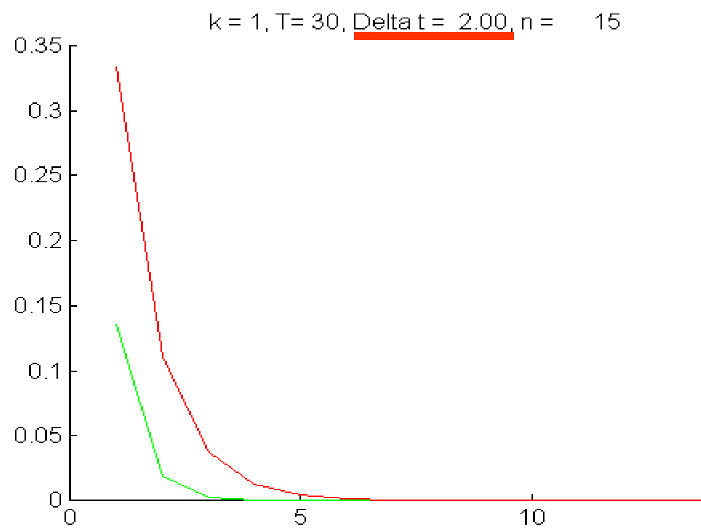
$$x_{n+1} + k\Delta t x_{n+1} = x_n$$

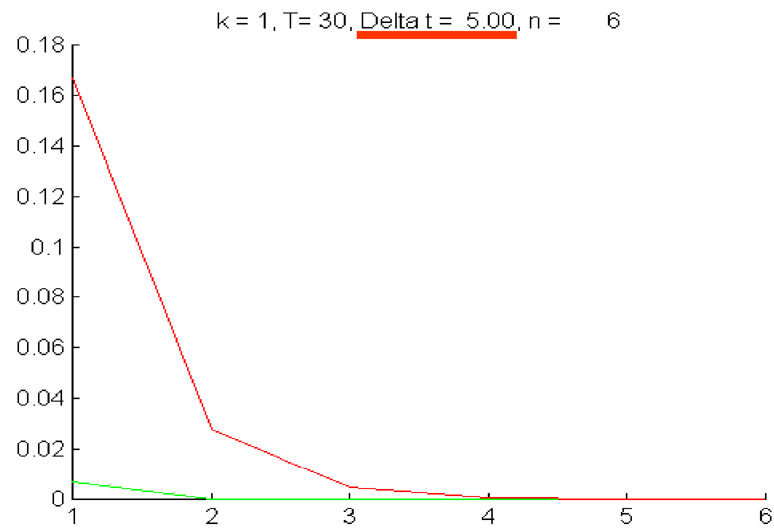
$$x_{n+1} = \frac{1}{1 + k\Delta t} x_n$$

$$\dot{x} = -x \qquad x_n = \frac{1}{(1+\Delta t)^n} x_o \qquad \text{30 - second simulation}$$









Observations (Implicit Euler)

- Accuracy depends on step size
- Absolute stability, regardless of the step size
- A magic bullet for many "stiff" system
- More complicated (not necessarily direct evaluation), try $f = ax^2 + bx + c$ or
  $f = a\cos x + b\sin x$
- Linearization maybe needed
- Expensive, seldom used in the real world (think about system of equations)

(Modified) Euler Method

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

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

$$\frac{x_{n+1} - x_n}{\Delta t} = \frac{1}{2}(f(x_{n+1}, t_{n+1}) + f(x_n, t_n))$$

$$x_{n+1} - \frac{\Delta t}{2} f(x_{n+1}, t_{n+1}) = x_n + \frac{\Delta t}{2} f(x_n, t_n)$$

$$\dot{x} = -kx$$

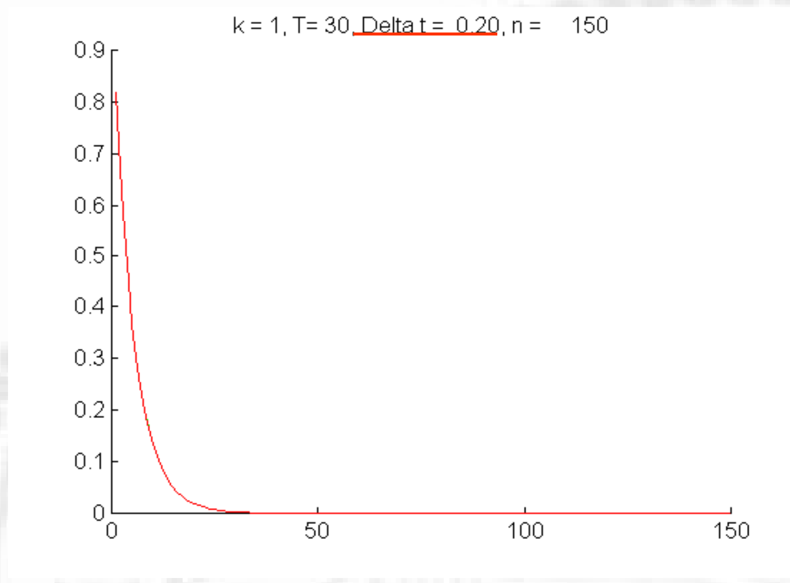
$$\frac{dx}{dt} = -kx$$

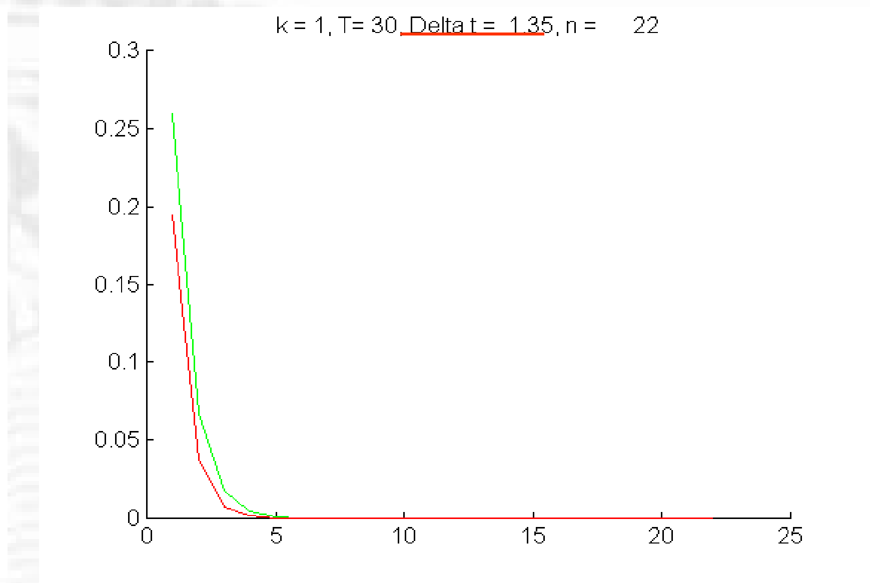
$$\frac{x_{n+1} - x_n}{\Delta t} = -\frac{k}{2}(x_{n+1} + x_n)$$

