

Interpolation, Integration, and ODE's

Computing for Astro Grad Students

Outline

- Interpolation - splines, Hermite, polynomial
- Integration - Simpson's rule, etc.
- Integration of initial value ODE's - Runge Kutta, Bulirsch-Stoer...
- Stiff equations
- Random points (boundary value, conservative...)

Interpolation

- We have a function at some set of points. What's it doing between the points?
- Question: Can I evaluate my function wherever I like?
- Question: How analytic is my function?
- Question: Are my function values noisy?

You probably want to know the answers...

Function is noiseless...

General solution is to come up with some polynomial that tells us the value of y between x_n and x_{n+1} (assuming $y=f(x)$).

Additional constraints tell us how to pick *which* polynomial to use.

Usually make polynomial go through tabulated points (y_n). Then, what order polynomial do we want? Do we want to make the n^{th} derivative continuous? Requires looking at $\sim n$ nearby points. Force $y_n \leq y(x) \leq y_{n+1}$ for $x_n < x < x_{n+1}$?

All important questions only *you* can answer.

Cubic Splines - the Granddaddy

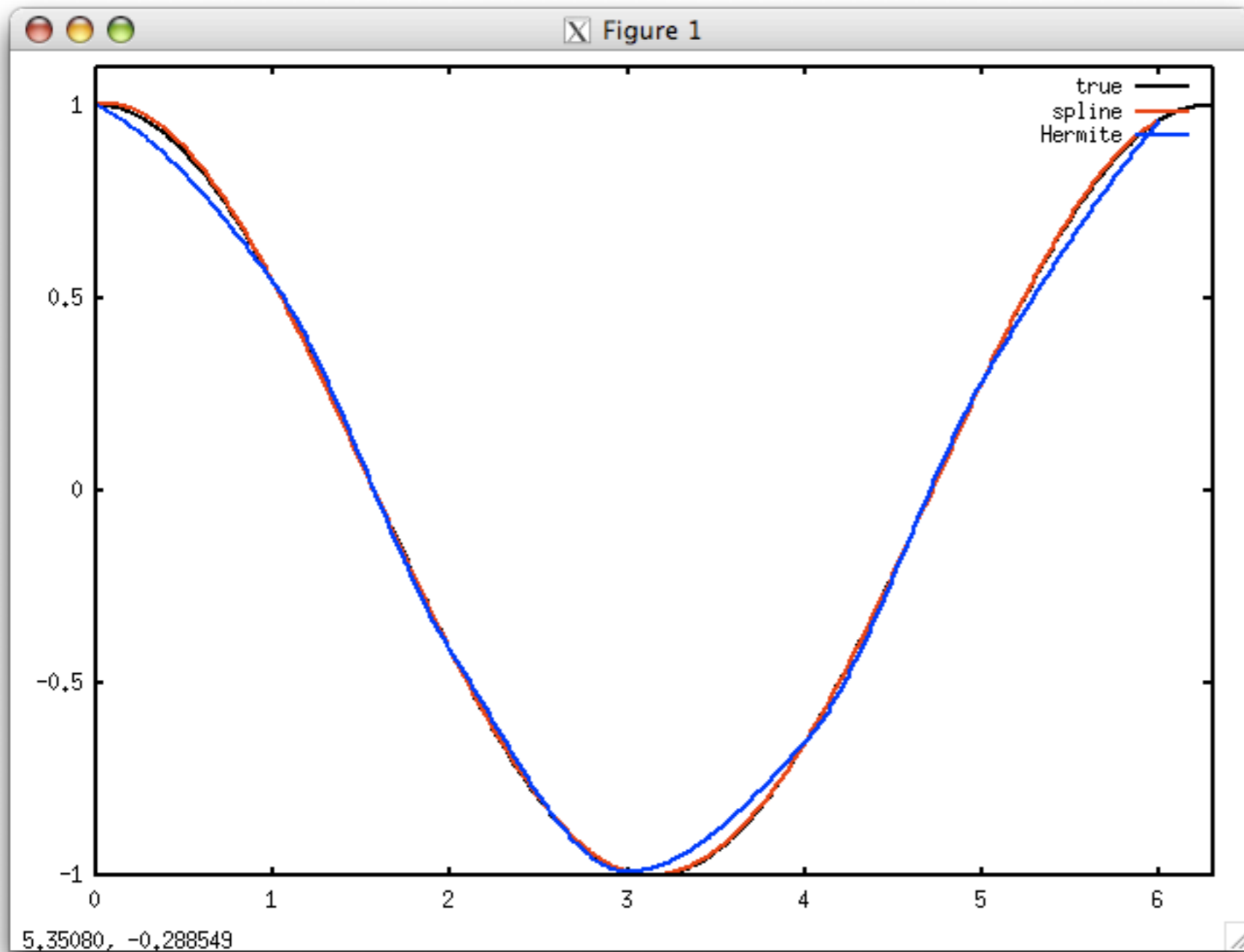
- Say we think we have a smooth function. We want to know values between points. How should we pick them?
- If function is smooth, then one choice is to say function goes through points, and that second derivative is continuous.
- Say I know (somehow) second derivative at my set of points. Then constraint that function goes through points and that second derivative varies linearly between points defines unique curve.
- How to set the second derivative? If inside region, can look at slope on left, slope on right. If on edge, must make something up. Often set $y''=0$, but not always.
- Gives tri-diagonal matrix system to get $y''(x_n)$.

