Interpolation & ODEs

Scientific Computing Course, Jan 2013

Homework

- Questions about Make for targets
- Imagine we had the following very simple (1d) diffusion in diffuse.cxx:

```
void derivative(double *y, double *x, double *d2y, int n) {

    for (int i=1; i<n-1; i++) {
        double dxl = x[i+1] - x[i-1];
        double dxr = x[i]   - x[i-1];
        double dx  = 0.5*(dxl + dxr);

        d2y[i] = ((y[i+1] - y[i])/dxr - (y[i] - y[i-1])/dxl ) / dx;
    }

    return;
}

void diffuse(double *tin, double *tout, double *x, int n, double coeff) {

    double *deriv = new double[n];

    derivative(tin, x, deriv, n);
    for (int i=1; i<n-1; i++) {
        tout[i] = tin[i] - coeff*deriv[i];
    }

}
```

Homework

- And a main program which drove it, main.cxx

```
double *old = tin;
double *cur = tout;
const int nsteps = 100;
for (int step=0; step < nsteps; step++) {
    diffuse(old, cur, x, npts, coeff);
    double *tmp = old;
    old = cur;
    cur = tmp;
}

out = fopen(outfilename,"w");
if (!out) {
    fprintf(stderr,"Could not open file \"%s\"; exiting\n", ou
    return -1;
}

fprintf(out, "%d\n", npts);
```

Homework

- We might have a Makefile that looks like this:

```
CXXFLAGS = -O2 -Wall -g
CXX = g++
LDLIBS = -lm

all: main

main: main.o diffuse.o
	$(CXX) -o $@ $(LDLIBS) $^

main.o: main.cxx diffuse.h
	$(CXX) -c -o $@ $(CXXFLAGS) $<

diffuse.o: diffuse.cxx
	$(CXX) -c -o $@ $(CXXFLAGS) $<

clean:
	rm -f *.o output*.txt *~ main
```

SciNet compute • calcul CANADA

Homework

- But we can add a different *main()* which does a couple simple tests on the diffusion routine (unit or integrated?)

```c
int main(int argc, char **argv) {

    int err;
    int allerr=0;
    int n=100;

    printf("Performing Constant Test...\n");
    err = doConstTest(n);
    if (!err)
        printf("PASS\n");
    else
        printf("FAIL\n");
    allerr += err;

    printf("Performing Linear Test...\n");
    err = doLinearTest(n);
    if (!err)
        printf("PASS\n");
    else
        printf("FAIL\n");
    allerr += err;

    return allerr;
}
```

Homework

- And create a makefile to automatically compile this and run it:

```
CXXFLAGS = -O2 -Wall -g
CXX = g++
LDLIBS = -lm

all: main tests

main: main.o diffuse.o
    $(CXX) -o $@ $(LDLIBS) $^

tests: tests.o diffuse.o
    $(CXX) -o $@ $(LDLIBS) $^

runtests: tests
    ./tests
```

Git Bisect

- Version Control (git) and automation (make) are tools to make your computing life better and more productive.
- Note that the tests had main return zero on success and non-zero on failure, by long convention.
- Now let's say I had been developing this program for a while without testing, and then...

Git Bisect

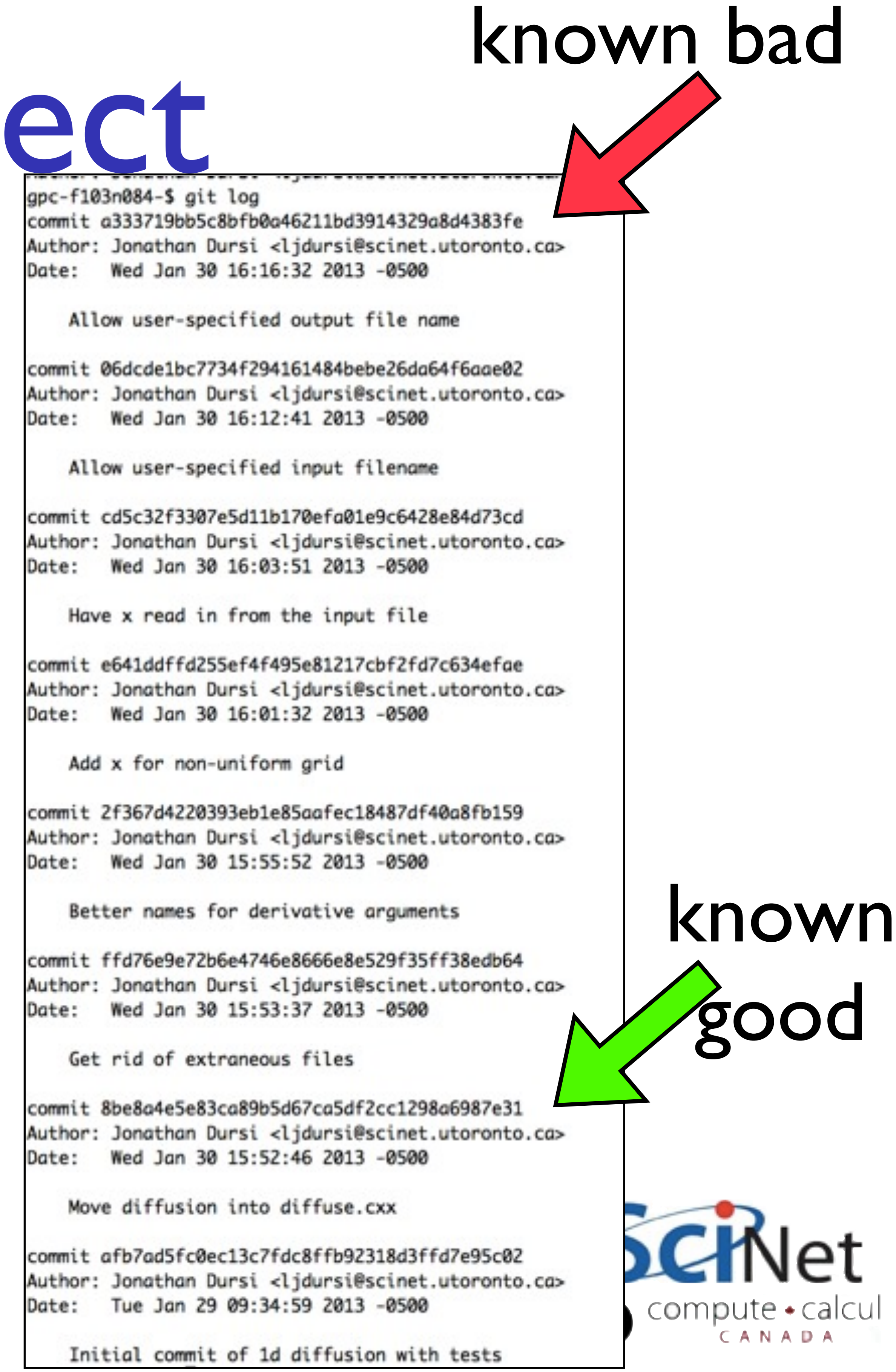
- Bah.
- We could use git diff to figure out what code change caused the bug..

```
gpc-f103n084-$ make runtests
g++ -c -o tests.o -O2 -Wall -g tests.cxx
g++ -c -o diffuse.o -O2 -Wall -g diffuse.cxx
g++ -o tests -lm tests.o diffuse.o
./tests
Performing Constant Test...
FAIL
Performing Linear Test...
FAIL
make: *** [runtests] Error 2
gpc-f103n084-$
```