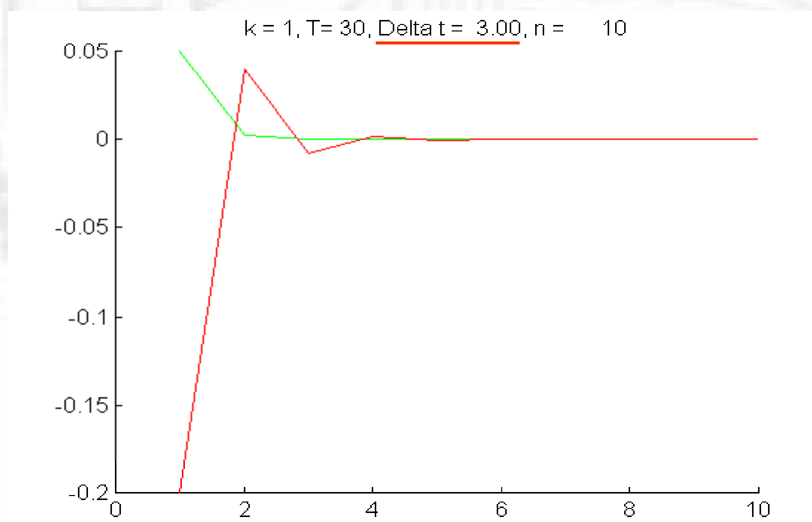
$$x_{n+1} + \frac{k}{2}\Delta t x_{n+1} = x_n - \frac{k}{2} x_n$$

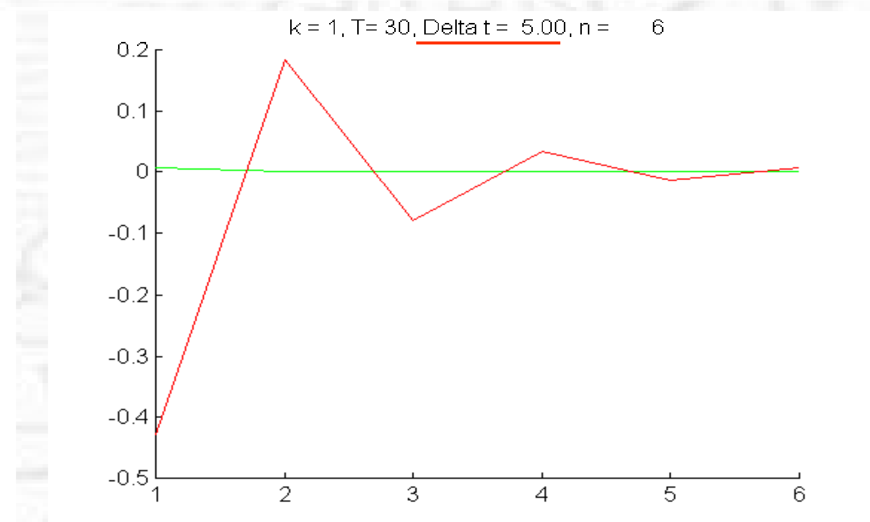
$$x_{n+1} = \frac{1 - \frac{k}{2}\Delta t}{1 + \frac{k}{2}\Delta t} x_n$$

$$\dot{x} = -x \qquad x_n = \frac{(1-0.5\Delta t)^n}{(1+0.5\Delta t)^n} x_o \qquad \text{30 - second simulation}$$









Observations (Modified Euler)

- Use trapezoidal rule
- Accuracy depends on step size (Surprisingly: it is more accurate than either explicit or implicit method)
- *Absolute* stability, *regardless of* the step size
- More complicated (not necessarily direct evaluation)
- Basis for other advanced techniques

Accuracy of Euler Methods

$$x(t+\Delta t) = x(t) + \dot{x}(t)\Delta t + \ddot{x}(t)\frac{\Delta t^2}{2!} + \dddot{x}(t)\frac{\Delta t^3}{3!} + \cdots$$

$\frac{d}{dt}$

$$\text{Explicit Euler}: x(t+\Delta t) = x(t) + \dot{x}(t)\Delta t$$

$$Error: O(\Delta t^2)$$

$$\text{Implicit Euler}: x(t+\Delta t) = x(t) + \dot{x}(t+\Delta t)\Delta t$$

$$\dot{x}(t+\Delta t) = \dot{x}(t) + \ddot{x}(t)\Delta t + \dddot{x}(t)\frac{\Delta t^2}{2!} + \cdots$$

$$x(t+\Delta t) = x(t) + \dot{x}(t)\Delta t + \ddot{x}(t)\Delta t^2 + \dddot{x}(t)\frac{\Delta t^3}{2!} + \cdots$$

$$Error: O(\Delta t^2)$$

Accuracy of Euler Methods (cont.)

$$\text{Modified Euler}: x(t+\Delta t) = x(t) + \frac{1}{2}(\dot{x}(t) + \dot{x}(t+\Delta t))\Delta t$$

$$\dot{x}(t+\Delta t) = \dot{x}(t) + \ddot{x}(t)\Delta t + \dddot{x}(t)\frac{\Delta t^2}{2!} + \cdots$$

$$x(t+\Delta t) = x(t) + \frac{1}{2}\dot{x}(t)\Delta t + \frac{1}{2}\{\dot{x}(t)\Delta t + \ddot{x}(t)\Delta t^2 + \dddot{x}(t)\frac{\Delta t^3}{2!} + \cdots\}$$

$$= x(t) + \dot{x}(t)\Delta t + \frac{1}{2}\ddot{x}(t)\Delta t^2 + \frac{1}{2}\dddot{x}(t)\frac{\Delta t^3}{2!} + \cdots$$

$$Error: O(\Delta t^3)$$

Why Implicit Methods Don't Blow Up?