Splines, Hermite

- Splines not local - continuity requirement means changing one point affects value several points away.
- Might be good if function smooth. Might be bad, too, if function not smooth.
- Another cubic choice: Hermite interpolation. If derivative changes sign, set second derivative to zero, otherwise look only at local conditions.
- Hermite bounded by values at knots (y_n), not true of splines. What you want depends on your function...

Any package you can find will have splines.
So, don't worry about details. Have an idea
what's going on, though...

Interpolation Comparison



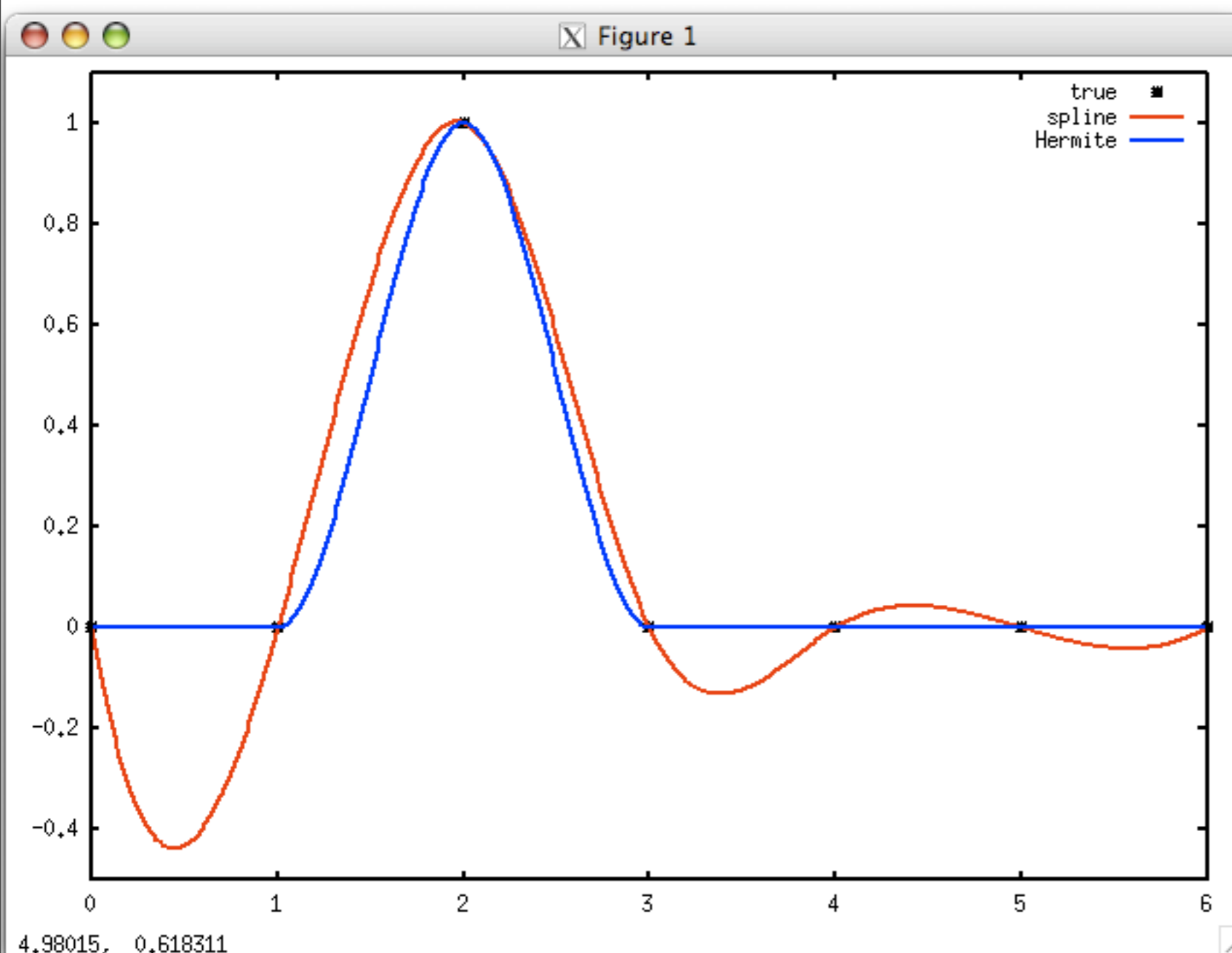
Interpolate $\cos(x)$.
Spline does an excellent job.
Hermite less so.

```
octave-3.0.0:33> x=0:1:2*pi;y=cos(x);xx=0:0.01:2*pi;  
octave-3.0.0:34> y_pchip=interp1(x,y,xx,'pchip');y_spline=interp1(x,y,xx,'spline');  
octave-3.0.0:35> clf;hold on;plot(xx,cos(xx),'k');plot(xx,y_spline,'r');plot(xx,y_pchip);  
octave-3.0.0:36> legend('true','spline','Hermite')  
octave-3.0.0:37>
```



Comparison

```
octave-3.0.0:46> x=0:6;y=0*x;y(3)=1;xx=0:0.01:max(x);
octave-3.0.0:47> y_pchip=interp1(x,y,xx,'pchip');y_spline=interp1(x,y,xx,'spline');
octave-3.0.0:48> clf;hold on;h1=plot(x,y,'k*');h2=plot(xx,y_spline,'r');h3=plot(xx,y_pchip);
octave-3.0.0:49> ll='linewidth';set(h1,ll,2);set(h2,ll,2);set(h3,ll,2);legend('true','spline','Hermite')
octave-3.0.0:50>
```



Interpolate flat data with a spike. For unsmooth functions, splines can ring when we might not want them to.