SciNet compute • calcul CANADA

Git Bisect

- But we're not sure when the bug was introduced, so it's a little hard to figure out which commit caused it.
- Could checkout different versions and test...




```
gpc-f103n084-$ git log
commit a333719bb5c8bfb0a46211bd3914329a8d4383fe
Author: Jonathan Dursi <ljdursi@scinet.utoronto.ca>
Date:   Wed Jan 30 16:16:32 2013 -0500

    Allow user-specified output file name

commit 06dcde1bc7734f294161484bebe26da64f6aae02
Author: Jonathan Dursi <ljdursi@scinet.utoronto.ca>
Date:   Wed Jan 30 16:12:41 2013 -0500

    Allow user-specified input filename

commit cd5c32f3307e5d11b170efa01e9c6428e84d73cd
Author: Jonathan Dursi <ljdursi@scinet.utoronto.ca>
Date:   Wed Jan 30 16:03:51 2013 -0500

    Have x read in from the input file

commit e641ddffd255ef4f495e81217cbf2fd7c634efae
Author: Jonathan Dursi <ljdursi@scinet.utoronto.ca>
Date:   Wed Jan 30 16:01:32 2013 -0500

    Add x for non-uniform grid

commit 2f367d4220393eb1e85aafec18487df40a8fb159
Author: Jonathan Dursi <ljdursi@scinet.utoronto.ca>
Date:   Wed Jan 30 15:55:52 2013 -0500

    Better names for derivative arguments

commit ffd76e9e72b6e4746e8666e8e529f35ff38edb64
Author: Jonathan Dursi <ljdursi@scinet.utoronto.ca>
Date:   Wed Jan 30 15:53:37 2013 -0500

    Get rid of extraneous files

commit 8be8a4e5e83ca89b5d67ca5df2cc1298a6987e31
Author: Jonathan Dursi <ljdursi@scinet.utoronto.ca>
Date:   Wed Jan 30 15:52:46 2013 -0500

    Move diffusion into diffuse.cxx

commit afb7ad5fc0ec13c7fdc8ffb92318d3ffd7e95c02
Author: Jonathan Dursi <ljdursi@scinet.utoronto.ca>
Date:   Tue Jan 29 09:34:59 2013 -0500

    Initial commit of 1d diffusion with tests
```

Git Bisect

known bad

known good

```
$ git bisect start

$ git bisect bad HEAD

$ git bisect good afb7ad5fc0ec13c7fdc8ffb92318d3ffd7e95c02

Bisecting: 3 revisions left to test after this (roughly 2 steps)
[2f367d4220393eb1e85aafec18487df40a8fb159] Better names for derivative arguments

$ git bisect run make runtests
running make runtests
g++ -c -o tests.o -O2 -Wall -g tests.cxx
g++ -c -o diffuse.o -O2 -Wall -g diffuse.cxx
g++ -o tests -lm tests.o diffuse.o
./tests
Performing Constant Test...
PASS
Performing Linear Test...
PASS
Bisecting: 1 revision left to test after this (roughly 1 step)
```

find culprit via "make runtests"

Git Bisect

```
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[e641ddffd255ef4f495e81217cbf2fd7c634efae] Add x for non-uniform grid
running make runtests
./tests
Performing Constant Test...
FAIL
Performing Linear Test...
FAIL
make: *** [runtests] Error 2
e641ddffd255ef4f495e81217cbf2fd7c634efae is the first bad commit
commit e641ddffd255ef4f495e81217cbf2fd7c634efae
Author: Jonathan Dursi <ljdursi@scinet.utoronto.ca>
Date:   Wed Jan 30 16:01:32 2013 -0500

    Add x for non-uniform grid

:100644 100644 24dfc324a72163252edb63cb013606cacad72c61 bde5145d11d29c687ccf50a007ce547bf66ecacf M  diffuse.cxx
:100644 100644 c9ef874c3fb731454965f43087856dfd809de2e2 bfdebae4341e1dfde11fc0905ff0831da312120c M  diffuse.h
:100644 100644 79e12641dbcc2e8776a7d45da99b6ca9e644ea7b 7984946e380d7ddd63318a59301f0ef9136[illegible]d M  main.cxx
:100644 100644 7b8da7f207dc9ca8346d2108f2d4644d8062b17f fc5be55fed2e275ee73257b80f88973eb4[illegible]e M  tests.cxx
bisect run success
```

commit that broke the test

by this known incompetent

who changed these files

HEAD is now the first broken commit

Ah ha! One of my dx's is wrong.

git bisect reset to get back to the way things were.

```
gpc-f103n084-$ git show HEAD diffuse.cxx
commit e641ddffd255ef4f495e81217cbf2fd7c634efae
Author: Jonathan Dursi <ljdursi@scinet.utoronto.ca>
Date:   Wed Jan 30 16:01:32 2013 -0500

    Add x for non-uniform grid

diff --git a/diffuse.cxx b/diffuse.cxx
index 24dfc32..bde5145 100644
--- a/diffuse.cxx
+++ b/diffuse.cxx
@@ -1,18 +1,21 @@
-// assumes regular spacing
-void derivative(double *y, double *d2y, int n) {
+void derivative(double *y, double *x, double *d2y, int n) {

     for (int i=1; i<n-1; i++) {
-        d2y[i] = (y[i+1] - 2.*y[i] + y[i-1]);
+        double dxl = x[i+1] - x[i-1];
+        double dxr = x[i]   - x[i-1];
+        double dx  = 0.5*(dxl + dxr);
+
+        d2y[i] = ((y[i+1] - y[i])/dxr - (y[i] - y[i-1])/dxl ) / dx;
     }

     return;
 }

-void diffuse(double *tin, double *tout, int n, double coeff) {
+void diffuse(double *tin, double *tout, double *x, int n, double coeff) {

     double *deriv = new double[n];

-    derivative(tin, deriv, n);
+    derivative(tin, x, deriv, n);
     for (int i=1; i<n-1; i++) {
         tout[i] = tin[i] - coeff*deriv[i];
     }
gpc-f103n084-$ git bisect reset
```

Testing

- Note that:
  - the more frequent the checkins, and
  - the more specific the unit tests,
- the more precisely this will hone in on the error.

Git Bisect

- **If** you
  - commit regularly,
  - have a good test suite,
  - have build/test automation,
- Then those tools can help you **automatically find where bugs were introduced**.
- Even without automation (say bug introduced before the tests were), you can use git bisect
- *$ git help bisect*