$\dot{x} = -kx, k > 0$

x

t

Δt

$\dot{x} = -kx$

t

Δt

t

Δt

Stability=(?)Accuracy

- In general, you can get a stable solution by using a reasonably small step size
- To get high accuracy, step size needs to be very small (with dumb methods)
- There are other techniques to increase accuracy at large step size

Why not Higher Orders?

- If
  - the goal is to reduce the *local* approximation error
  - Local approximation error is expressed in terms of Taylor series
- Then
  - Why not compute more terms in the expansion?

$$x(t+\Delta t) = x(t) + \dot{x}(t)\Delta t + \ddot{x}(t)\frac{\Delta t^2}{2!} + \dddot{x}(t)\frac{\Delta t^3}{3!} + \cdots$$

$$x(t+\Delta t) = x(t) + f\Delta t + f'\frac{\Delta t^2}{2!} + f''\frac{\Delta t^3}{3!} + \cdots$$

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

Answer

- Only $f(x,t)$ (*or* x') is given explicitly
- $f'(x,t)$, etc (*or* x'', x''') can be hard to evaluate
- There are other methods to match the terms in Taylor series to achieve high-order of accuracy
  - Single-step, Compounding methods
  - Mutli-step methods

Midpoint Method (2nd order Runge-Kutta)

- Compound one-step method
- *Explicit* for easy computing
- 2nd order is very similar to trapezoidal rule
- Caution: there are a number of 2nd order Runge-Kutta formulation, all with the same error bound

Trapezoidal

- Modified Euler

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

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

$$\frac{x_{n+1} - x_n}{\Delta t} = \frac{1}{2}\left(f(x_n,t_n) + f(x_{n+1},t_{n+1})\right)$$

Implicit!

- Trapezoidal – similar to Modified Euler, but practical

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

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

$$\frac{x_{n+1} - x_n}{\Delta t} = \frac{1}{2}\left(f(x_n,t_n) + f(x_n + \Delta t f(x_n,t_n),t_{n+1})\right)$$

$$x_{n+1} = x_n + \Delta t f(x_n,t_n)$$

Explicit!

Example

$$\dot{x} = -kx$$

$$\frac{dx}{dt} = -kx$$

$$\frac{x_{n+1} - x_n}{\Delta t} = \frac{1}{2}(f(x_n, t_n) + f(x_n + \Delta t f(x_n, t_n), t_{n+1}))$$

$$\frac{x_{n+1} - x_n}{\Delta t} = -\frac{1}{2}(kx_n + k(x_n + \Delta t k x_n))$$

$$x_{n+1} = x_n - \Delta t k x_n - \frac{1}{2}\Delta t^2 k^2 x_n = (1 - \Delta t k - \frac{1}{2}\Delta t^2 k^2) x_n$$

Facts

- $O(\Delta t^3)$ accuracy
  - Explicit, easy to compute, but not absolute stable
- Other compounding methods possible
- Compound can be done many many times, the most famous is the 4th order Runge-Kutta with $O(\Delta t^5)$

General Compounding Methods

- Evaluate the expression *f(x,t) multiple* times in a *single* step
  - At *different* location (x) and *different* time (t)
  - But all *explicit*
- *2nd* order (midpoint) actually corresponds to determine a, b,c in the following expression
  - Evaluate the derivative at different place (b), different time (c), and different weighting (a)

$$\dot{x} = af(x+b, t+c) \qquad cf \quad \dot{x} = f(x,t) \text{ for Explict Euler}$$

Matching Process

$$x_{n+1} = x_n + \Delta t x_n' + \frac{\Delta t^2}{2!} x_n'' + \cdots$$

$$\dot{x} = af(x+b, t+c)$$

$$x_{n+1} = x_n + \Delta t\, f(x_n, t_n) + \frac{\Delta t^2}{2!} f'(x_n, t_n) + \cdots$$

$$= x_n + \Delta t\, f(x_n, t_n) + \frac{\Delta t^2}{2!}\left(\frac{\partial f(x_n, t_n)}{\partial t} + \frac{\partial f(x_n, t_n)}{\partial x}\frac{\partial x}{\partial t}\right) + \cdots$$

$$= x_n + \Delta t\, f(x_n, t_n) + \frac{\Delta t^2}{2!}\left(\frac{\partial f(x_n, t_n)}{\partial t} + \frac{\partial f(x_n, t_n)}{\partial x} f(x_n, t_n)\right) + \cdots$$

$$= x_n + \Delta t\, f(x_n, t_n) + \frac{\Delta t^2}{2!}\frac{\partial f(x_n, t_n)}{\partial t} + \frac{\Delta t^2}{2!}\frac{\partial f(x_n, t_n)}{\partial x} f(x_n, t_n)$$

$$x_{n+1} = x_n + a\Delta t f(x+b, t+c)$$

$$x_{n+1} = x_n + a\Delta t[f(x,t) + \frac{\partial f}{\partial x} b + \frac{\partial f}{\partial t} c + \cdots]$$

$$x_{n+1} = x_n + \Delta t a f(x,t) + \Delta t \frac{\partial f}{\partial x} ab + \Delta t \frac{\partial f}{\partial t} ac + \cdots]$$

Matching Process

$$x_{n+1} = x_n + \Delta t\, f(x_n, t_n) + \frac{\Delta t^2}{2!}\frac{\partial f(x_n, t_n)}{\partial t} + \frac{\Delta t^2}{2!}\frac{\partial f(x_n, t_n)}{\partial x} f(x_n, t_n)$$

$$c = \frac{\Delta t}{2}$$

$$a = 1$$

$$b = \frac{\Delta t}{2} f(x_n, t_n)$$

$$f(x + \frac{\Delta t}{2} f, t + \frac{\Delta t}{2})$$

$$x_{n+1} = x_n + \Delta t a f(x,t) + \Delta t \frac{\partial f}{\partial x} ab + \Delta t \frac{\partial f}{\partial t} ac + \cdots]$$

$$f(x_n, t_n)$$

Mid-point methods

$$\dot{x} = af(x + b, t + c)$$

$$= f(x + \frac{\Delta t}{2} f, t + \frac{\Delta t}{2})$$

General Compound Methods

- Successive recursions (many more times)
- 4th-order is the most popular

$$\dot{x} = af(x+b, t+c)$$
$$= f(x + \frac{\Delta t}{2} f, t + \frac{\Delta t}{2})$$

$$f_4 = f(x + \Delta t f_3, t + \Delta t)$$
$$f_3 = \Delta t f(x + \frac{\Delta t}{2} f_2, t + \frac{\Delta t}{2})$$
$$f_2 = \Delta t f(x + \frac{\Delta t}{2} f_1, t + \frac{\Delta t}{2})$$
$$f_1 = \Delta t f(x, t)$$
$$x_{n+1} = x_n + \frac{\Delta t}{6}(f_1 + 2f_2 + 2f_3 + f_4)$$