(used Octave, since Scilab doesn't have Hermite interp.)

Know thy function, and thy function shall set thee free.

- Interpolation a very common problem. There are multiple choices - the best one depends on what your function/data look like. Nobody can tell you.
- Other choices: high-order polynomial interpolation.
Rational function interpolation.
- Do you maybe want to interpolate some transform of your data, e.g. $\log(y)$? Might work better.

Data Smoothing (an aside)

- We may want to interpolate on noisy data. If errors are Gaussian, optimal solution is Kriging, as discussed previously. Drawbacks: Can be slow, have to measure variance.
- Another often good solution is to fit surrounding data with polynomial (or other form), then evaluate. If points evenly sampled, trick exists. Called Savitsky-Golay smoothing
- Of course, whole world of Fourier techniques, presumably to be covered in Fourier Transform class.

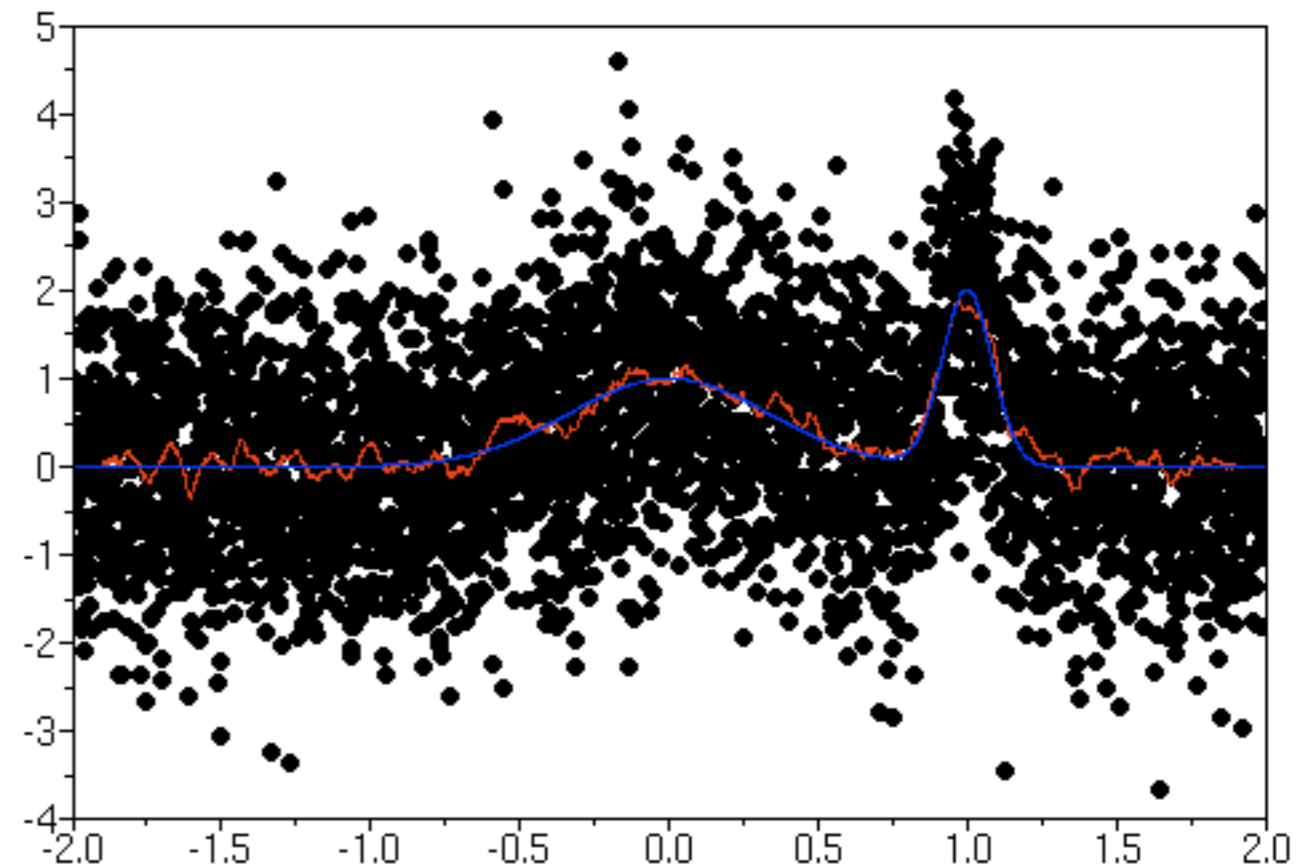
Savitsky-Golay

- Assume data is evenly spaced, constant noise.
- Do least-squares fit to chunk of data, evaluate at central point. Gives us smoothed data on which we can interpolate.
- Recall equations: $\langle y \rangle = Ap$, $p = R^{-1}Q^T y$. So, $\langle y \rangle = AR^{-1}Q^T y$. If we only want one value of y , then we pull that row out of A . Gives a vector $(A_{n,:}R^{-1}Q^T)$. Then to get smoothed value, multiply surrounding (noisy) data by vector.
- Could be efficiently evaluated with FFT's, but filter is defined in time domain.