Final Testing Note

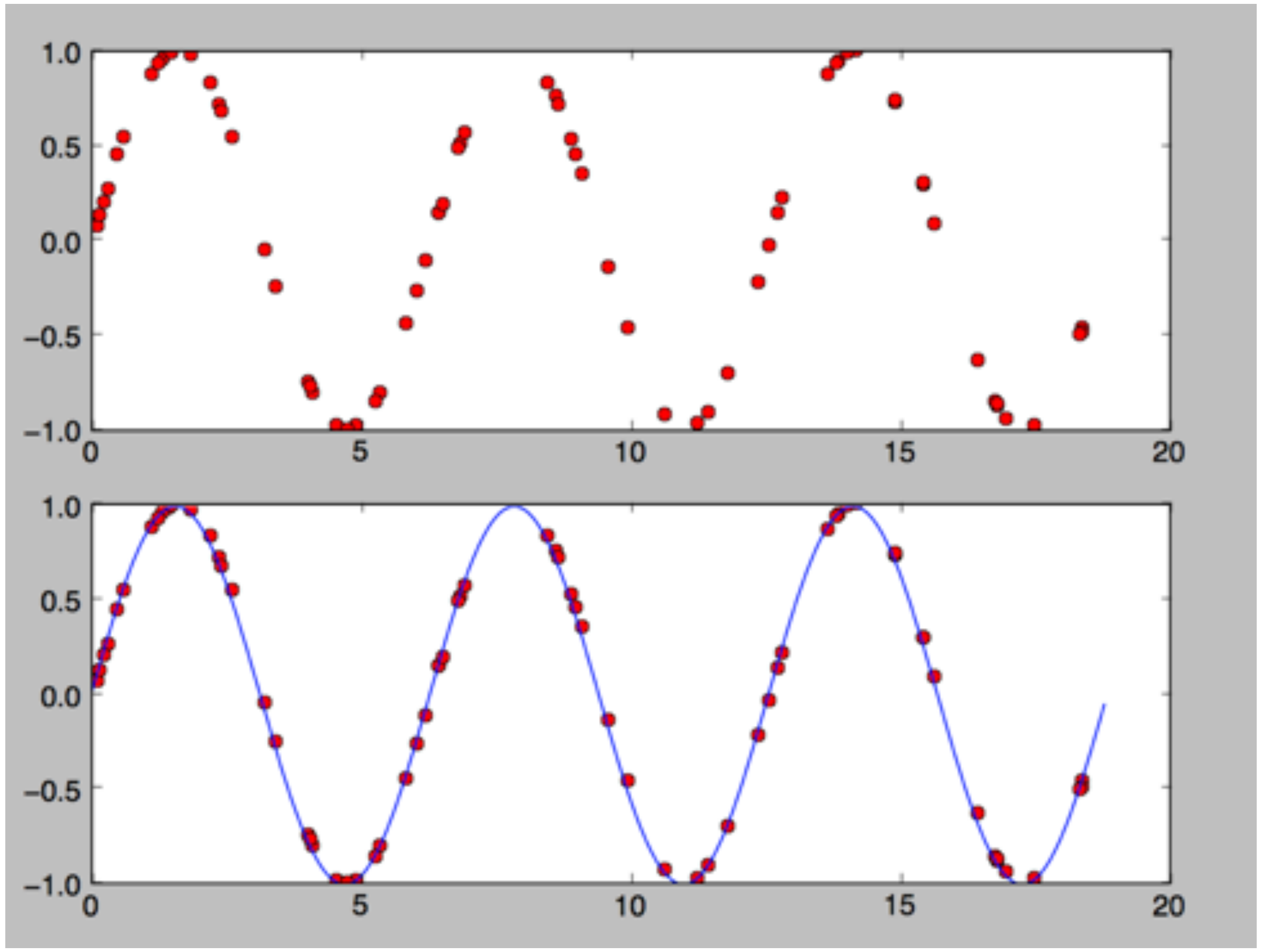
- You're not finished when you fix a bug.
- If it's the sort of bug that could conceivably crop up again, *add a test for it*, in your test suite or just in the code (eg, `assert(n > 0) .`)
- **Nothing** is more frustrating than finding and fixing the same bug **twice**.

Interpolation & ODEs

Scientific Computing Course, Jan 2013

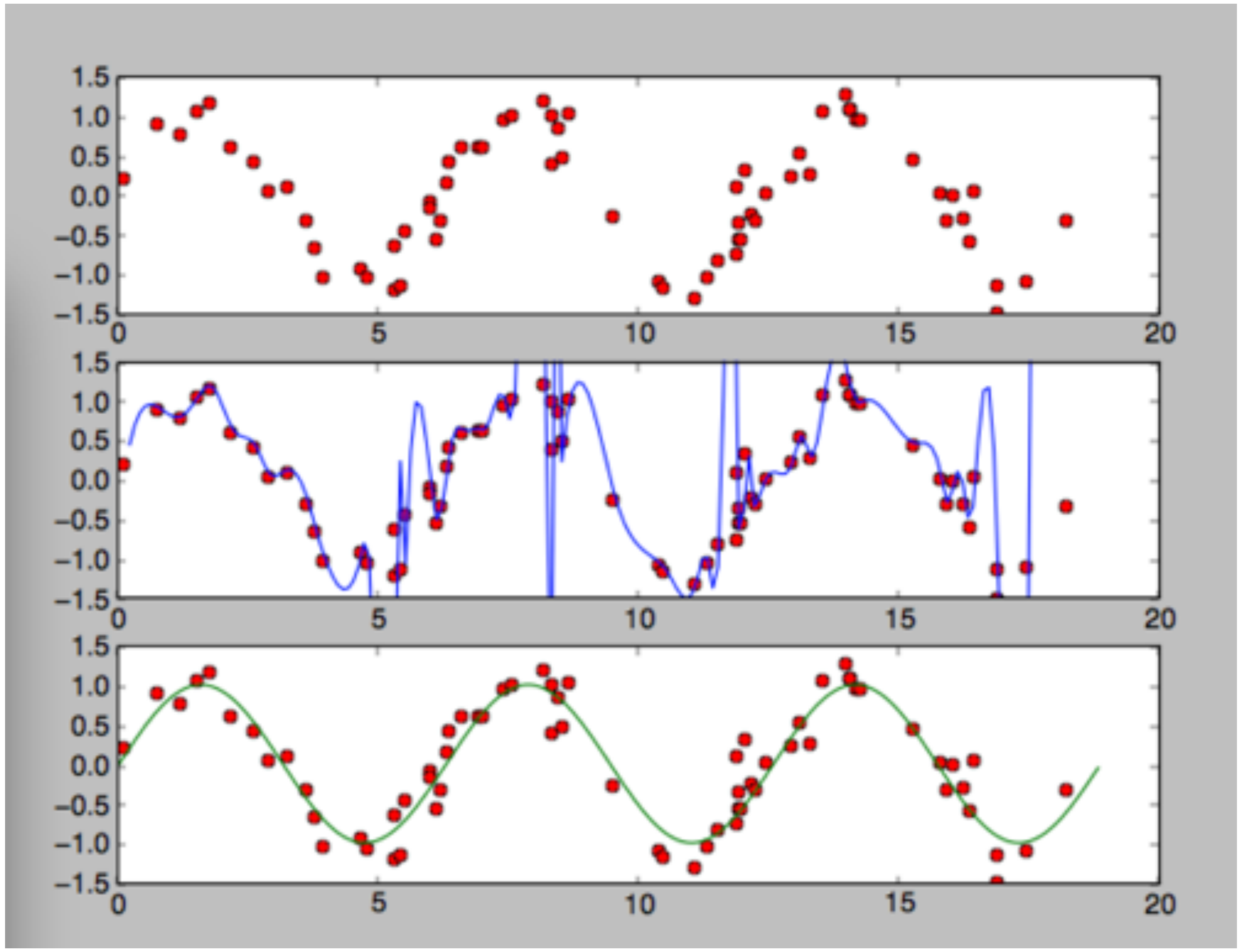
Interpolation

- We're often given, or compute, discrete data
- But to use our mathematical machinery on it we need continuous function
- Or need to know value between points, if even just to plot.



Interpolation

- Interpolation returns a function that passes through all input points,
- Or values of that function at intermediate points.
- Not what you want when you have noisy data: *fitting* or *regression*. Different topic.