One evaluation on the left

Two evaluations in the middle

One evaluation on the right

Multi-Step Methods

- Utilize multiple *past* and *present* values instead of just one (at time *n-1*)
- Explicit, easy to compute
- Implicit also possible for stability
- Can be made into arbitrary orders of accuracy (using enough terms)
- For smooth fields (*f'*, *f''*, etc. should exist)
- Compare to

Comparison

- Compound methods
- *Multiple* evaluations of f in a single step
- Do not use past values
- Do not assume smoothness over time
- Good accuracy (4th order)

- Multi-step methods
- *Single* evaluation of f in a single step
- Use past values of f
- Work best with smooth fields (f', f'', f''', etc. exist)
- Very high accuracy

Multi-Step Methods

- Explicit (Adams-Bashforth)

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

f at previous times

Already calculated, might as well use them

$$\frac{x_{n+1} - x_n}{\Delta t} = c_1 f_n + c_2 f_{n-1} + \ldots + c_p f_{n-p+1}$$

- Implicit (Adams-Moulton)

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

$$\frac{x_{n+1} - x_n}{\Delta t} = \boxed{c_0 f_{n+1}} + c_1 f_n + c_2 f_{n-1} + \ldots + c_p f_{n-p+1}$$

- Q: What should the coefficient c_p be?
- A: Chosen to match as many terms in the Taylor expansion as possible

Example - Explicit, 2nd order

$$f_{n-1} = f_n - f_n'\Delta t + f_n''\frac{\Delta t^2}{2!} - \cdots$$

$$\frac{x_{n+1} - x_n}{\Delta t} = \alpha f(x_n, t_n) + \beta f(x_{n-1}, t_{n-1})$$

$$x_{n+1} = x_n + \alpha \Delta t f_n + \beta \Delta t \{ f_n - f'_n \Delta t + f_n''\frac{\Delta t^2}{2!} - \cdots \}$$

$$x_{n+1} = x_n + (\alpha + \beta) f_n \Delta t - \beta f_n' \Delta t^2 + \beta f_n''\frac{\Delta t^3}{2!} - \cdots$$

$$Error : O(\Delta t^3)$$

$$\alpha + \beta = 1$$

$$\beta = -\frac{1}{2}$$

Observation:Multi-Steps

- Again
  - implicit methods afford high accuracy with good stability
  - the cost is again in evaluating the extra term in the implicit methods (pretty difficult if f is not linear!)
  - Good for smooth functions requiring high accuracy

Explicit

	c_1	c_2	c_3	c_4	*stability threshold*	*error constant*
$O(\Delta t^2)$	1				-2	$1/2$
$O(\Delta t^3)$	$3/2$	$-1/2$			-1	$5/12$
$O(\Delta t^4)$	$23/12$	$-16/12$	$5/12$		$-6/11$	$3/8$
$O(\Delta t^5)$	$55/24$	$-59/24$	$37/24$	$-9/24$	$-3/10$	$251/720$

$$-k\Delta t \geq \textit{stability threhold}$$

$$\Delta t \leq \frac{\textit{stability threhold}}{-k}$$

Predictor-Corrector Method

- Use the explicit formula to *predict* a new x_{n+1}^*

$$x_{n+1}^* = x_n + \Delta t(c_1 f_n + c_2 f_{n-1} + \ldots + c_p f_{n-p+1})$$

- Use x_{n+1}^* to evaluate the right side

$$f_{n+1}^* = f(t_{n+1}, x_{n+1}^*)$$

Use f_{n+1}^* in the implicit formula to correct the new x_{n+1}

$$x_{n+1} = x_n + \Delta t(c_0 f_{n+1}^* + c_1 f_n + c_2 f_{n-1} + \ldots + c_p f_{n-p+1})$$

Error Sources

- So far we concentrated on truncation error – i.e., we didn't use enough terms in Taylor series
- Another source is numerical (10^-16 for double due to limited machine precision)

$$E = E_{round} + E_{truncate}$$

- Eround: error in calculating derivatives
- Etruncate: omission in Taylor expansion

Error Sources

- Reduce step size
  - Truncation error becomes smaller
  - Numerical error accumulates faster from many steps
  - Simulation becomes smaller
- Increase step size
  - Truncation error becomes larger
  - Numerical error accumulates slower from fewer steps
  - Simulation becomes faster