Example:

Filtering in practice. We could have used any linear basis functions. (The evaluation can be done efficiently with FFT's. This gives us a way of generating filters with desirable time-domain properties)

```
1 function [smoothed, filtvec]=savgol(data, n, order)
2   x=(-n:n)'/n*0.999; //shrink a bit so legendre doesn't puke
3   a=legendre(0:order, 0, x)';
4   smoothed=0;
5   [q, r]=qr(a, 'e');
6   filtvec=(a(n+1, :)*inv(r))*q';
7   smoothed=0*data;
8   //quick and dirty: only do interior
9   //also, this could usually be done
10  //faster with FFTs.
11  for j=n+1:length(data)-n-1,
12    smoothed(j)=sum(data(j-n:j+n).*filtvec);
13  end
14  endfunction
```



```
-->getf('savgol.sce');
-->x=-2:0.001:2;y=exp(-0.5*x.^2*3^2)+2*exp(-0.5*(x-1).^2*12^2);
-->yy=y+rand(size(x,1),size(x,2),'normal');
-->yfit=savgol(yy,100,4);
-->clf;plot(x,yy,'k. ');plot(x,yfit,'r');plot(x,y);
```

Integration

- Want $\int y(x)$. Goal is to get to specified accuracy with as few evaluations of y as possible.
- Raises two basic questions:
- If $y(x)$ is smooth, how do we use our points to get high accuracy?
- How do we focus in on points where y is changing quickly? We need to pay more attention there.

Basic Integration: Assume have $y(x_i)$ for evenly spaced x_i .

- Very simplest integration - boxcars. $F=h\sum(y_i)$, h is x spacing.
- Next up: line. Area in one chunk is $h(y_i+y_{i+1})/2$. Gives $F/h=\sum_{i=1}^{n-1} y_i + (y_0+y_n)/2$. Same as boxcar, except for end points.
- Next (you guessed it) quadratic. If we have three points, area is $h(y_{n-1}+4y_n+y_{n+1})/6$. Add up adjacent regions, and we get $6F/h=y_0+4y_1+2y_2+4y_3+2y_4+4y_5+\dots+y_n$. (Simson's rule)
- Of course, could shift over by one half bin and average two sets. Then inside would look like $\dots+y_3+y_4+y_5+\dots$

Integrate $\exp(x)$ from 0 to 1

```
1 function[total]=boxcar_sum(y,x)
2   h=x(2)-x(1);
3   total=h*sum(y(1:$-1));
4 endfunction
5
6 function[total]=linear_sum(y,x)
7   h=x(2)-x(1);
8   total=(sum(y(2:$-1))+0.5*y(1)+0.5*y($))*h;
9 endfunction
10
11 function[total]=simpson(y,x)
12   h=x(2)-x(1);
13   total=h*(y(1)+4*sum(y(2:2:$-1))+2*sum(y(3:2:$-1))+y($))/3;
14 endfunction
```

```
-->getf('simple_int.sce');
-->disp(boxcar_sum(y,x)-%e+1)
- 0.0171255
-->disp(linear_sum(y,x)-%e+1)
0.0000573
-->disp(simpson(y,x)-%e+1)
1.5270-09
```

If function is smooth, higher order does much better. If we shifted bins by one, most coefficients would be 1 in quadratic. So, we get increasingly accurate integration by tweaking the edges? **NO!!!** We get higher order. May or may not be more accurate.

Adaptive Stepping

- Often the case that most of the work in integral is over small region of function. Can take large steps except there.
- **If you only remember one thing, remember this: know if that is happening!** If so, start integration from there. Otherwise might step right past it.
- Can do *much* less work with adaptive steps than with uniform. Alternatively, same work will give much better answers.

Adaptive Example

```
17 function [total, nval] = simpson_adaptive (func, x0, x1, h0, tol)
18     if ~exists('tol')
19         tol = 1e-8;
20     end
21     if ~exists('h0')
22         h0 = (x1 - x0) / 4;
23     end
24     xcur = x0;
25     total = 0;
26     if h0 > x1 - x0
27         h0 = x1 - x0;
28     end
29     h = h0;
30     nval = 0;
31     vec = (0:4) / 4;
32     while (xcur < x1) //core - take steps until we're done.
33         if xcur + h > x1, //if a step worked, double h
34             h = x1 - xcur; //if it failed, shrink h and try again
35         end;
36         vals = func (xcur + h * vec);
37         nval = nval + 1;
38         val1 = vals (1) + 4 * vals (3) + vals (5); //1 term simpson
39         val2 = (vals (1) + 4 * vals (2) + 2 * vals (3) + 4 * vals (4) + vals (5)) / 2;
40         if abs (val1 - val2) * h / (x1 - x0) < tol //cancel fourth order
41             total = total + h * (16 * val2 - val1) / 15 / 6;
42             xcur = xcur + h;
43             h = h * 2; //try a bigger step since we worked
44         else
45             h = h / 2; //we failed, so shrink and try again
46         end
47     end
48 endfunction
```

```
--> getf('simple_int.sce')
--> [val, niter] = simpson_adaptive(myfun, 0, 2, 0.01, 1e-5); val-3
ans =
    1.3740e-08
--> x = (0:5*niter-1); x = x*2/x($); y = myfun(x); simpson(y, x)-3
ans =
    - 0.0336552
```

Simple example: try Simpson's rule for a step and half that step. If they're close, accept the step, move on, and double h. If they're not, halve h and try again. Repeat until done. Can also use two estimates for a free fourth-order cancellation. Probably better, but no guarantee.