Polynomial interpolation

- Common approach
- For n points, use n-1th order polynomial: n coefficients
- Solve a linear system (nonlinear in input data)

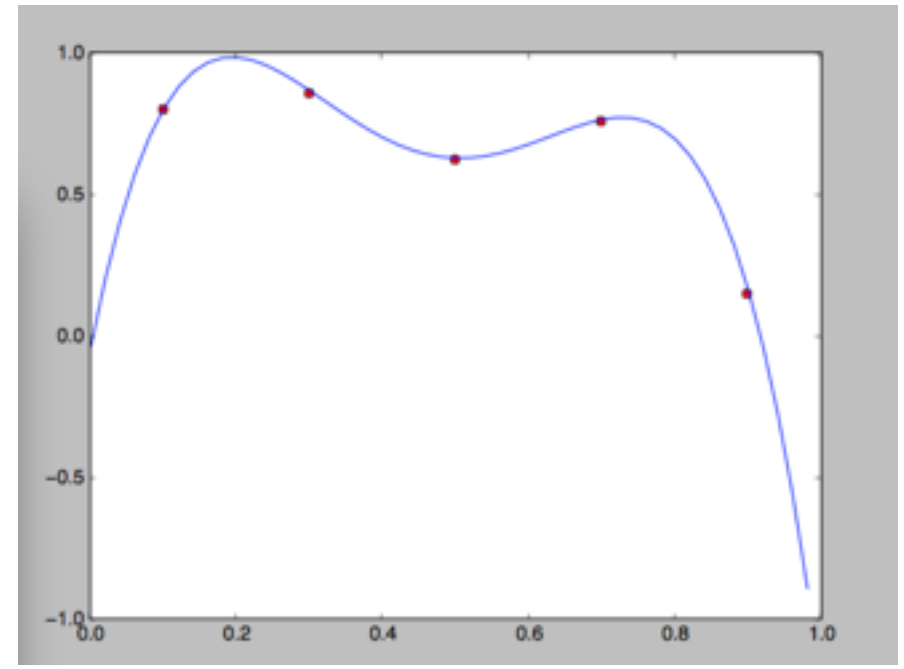
$$y_1 = a_0 + a_1 x_1^1 + \cdots + a_{n-1} x_1^{n-1}$$
$$y_2 = a_0 + a_1 x_2^1 + \cdots + a_{n-1} x_2^{n-1}$$
$$\cdots$$
$$y_n = a_0 + a_1 x_n^1 + \cdots + a_{n-1} x_n^{n-1}$$

$$\begin{pmatrix} y_1 \\ y_2 \\ \cdots \\ y_n \end{pmatrix} = \begin{pmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \ldots & x_2^{n-1} \\ \cdots & & & & \\ 1 & x_n & x_n^2 & \ldots & x_n^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \cdots \\ a_{n-1} \end{pmatrix}$$

$$\mathbf{y} = \mathbf{X}\mathbf{a}$$

Polynomial Interpolation

- Common approach
- For n points, use n-1th order polynomial: n coefficients
- Solve a linear system (nonlinear in input data)




```
In [92]: x = arange(.1,.91,.2); y = rand(5);
    ...: xx = arange(0,1,.02)
    ...:

In [93]: plot(x,y,'ro')
Out[93]: [<matplotlib.lines.Line2D at 0x8933070>]

In [94]: polyInterpFun = scipy.interpolate.lagrange(x,y

In [95]: yy = polyInterpFun(xx)

In [96]: plot(xx,yy,'b-')
Out[96]: [<matplotlib.lines.Line2D at 0x8933470>]
```

Basis Functions

- Here we're solving for parameters which generate a linear combination of basis functions

$$y_1 = a_0 + a_1 x_1^1 + \cdots + a_{n-1} x_1^{n-1}$$
$$y_2 = a_0 + a_1 x_2^1 + \cdots + a_{n-1} x_2^{n-1}$$
$$\cdots$$
$$y_n = a_0 + a_1 x_n^1 + \cdots + a_{n-1} x_n^{n-1}$$

- The basis functions here are 1, x, x^2, x^3, ...

$$\begin{pmatrix} y_1 \\ y_2 \\ \cdots \\ y_n \end{pmatrix} = \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \cdots & & & & \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \cdots \\ a_{n-1} \end{pmatrix}$$

- They can be any other functions that span the relevant function space.

$$\mathbf{y} = \mathbf{X}\mathbf{a}$$

Orthogonal Basis Functions

- We have to solve a linear system, which is expensive
- Can't just write down form for (say) a_1 without calculating all others; basis functions overlap.
- If the basis functions are **orthogonal** in some (any) sense, can skip this; can calculate individual coefficients explicitly
- Any set of basis functions can be orthogonalized

$$y = \sum_i a_i f_i(x)$$

$$\langle y, f_j(x) \rangle = \sum_i a_i \langle f_i(x), f_j(x) \rangle$$

$$\langle y, f_j(x) \rangle = \sum_i a_i \delta_{i,j}$$

$$a_j = \langle y, f_j(x) \rangle$$

SciNet compute • calcul CANADA

Orthogonal Basis Functions

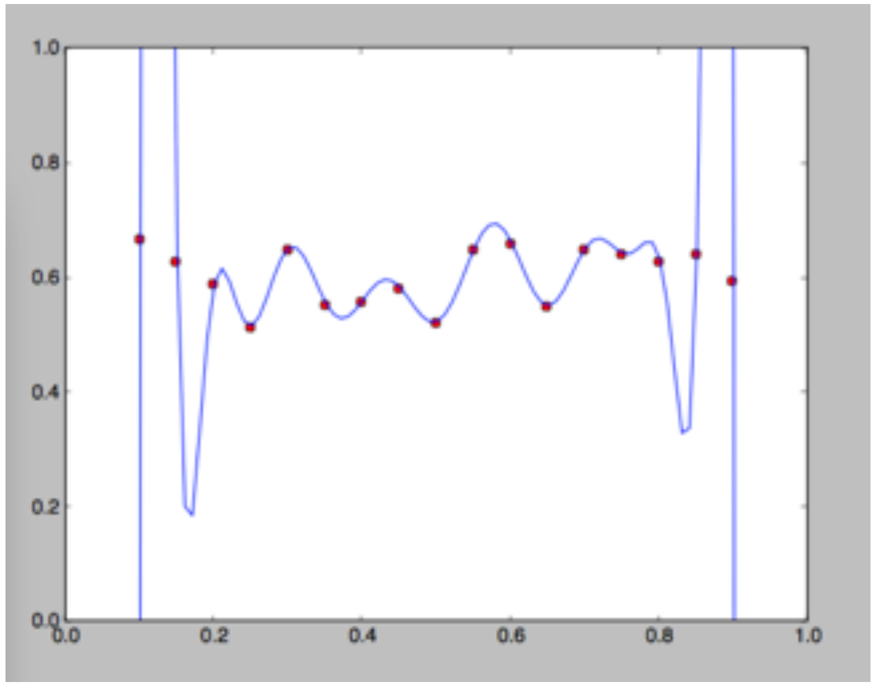
- In polynomials, there are several ways of orthogonalization (depending on your inner product)
- Lagrange interpolating polynomials particularly straightforward
- Functions in a Fourier series are orthogonal