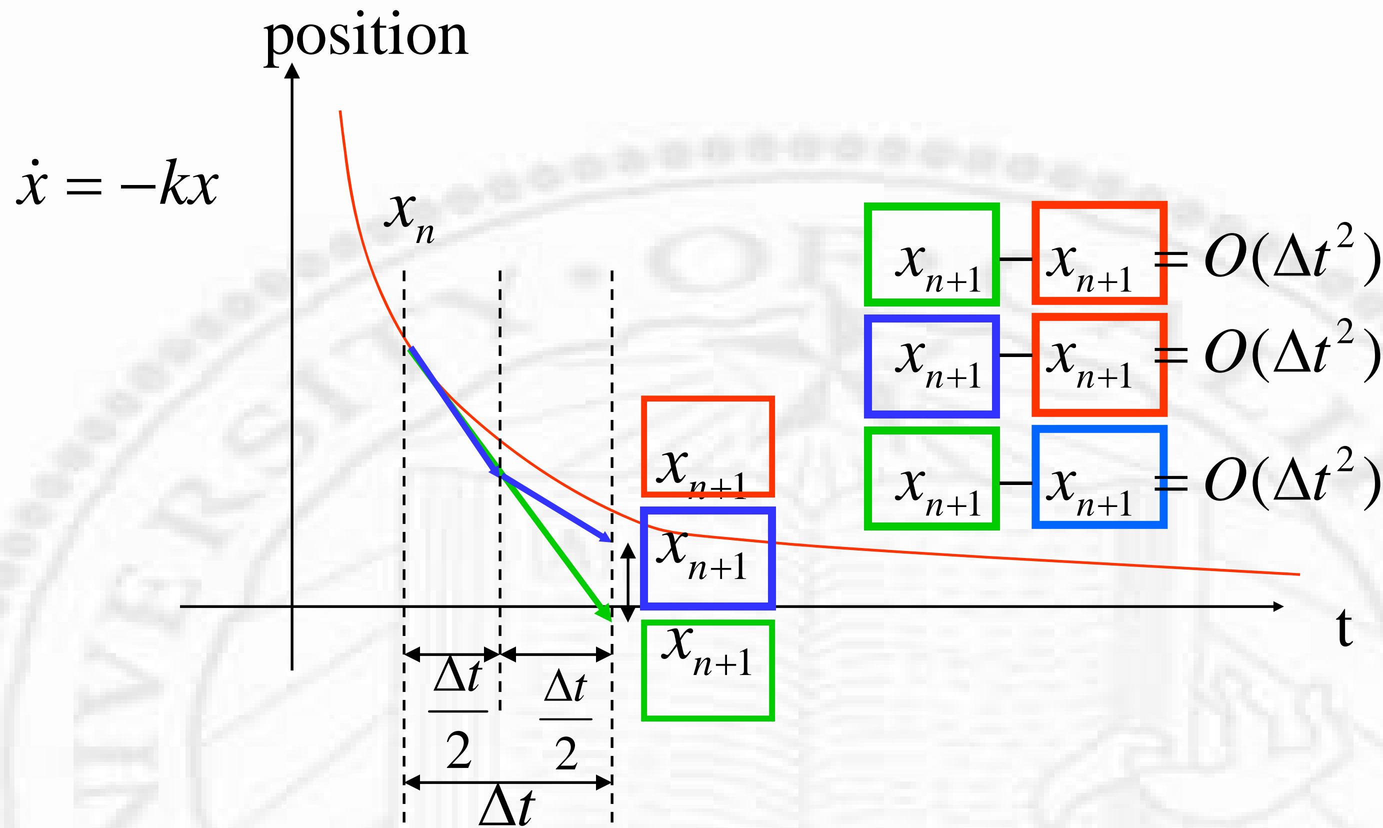
$$E = E_{round} + E_{truncate}$$

Error Sources (cont)

- Obviously, some trade-off exists with an optimial step size ~(Eround/max(Etruncate))^0.5
- Easier said than done
  - How to find Etruncate
  - Etruncate is not a constant over the whole domain
- Adaptive step size is the key

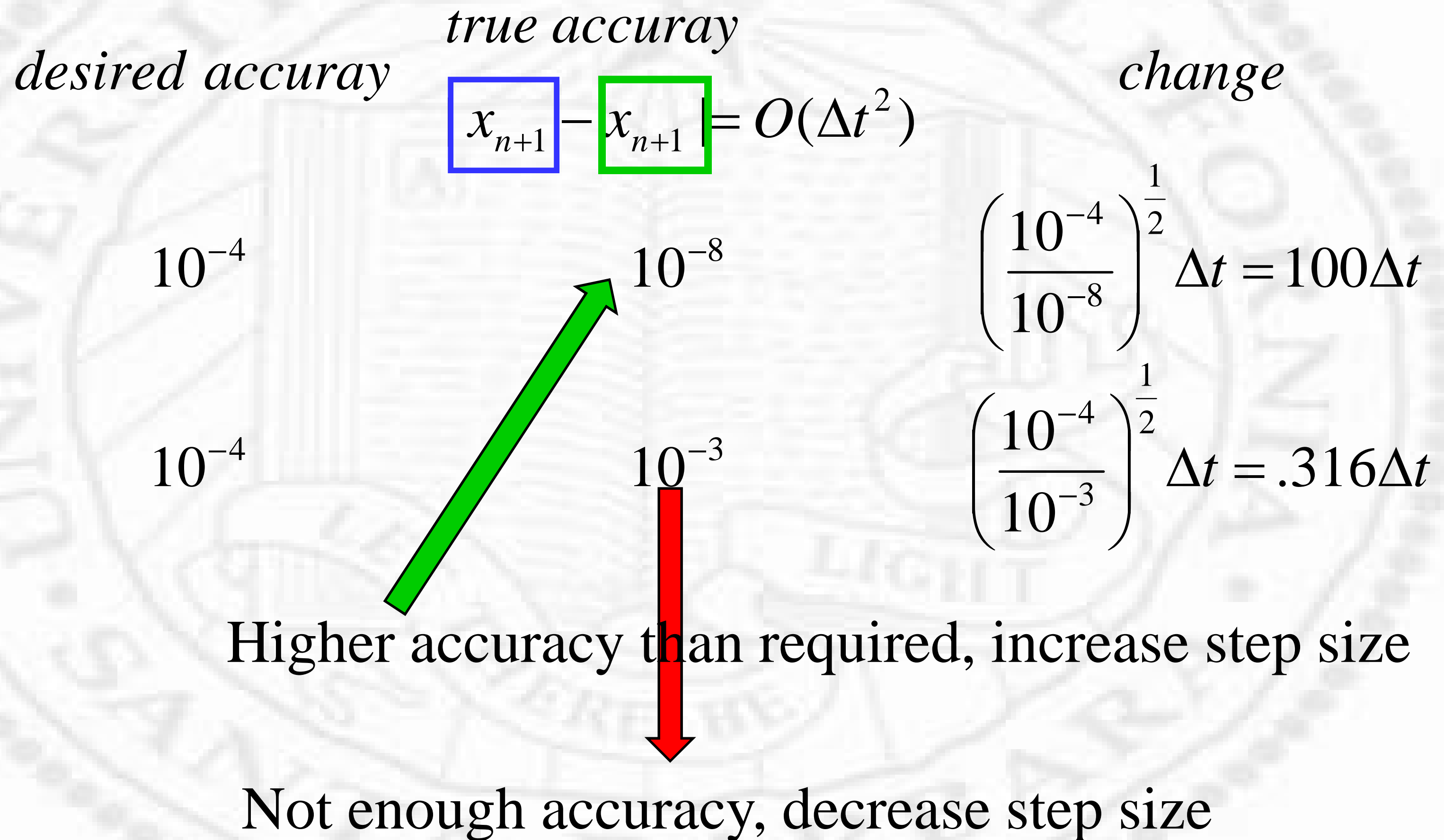
Adaptive Step Size

- To speed up simulation needs large step size
- To maintain stability and improve accuracy need small step size
- The step size should be adapted depending on the required accuracy