For same work, even sample is more than a million times worse.

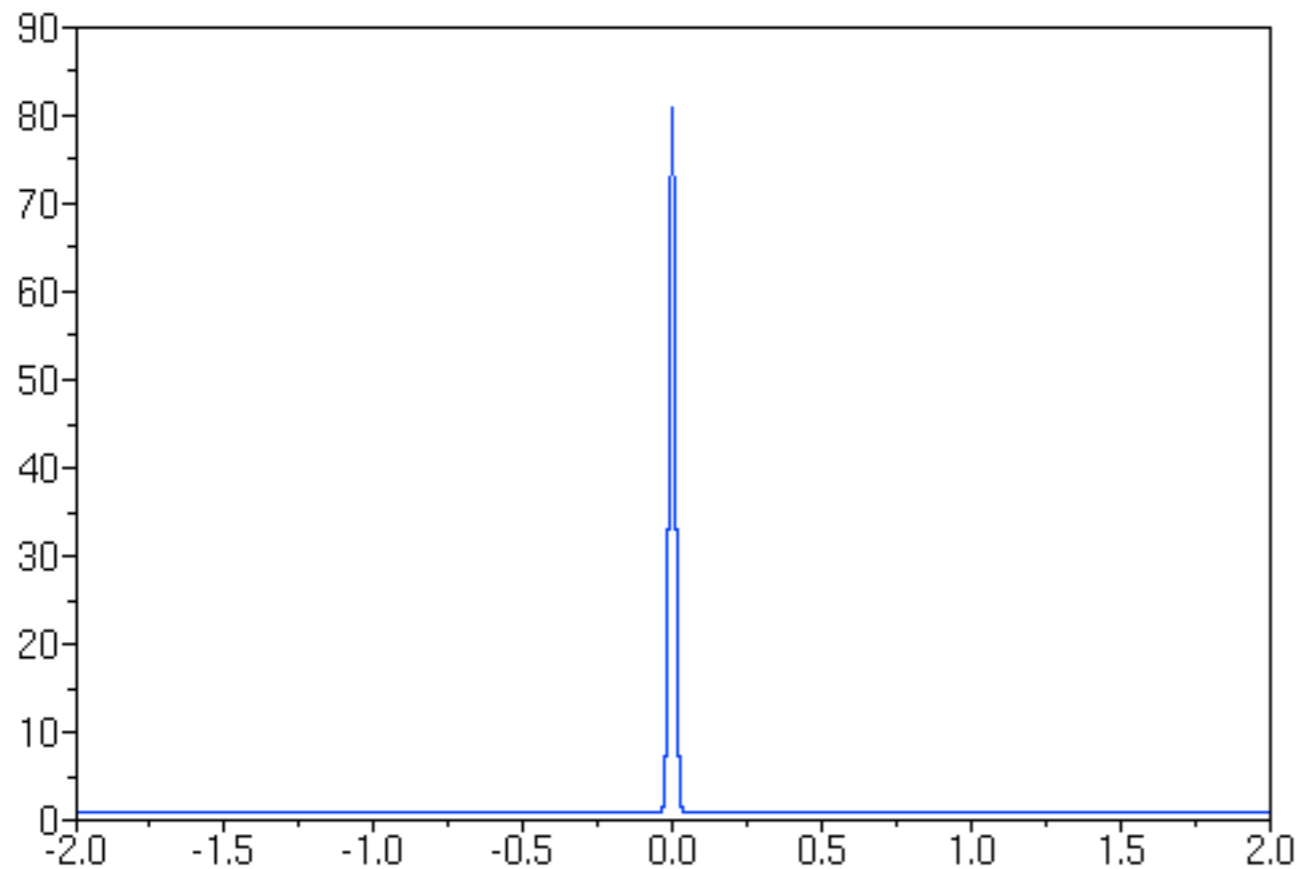
A Note on my 15/16^{ths}

Where did I pull that from? Some key points: Simpson's rule fits quadratics, so fit is accurate to second order. Third order term is odd across interval, so integrates to zero. Leaves leading order term of h^4 .

Say step gives $F + \epsilon h^4$ where F is true value and ϵ is (unknown) error. Then if I take two steps instead of 1, new value is $F + \epsilon (h/2)^4$. Eliminate ϵ and solve for F - if function is smooth, we've gotten another two orders of accuracy! Basis of many techniques for integrating ODE's.

Adaptive Can Still Fail Spectacularly!!!

```
-->val1=simpson_adaptive(myfun,-2,0,0.01,1e-5);  
-->val2=simpson_adaptive(myfun,0,2,0.01,1e-5);  
-->val=simpson_adaptive(myfun,-2,2,0.01,1e-5);  
-->[val1+val2 val]  
ans =  
    6.    4.
```



Try integral from $(-2,0) + \text{integral}(0,2)$, compare to $\text{integral}(-2,2)$, accuracy of 10^{-5} . Should be the same, right? NO! Going from $(-2,2)$ let integrator skip the spike. Putting boundary on 0 forced integrator to do the right thing. If you find integrals changing dramatically with tolerance, suspect this.