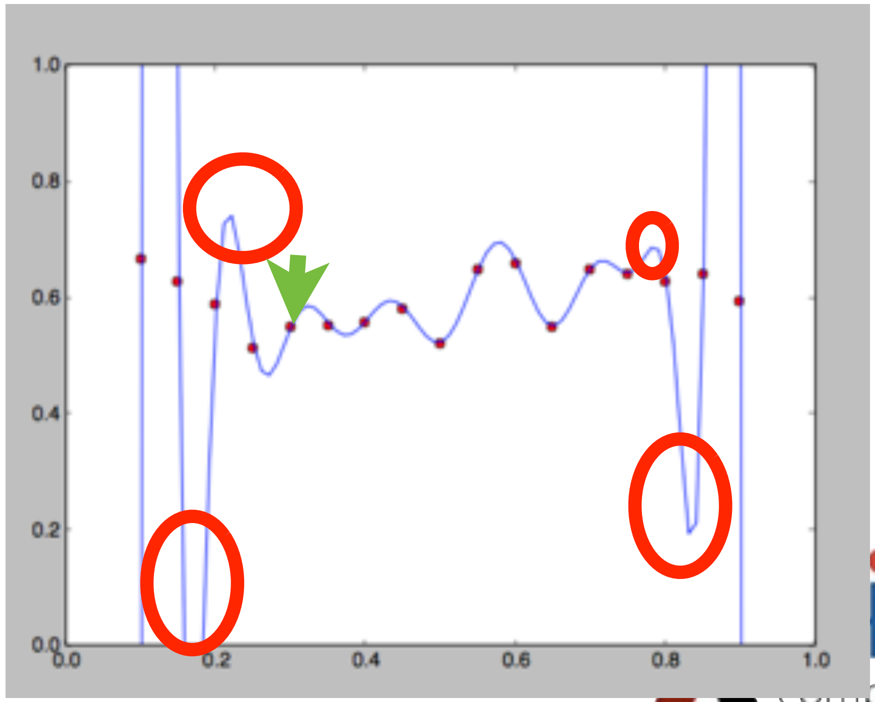
$$l_j = \frac{\prod_{m \neq j}(x - x_m)}{\prod_{m \neq j}(x_j - x_m)}$$

SciNet compute • calcul CANADA

Piecewise Interpolation

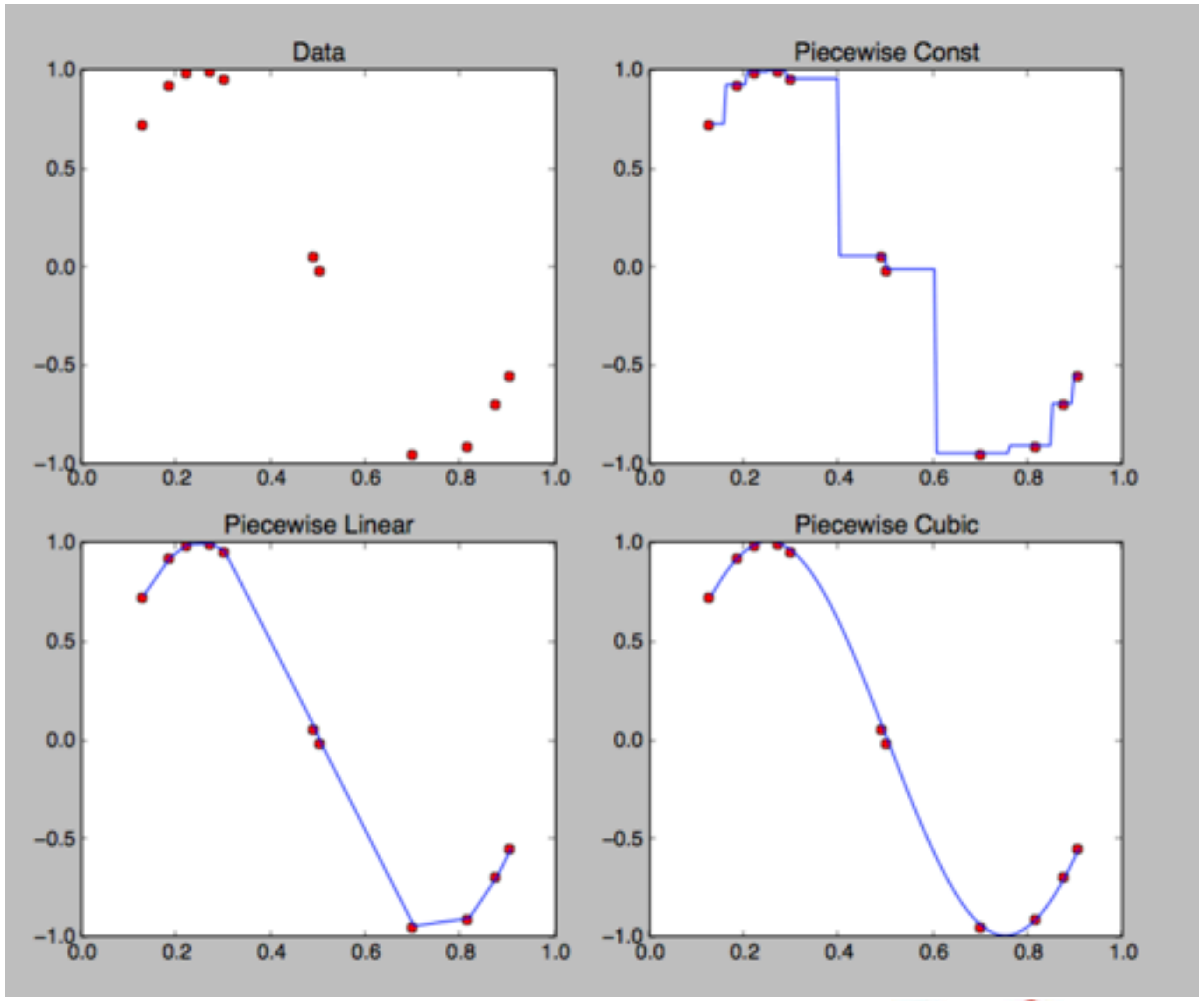
- Often don't want a single, global closed-form function to describe our data.
- (But note: spectral methods)
- Global function very dependent on every piece of data
- That high order polynomial - very wiggly





Piecewise Interpolation

- In each between-points, perform an interpolant as before based on nearby points.
- Piecewise constant; pick value of closest point
- Linear: draw a straight line between neighbouring points, etc.



Piecewise Interpolation

- In each between-points, perform an interpolant as before based on nearby points.
- Piecewise constant; pick value of closest point
- Linear: draw a straight line between neighbouring points, etc.

```
import scipy
import scipy.interpolate

x = sort(rand(11))
y = sin(x*2*pi)

xx = arange(0,.99,.005)

nearest = scipy.interpolate.interp1d(x,y,kind='nearest',bounds_error=False)
linear  = scipy.interpolate.interp1d(x,y,kind='linear',bounds_error=False)
cubic   = scipy.interpolate.interp1d(x,y,kind='cubic',bounds_error=False)

subplot(2,2,1)
plot(x,y,'ro')
xlim([0,1])
title("Data")

subplot(2,2,2)
plot(x,y,'ro')
plot(xx, nearest(xx),'b-')
xlim([0,1])
title("Piecewise Const")

subplot(2,2,3)
plot(x,y,'ro')
plot(xx, linear(xx),'b-')
xlim([0,1])
title("Piecewise Linear")

subplot(2,2,4)
plot(x,y,'ro')
plot(xx, cubic(xx),'b-')
xlim([0,1])
title("Piecewise Cubic")
```