- Error can be estimated from the difference of two Euler steps
- Steps can be enlarged or shrunk based on the desired accuracy

Adaptive step sizes



desired accuray	*true accuray* $|x_{n+1} - x_{n+1}| = O(\Delta t^2)$	*change*
10^{-4}	10^{-8}	$\left(\frac{10^{-4}}{10^{-8}}\right)^{\frac{1}{2}} \Delta t = 100\Delta t$
10^{-4}	10^{-3}	$\left(\frac{10^{-4}}{10^{-3}}\right)^{\frac{1}{2}} \Delta t = .316\Delta t$

Higher accuracy than required, increase step size

Not enough accuracy, decrease step size

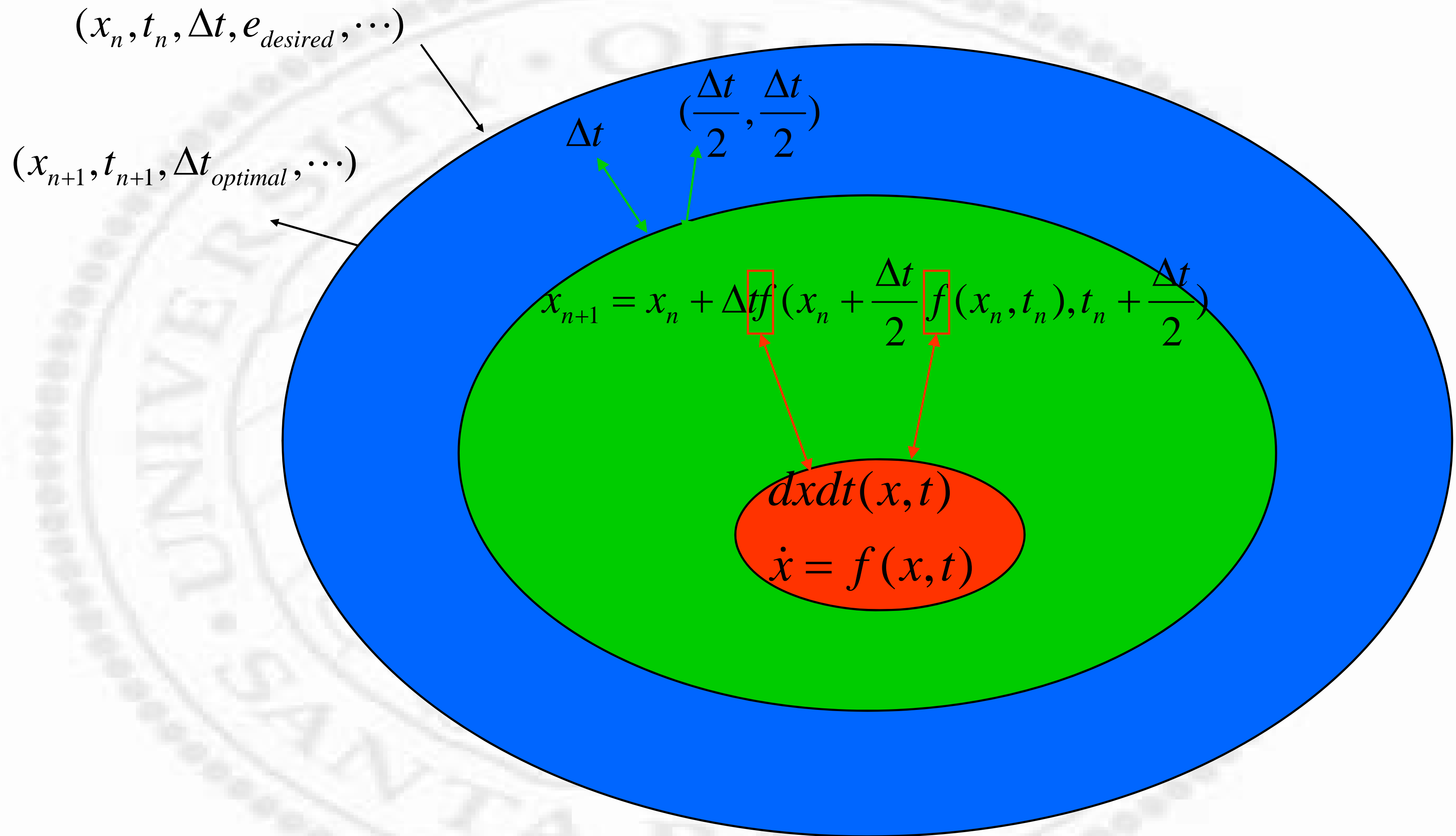
Last Word

- If you are a numerical analyst, this does not apply to you ☺
- Otherwise
  - Don't use Explicit Euler (not accurate, blowing up)
  - Don't use Implicit Euler (too expensive)
  - 2nd or 4th order Runge-Kutta with step control is for you

Solver Structure

- Heart is dxdt (a function you supply)
- Wrapped around by fixed step Euler, Midpoint, Runge-Kutta, etc.
- Wrapped around by a step size controller

Solver Structure



Example: 4th-order Runge-Kuta

```
#include "nrutil.h"

void rk4(float y[], float dydx[], int n, float x, float h, float yout[],
    void (*derivs)(float, float [], float []))
```