Not an arcane problem. I spent long time in grad school tracking this very bug down.

ODE's

Goal: you won't write you own ODE solver.
However, you'll have to pick the right one
someone else has written for you.

- More complicated than integrals, but many similarities.
- Integration special case of ODE: $dy/dx=f(x)$ (usually $f(x,y)$)
- In general will have to solve *systems* of equations.
- We will only talk about first order systems of equations.
Note that high order can always be transformed into more first order by adding variables: $y''=f(x,y) \rightarrow z=y'; z''=f(x,y)$

Basics

- Simplest: $y' = f(x, y)$: $y(t + \delta t) = y(t) + f(x, y) \delta t$ called Euler method. Don't ever use (analogous to boxcar integration).
- Improved Euler: take Euler step, but then evaluate derivative at new spot. Average the two and take that step. Analogous to linear integration.
- Suggestive, no? One more step and we're at equivalent of Simpson's rule. That gives us the ODE warhorse:

This is not my day job. So, if you have something to add (even if it's "Sievers is an idiot"), please do so.

Fourth-Order Runge-Kutta

- Take half a step, step back, then use the two to step to the edge. Gives us two points in middle, average them for the classic RK4 algorithm.

$$k_1 = hf(x_n, y_n) \quad (\text{Euler step})$$

$$k_2 = hf(x_n + h/2, y_n + k_1/2) \quad (\text{Where we ended up})$$

$$k_3 = hf(x_n + h/2, y_n + k_2/2) \quad (\text{Get guess at the middle})$$

$$k_4 = hf(x_n + h, y_n + k_3) \quad (\text{Step to the far side})$$

$$y_{n+1} = y_n + (k_1 + 2k_2 + 2k_3 + k_4)/6 + O(h^5).$$

RK4, cont'd

- Does many useful things *iff* hooked up with adaptive step size.
- Good place to start if you don't think your function is particularly smooth, don't know much extra information about it, and don't need huge accuracy.
- Beware of varying scales in your problem.

If You're Smooth...

- Bulirsch-Stoer: Extension of error-term canceling trick.
- Basic idea: try to map out $y_n(h)$, then take limit as h goes to zero.
- Predictor-corrector: Use information from last few steps to figure out where we should be. Adams method one implementation.
- Scilab has built-in Adams for its ode solver,

Stiff Equations.

- Take simple case: $y' = -100y$, Euler integration with fixed step size h . Should decay to zero.
- $y(t+h) = y(t) - 100hy(t) = y(t)(1 - 100h)$
- Keep going: $y(t) = y(0)(1 - 100h)^{t/h}$.
- If $100h$ is bigger than 2, then solution *grows*. Not exponential decay.
- Common situation: say I have two solutions, one of which dies very quickly. If not careful, will have to track fastest solution, even if it's 0!
- Called stiff equations. Very common, e.g. in BBNS.

Stiff, Cont'd

- Say have system: $y' = Cy$. Then Euler puts us at $y_{n+1} = y_n + hCy_n = y_n(1 + hC)$. Not gonna work if any eigenvalue more negative than $2/h$ without close tracking.
- Try instead $y_{n+1} = (1 - hC)^{-1} y_n$ - implicit differencing (what I would have gotten starting at y_{n+1} and working backwards to y_n . $y(t) = (1 - hC)^{-t/h} y(0)$. Same to first order, but look: large negative eigenvalue is large and positive in $1 - hC$, raised to a large negative power makes it go away. Stable now!
- What's that, you say? Have to track large positive eigenvalues? Well, you do have a growing solution...

Actual Useful Methods

- I'm not even going to tell you how they work. Figure out if you have stiff equations, and if so, use a stiff solver.
- To figure out if you have them, look at eigenvalues of linear expansion of coefficients. Should let you know.