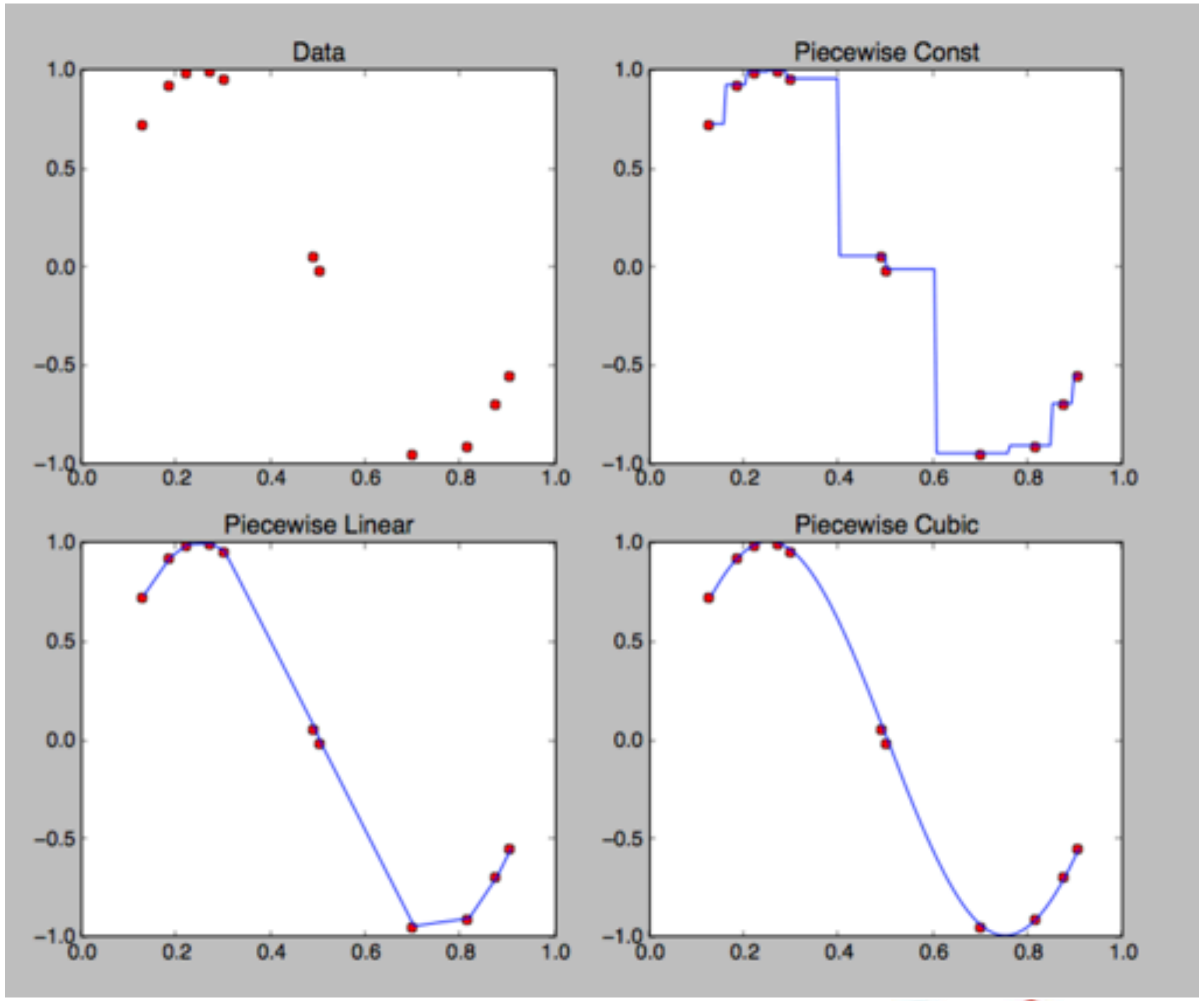
Piecewise Polynomial

interpolant($\mathbf{x}$,$\mathbf{y}$,newx,p) =

find i : x_i < newx < x_{i+1}

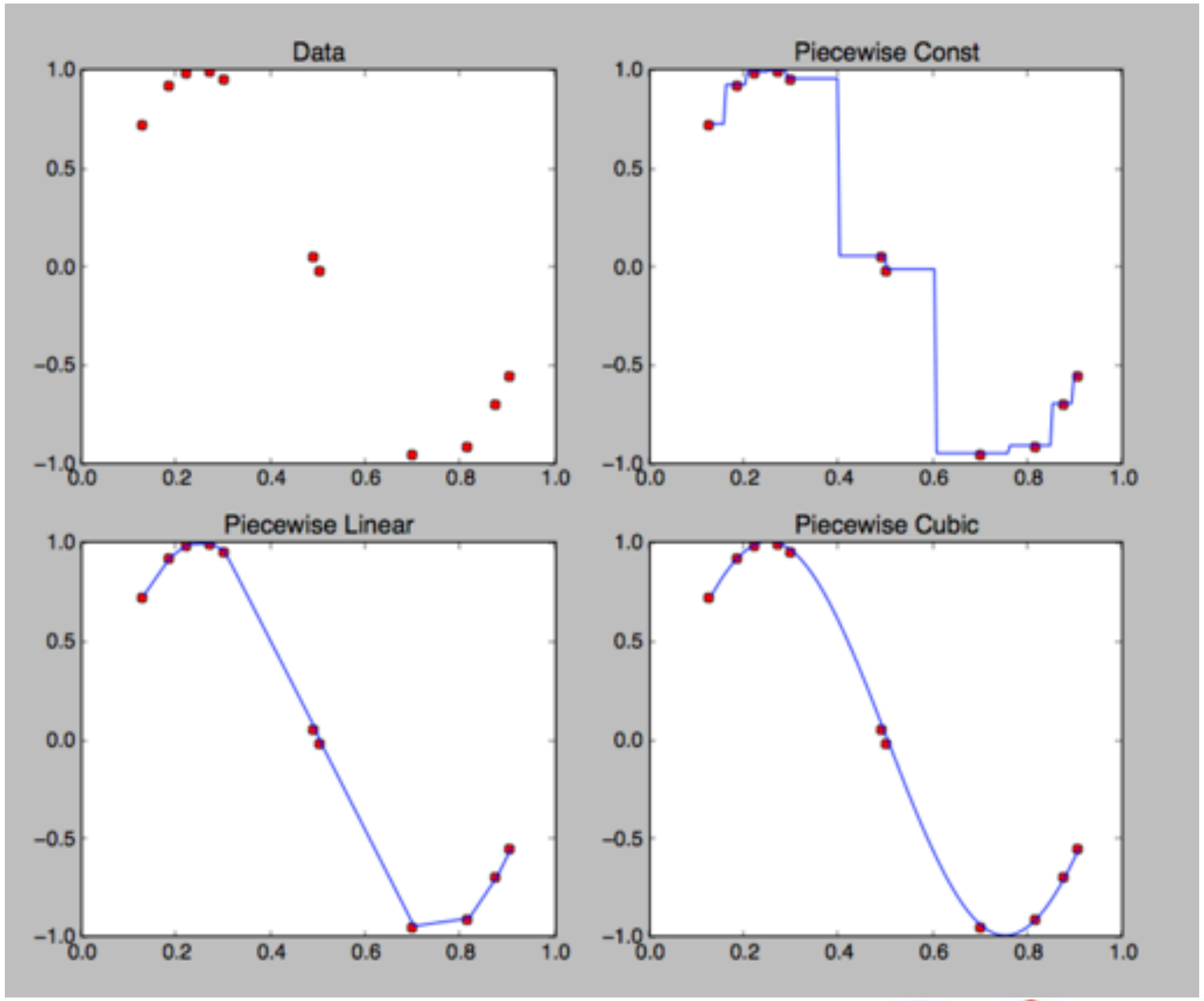
build lagrange polynomial from $(x_{i-p/2},\ldots,x_{i+p/2+1})$, $(y_{i-p/2},\ldots,y_{i+p/2+1})$