Given values for the variables y[1..n] and their derivatives dydx[1..n] known at x, use the fourth-order Runge-Kutta method to advance the solution over an interval h and return the incremented variables as yout[1..n], which need not be a distinct array from y. The user supplies the routine derivs(x,y,dydx), which returns derivatives dydx at x.

```
{
    int i;
    float xh,hh,h6,*dym,*dyt,*yt;

    dym=vector(1,n);
    dyt=vector(1,n);

    yt=vector(1,n);
    hh=h*0.5;
    h6=h/6.0;
    xh=x+hh;
    for (i=1;i<=n;i++) yt[i]=y[i]+hh*dydx[i];        First step.
    (*derivs)(xh,yt,dyt);                            Second step.
    for (i=1;i<=n;i++) yt[i]=y[i]+hh*dyt[i];
    (*derivs)(xh,yt,dym);                            Third step.
    for (i=1;i<=n;i++) {
        yt[i]=y[i]+h*dym[i];
        dym[i] += dyt[i];
    }
    (*derivs)(x+h,yt,dyt);                           Fourth step.
    for (i=1;i<=n;i++)                               Accumulate increments with proper
        yout[i]=y[i]+h6*(dydx[i]+dyt[i]+2.0*dym[i]);     weights.
    free_vector(yt,1,n);
    free_vector(dyt,1,n);
    free_vector(dym,1,n);
}
```

Example: RK with Step Control

```
#include <math.h>
#include "nrutil.h"
#define SAFETY 0.9
#define PGROW -0.2
#define PSHRNK -0.25
#define ERRCON 1.89e-4
```

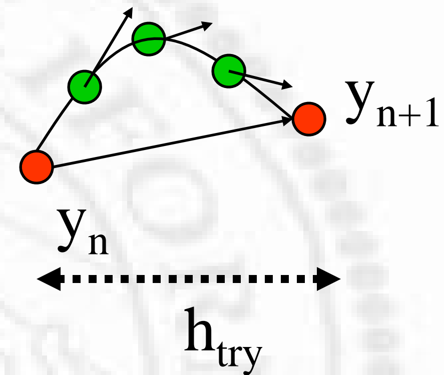
The value ERRCON equals (5/SAFETY) raised to the power (1/PGROW), see use below.

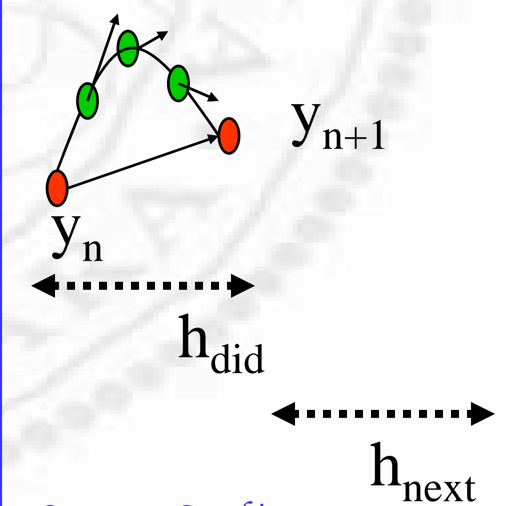
```
void rkqs(float y[], float dydx[], int n, float *x, float htry, float eps,
    float yscal[], float *hdid, float *hnext,
    void (*derivs)(float, float [], float []))
```

Fifth-order Runge-Kutta step with monitoring of local truncation error to ensure accuracy and adjust stepsize. Input are the dependent variable vector y[1..n] and its derivative dydx[1..n] at the starting value of the independent variable x. Also input are the stepsize to be attempted htry, the required accuracy eps, and the vector yscal[1..n] against which the error is scaled. On output, y and x are replaced by their new values, hdid is the stepsize that was actually accomplished, and hnext is the estimated next stepsize. derivs is the user-supplied routine that computes the right-hand side derivatives.

```
{
    void rkck(float y[], float dydx[], int n, float x, float h,
        float yout[], float yerr[], void (*derivs)(float, float [], float []));
    int i;
    float errmax,h,htemp,xnew,*yerr,*ytemp;

    yerr=vector(1,n);
    ytemp=vector(1,n);
    h=htry;                                Set stepsize to the initial trial value.
    for (;;) {
        rkck(y,dydx,n,*x,h,ytemp,yerr,derivs);        Take a step.
        errmax=0.0;                        Evaluate accuracy.
        for (i=1;i<=n;i++) errmax=FMAX(errmax,fabs(yerr[i]/yscal[i]));
        errmax /= eps;                     Scale relative to required tolerance.
        if (errmax <= 1.0) break;          Step succeeded. Compute size of next step.
        htemp=SAFETY*h*pow(errmax,PSHRNK);
        Truncation error too large, reduce stepsize.
        h=(h >= 0.0 ? FMAX(htemp,0.1*h) : FMIN(htemp,0.1*h));
        No more than a factor of 10.
        xnew=(*x)+h;
        if (xnew == *x) nrerror("stepsize underflow in rkqs");
    }
    if (errmax > ERRCON) *hnext=SAFETY*h*pow(errmax,PGROW);
    else *hnext=5.0*h;                     No more than a factor of 5 increase.
    *x += (*hdid=h);
    for (i=1;i<=n;i++) y[i]=ytemp[i];
    free_vector(ytemp,1,n);
    free_vector(yerr,1,n);
}
```



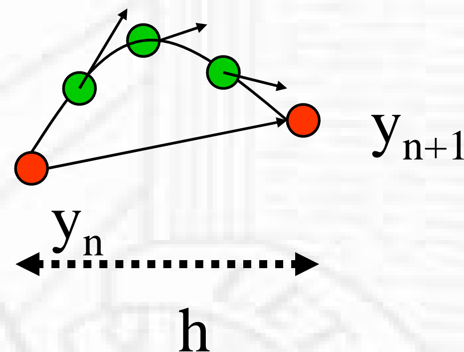


Example: Other Options

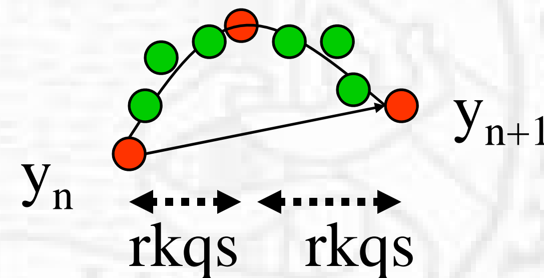
```
void rkck(float y[], float dydx[], int n, float x, float h, float yout[],
    float yerr[], void (*derivs)(float, float [], float []))
```

Given values for `n` variables `y[1..n]` and their derivatives `dydx[1..n]` known at `x`, use the fifth-order Cash-Karp Runge-Kutta method to advance the solution over an interval `h` and return the incremented variables as `yout[1..n]`. Also return an estimate of the local truncation error in `yout` using the embedded fourth-order method. The user supplies the routine `derivs(x,y,dydx)`, which returns derivatives `dydx` at `x`.