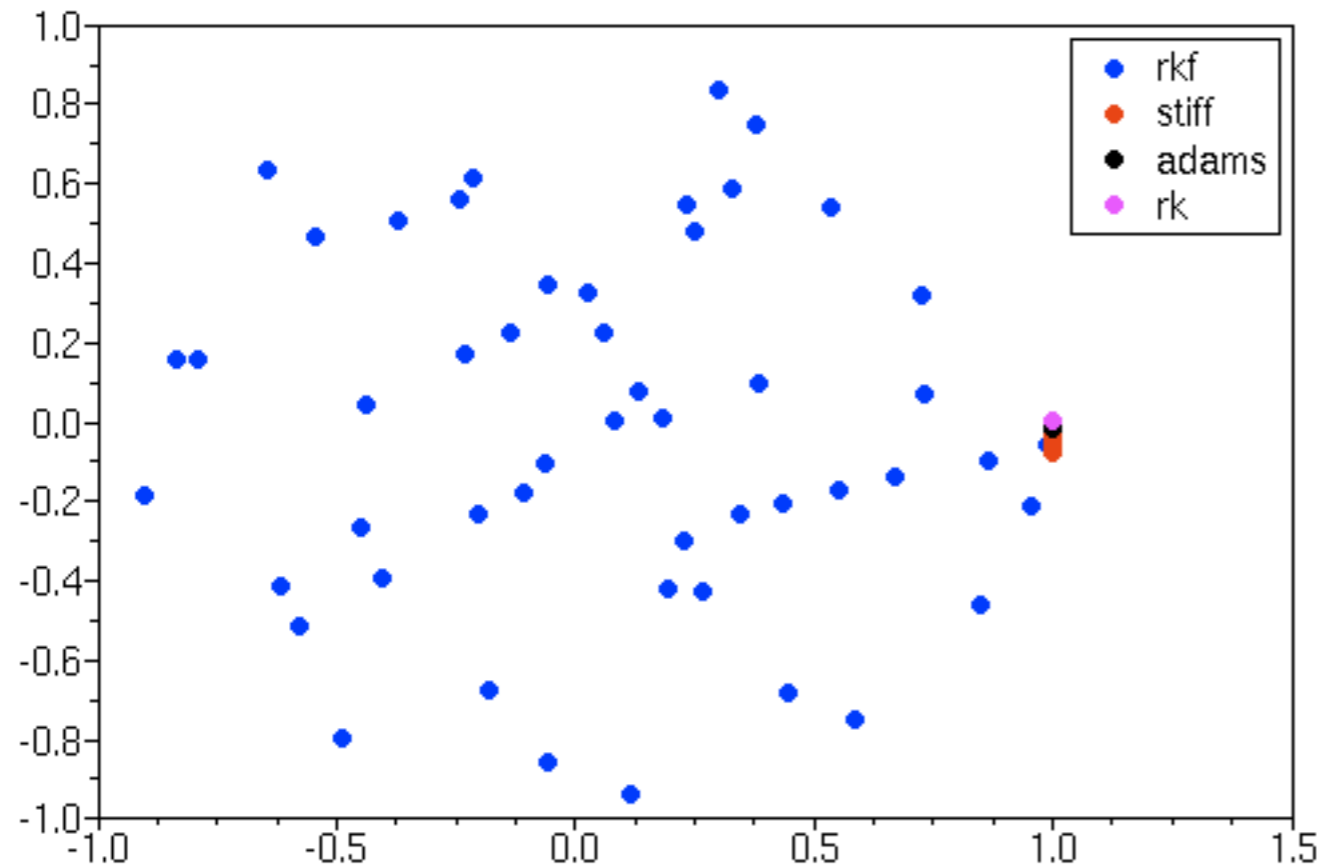
```
-->getf('integrate_planet.sce')
-->tic;y_rk=ode('rk',[1;1],0,[1:20],mystiff);toc
ans =
    15.571
-->tic;y_stiff=ode('stiff',[1;1],0,[1:20],mystiff);toc
ans =
    0.011
```

```
-->tic;y_adams=ode('adams',[1;1],0,[1:20],mystiff);toc
ans =
    0.081
```

```
12 function[dydx]=mystiff(t,y)
13     v=[1 -0.5;-0.5 1];
14     lambda=[1 0;0 -10000];
15     a=v+lambda*v';
16     dydx=a*y;
17 endfunction
```

Scilab has built-in stiff solver. For his particular example, it's >7 times faster than smooth solver, and >1400 times faster than RK!

Circular Orbit



```
1 function(dydx)=planet_dydx(t,y)
2   r=sqrt(y(1)^2+y(3)^2);
3   r3=r*r*r;
4   dydx(4)=0;
5   dydx(1)=y(2);
6   dydx(2)=-y(1)/r3;
7   dydx(3)=y(4);
8   dydx(4)=-y(3)/r3;
9 endfunction
```