interpolate to newx



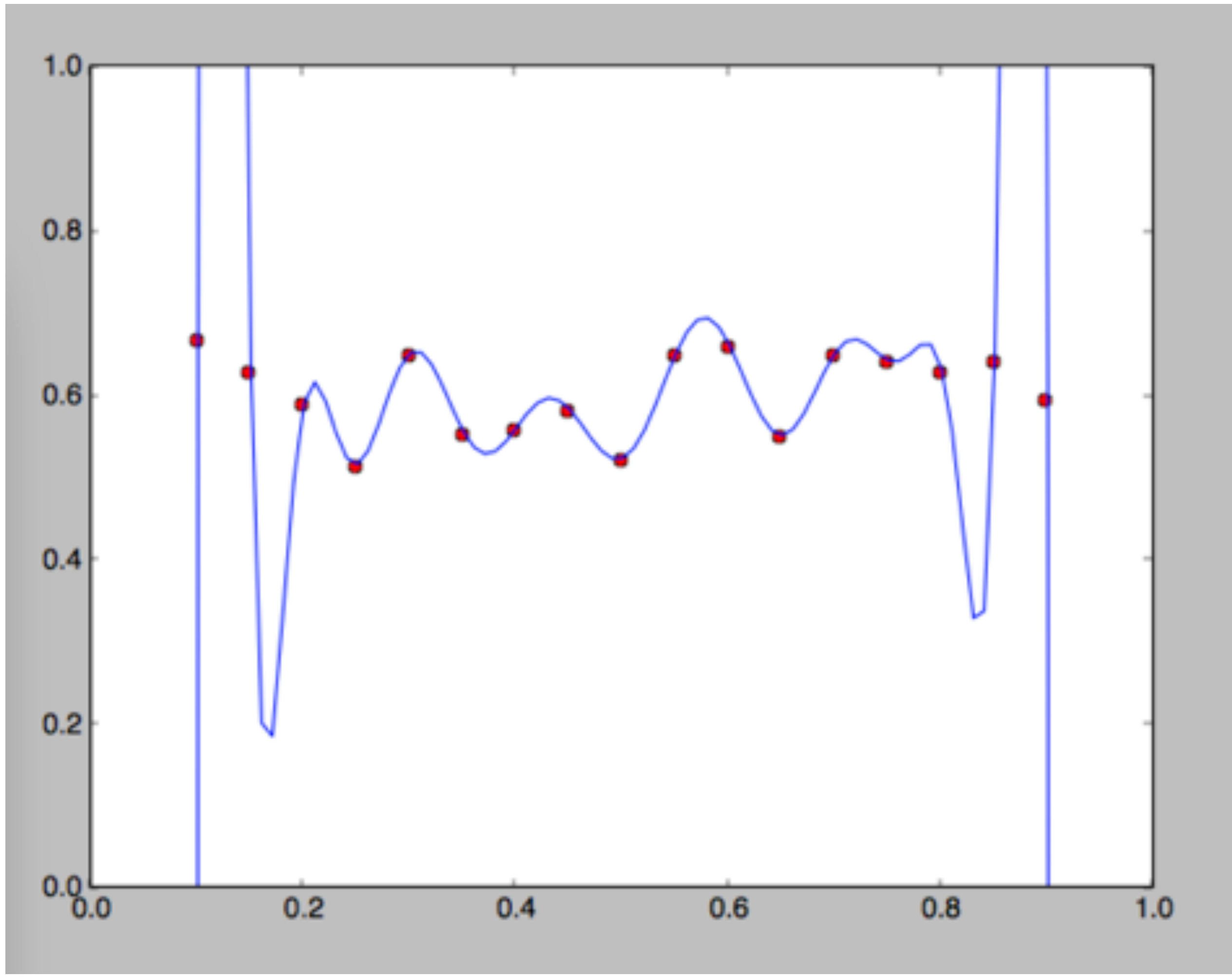
Piecewise Interpolation

- There's obviously some sense in which higher-order local interpolants approximate the "true" function better.
- Can formalize this intuition with Taylor series analysis.
- Approximation error of a p^{th} order polynomial leaves error of only $O(\Delta x^{p+1})$



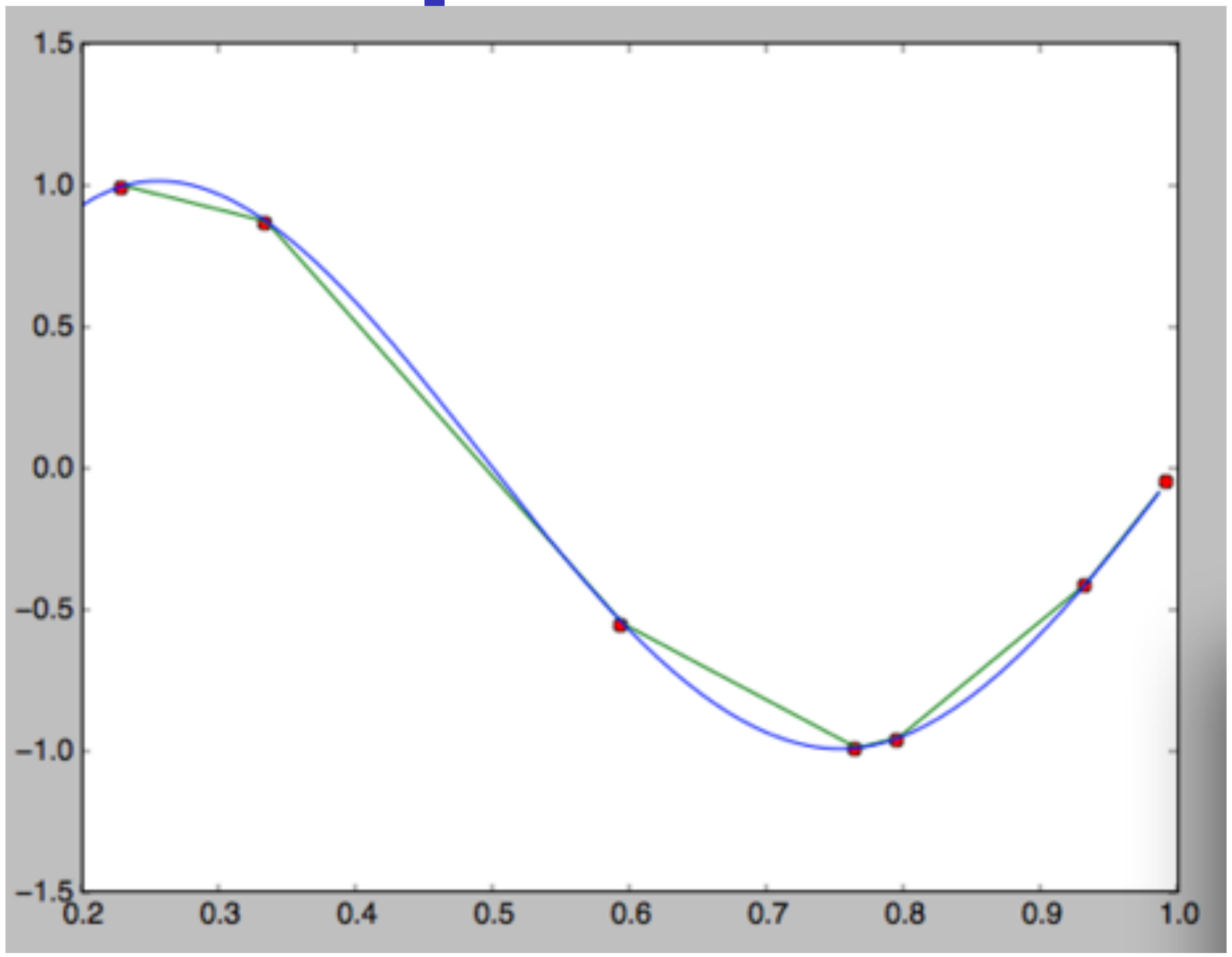
Danger! Danger!