Final results



In reality



- Will advance by *h* (may take multiple steps, or multiple calls to rkqs)

Example: Other Options

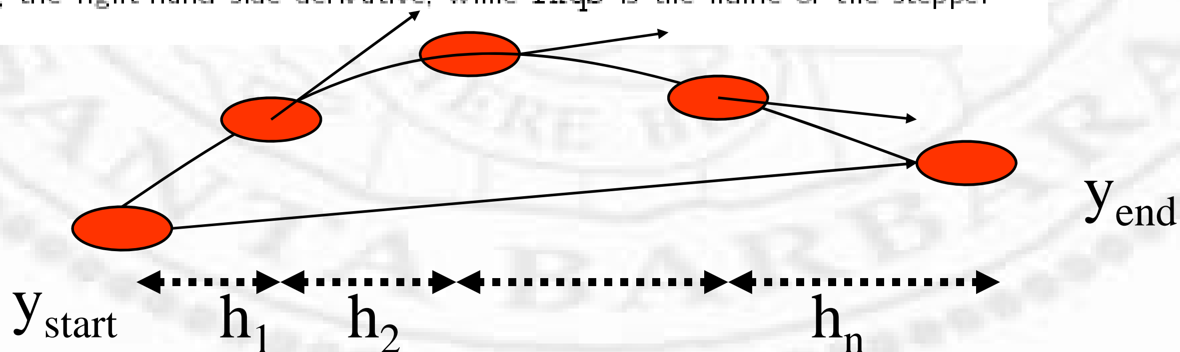
```
#include <math.h>
#include "nrutil.h"
#define MAXSTP 10000
#define TINY 1.0e-30

extern int kmax,kount;
extern float *xp,**yp,dxsav;
```

User storage for intermediate results. Preset `kmax` and `dxsav` in the calling program. If `kmax` $\neq$ 0 results are stored at approximate intervals `dxsav` in the arrays `xp[1..kount]`, `yp[1..nvar][1..kount]`, where `kount` is output by `odeint`. Defining declarations for these variables, with memory allocations `xp[1..kmax]` and `yp[1..nvar][1..kmax]` for the arrays, should be in the calling program.

```
void odeint(float ystart[], int nvar, float x1, float x2, float eps, float h1,
    float hmin, int *nok, int *nbad,
    void (*derivs)(float, float [], float []),
    void (*rkqs)(float [], float [], int, float *, float, float, float [],
    float *, float *, void (*)(float, float [], float [])))
```

Runge-Kutta driver with adaptive stepsize control. Integrate starting values `ystart[1..nvar]` from `x1` to `x2` with accuracy `eps`, storing intermediate results in global variables. `h1` should be set as a guessed first stepsize, `hmin` as the minimum allowed stepsize (can be zero). On output `nok` and `nbad` are the number of good and bad (but retried and fixed) steps taken, and `ystart` is replaced by values at the end of the integration interval. `derivs` is the user-supplied routine for calculating the right-hand side derivative, while `rkqs` is the name of the stepper routine to be used.



Stiff Systems

- Nightmare in numerical solutions
- Need small steps to maintain numerical stability
- But the interesting phenomena are really at large time scale
- Take a long time vs. blowing up
- May need implicit methods to achieve fast simulation without blowing up

Example

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} x_n \\ y_n \end{bmatrix} = \begin{bmatrix} x_{n-1} \\ y_{n-1} \end{bmatrix} + \Delta t \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x_{n-1} \\ y_{n-1} \end{bmatrix} = (I + \Delta t A) \begin{bmatrix} x_{n-1} \\ y_{n-1} \end{bmatrix}$$

$$= \cdots = (I + \Delta t A)^n \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$$

If A has eigen vect ors (u_1, u_2), and eigen valu es (λ_1, λ_2)

$\Rightarrow I + \Delta t A$ has eigen vect ors (u_1, u_2), and eigen valu es $(1 + \Delta t \lambda_1, 1 + \Delta t \lambda_2)$

$$\Rightarrow \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} = c_1 u_1 + c_2 u_2$$

$$\Rightarrow \begin{bmatrix} x_n \\ y_n \end{bmatrix} = (I + \Delta t A)^n (c_1 u_1 + c_2 u_2) = c_1 (1 + \Delta t \lambda_1)^n u_1 + c_2 (1 + \Delta t \lambda_2)^n u_2$$

Observations: stiff systems

- Whether the simulation blows up depends on

$$|(1+\Delta t\lambda_i)^n| \le 1$$

- If eigen values are of vast distinct sizes, a stiff system results
- The most negative eigen value determines the speed of simulation
- It is also the part of the simulation that quickly dies out and not of interest

Try this!

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- First order
  - convergent (e.g., a=d=-5, c=d=0)
  - orbiting (a rotation, e.g. a=d=0, b=2,d=-2)
  - spiral - combination of the above

$$\begin{bmatrix} \ddot{x} \\ \ddot{y} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- Second order

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta - 1 & -\sin\theta \\ \sin\theta & \cos\theta - 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \cong \begin{bmatrix} 0 & -\theta \\ \theta & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

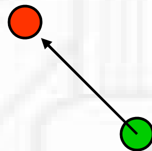
- What is the difference between 1st and 2nd order simulations?

1st order

specify location directly

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = f(x, y)$$

$$or \begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} x_n \\ y_n \end{bmatrix} + \Delta t f(x, y)$$

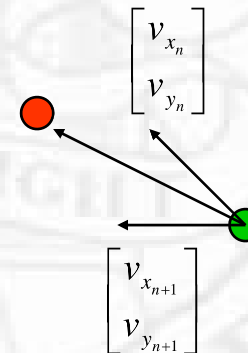


2nd order

specify velocity instead of location

$$\begin{bmatrix} \dot{v}_x \\ \dot{v}_y \end{bmatrix} = f(x, y)$$

$$or \begin{bmatrix} v_{x_{n+1}} \\ v_{y_{n+1}} \end{bmatrix} = \begin{bmatrix} v_{x_n} \\ v_{y_n} \end{bmatrix} + \Delta t f(x, y)$$



- A simulation with $a=d=-5$, $b=c=0$
  - can be free of oscillation if step size is properly chosen
  - cannot be made oscillation free without a damping term