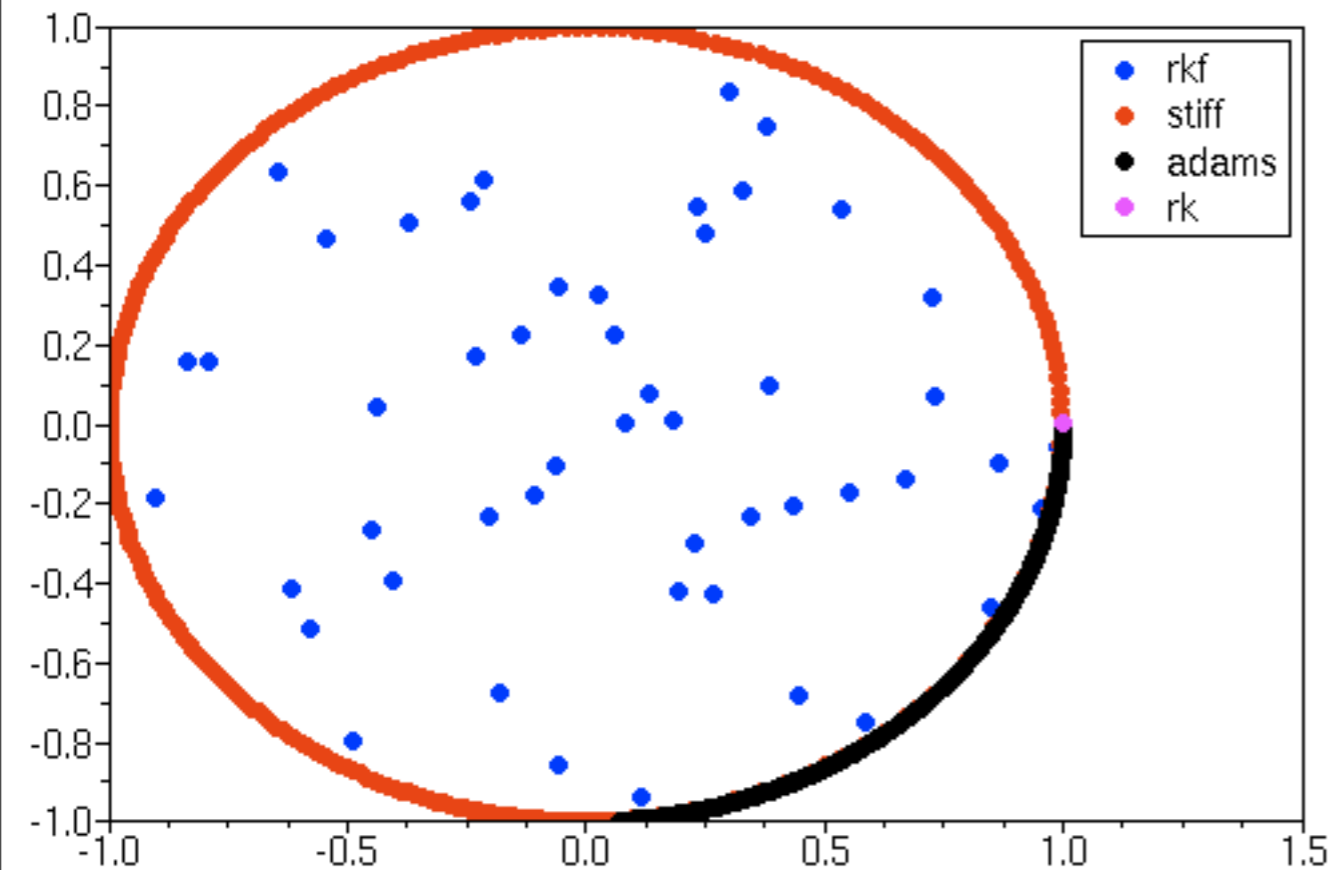
```
-->demonstrate_odes(50);
t_adams: 0.20
t_stiff: 0.24
t_rk: 3.47
t_rkf: 0.34
-->■
```

Integrate circular orbit equations in cartesian coordinated for 50 cycles. Sample once per orbit: perfect integrator should stay put.

RKF useless. Is RK better? Well, took 20 times longer..

```
18 function[value]=demonstrate_odes(norbit)
19   if ~exists('norbit')
20     norbit=200;
21   end
22   t=(1:norbit)*2*pi;
23   y0=[1 0 0 -1]';
24   tic;y_adams=ode('adams',y0,0,t,planet_dydx);t_adams=toc()
25   tic;y=ode(y0,0,t,planet_dydx);tt=toc()
26   tic;y_rk=ode('rk',y0,0,t,planet_dydx);t_rk=toc()
27   tic;y_rkf=ode('rkf',y0,0,t,planet_dydx);t_rkf=toc()
28   tic;y_stiff=ode('stiff',y0,0,t,planet_dydx);t_stiff=toc()
29
30   clf;plot(y_rkf(1:4:$),y_rkf(3:4:$),'.')
31   plot(y_stiff(1:4:$),y_stiff(3:4:$),'r. ');
32   plot(y_adams(1:4:$),y_adams(3:4:$),'k. ');
33   plot(y_rk(1:4:$),y_rk(3:4:$),'m. ');
34   printf('t_adams: %0.2f\n',t_adams);
35   printf('t_stiff: %0.2f\n',t_stiff);
36   printf('t_rk: %0.2f\n',t_rk);
37   printf('t_rkf: %0.2f\n',t_rkf);
38   legend('rkf','stiff','adams','rk');
39   value=0;
40 endfunction
41
```

Again, 500 Orbits



```
-->demonstrate_odes(500);  
Warning: integration not completed!  
ep size  
  
t_adams: 2.00  
t_stiff: 2.41  
t_rk: 35.16  
t_rkf: 0.43  
  
-->□
```

At least this time we got a warning. Now that equations aren't stiff, predictor-corrector does better than stiff.

And this is why solar-system dynamics people have their own integrators.

Closing ODE Points

- If have conservative equations (e.g. gravity), then symplectic integrators are what you want. Explicitly conserve energy.
- We only talked about initial value problems. Sometimes also know $y(0), y(l)$. Boundary value problems more complicated.
- Please, for the love of all that is holy, if your function has a singularity a) know it, and b) know if it's integrable. If so, you can often start on top (or immediately adjacent) to singularity and get good answers.
- Again, **don't** write your own solver. **Do** know which one you might want to use. Only way is to have some idea of what the function should look like.

Quick & Dirty Summary

- We often have to interpolate. What method you pick depends on what your function/data looks like
- Integration usually relies on interpolation of a function. Adaptive integration works much better. High order \neq high accuracy. Unless it does...
- ODE's: like integration, but more complex. Don't know exactly which points to use, but can use many of same techniques to guess.
- Many different ODE solvers. Which one to use depends on what your equations look like.
- ***Always know your function!***