- Thinking in terms of $O(\Delta x^{p+1})$ can be helpful; error converging faster rather than slower is good. **But remember:**
- Assumes *smooth* underlying function
- Higher order - more sensitive to ringing
- Performs abysmally at extrapolation
- Needs more data - more difficulty at ends of domain
- Statement of *asymptotics*. For a given Δx, a specific p^{th}-order accurate approximation may or may not be more accurate than a specific $(p-1)^{th}$ order method.



Splines

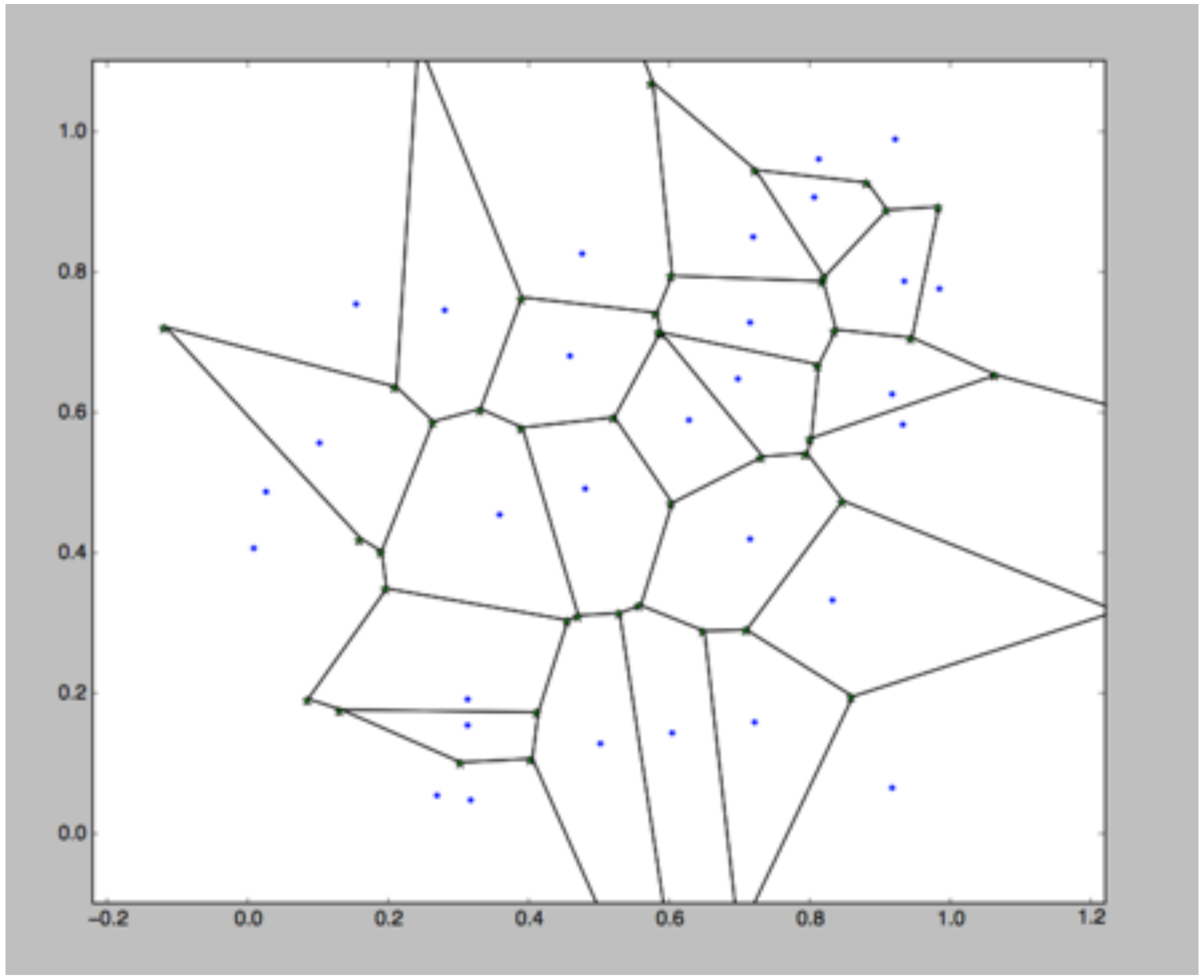
- Desired properties of interpolant depends on what you're going to use them for
- Piecewise polynomials as above: good, and continuous: but **derivatives** aren't continuous.
- If needed, use same lower number of points but higher order interpolating polynomial.
- Use extra d.o.f.s to match derivatives at interpolated points.
- Impose some condition at ends of interpolated region.




```
x = sort(rand(7))
y = sin(x*2*pi)

xx = arange(0.,.99,.005)

linear  = scipy.interpolate.interp1d(x,y,
            kind='linear',bounds_error=False)
spline = scipy.interpolate.
          InterpolatedUnivariateSpline(x,y)

plot(x,y,'ro')
plot(xx,linear(xx),'g-')
plot(xx,spline(xx),'b-')
```

Multidimensional piecewise interpolation

- Note that piecewise interpolation of irregular multidimensional data is harder
- Not trivial to figure out which region a given point is in
- On regular lattice, however, much simpler



Bilinear interpolation

- On 2d grids, simple approaches such as bilinear interpolants are sometimes used

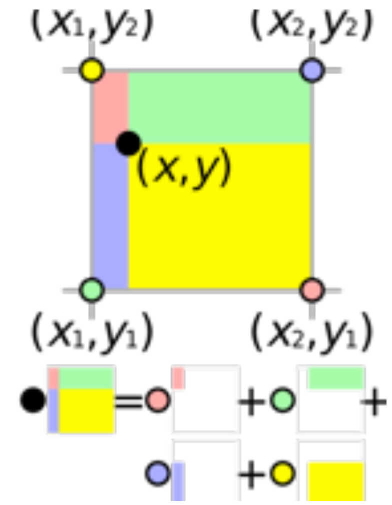
- Product of two linear interpolations

- 4 values, 4 unknowns.

- Lends itself to an interesting geometric interpretation.

$$f(x,y) = (a_1 + a_2(x - x_0))(a_3 + a_4(y - y_0))$$

$$\begin{aligned} f(x,y) = & b_1 + b_2(x - x_0) \\ & + b_3(y - y_0) \\ & + b_4(x - x_0)(y - y_0) \end{aligned}$$



http://en.wikipedia.org/wiki/File:Bilinear_interpolation_visualisation.svg

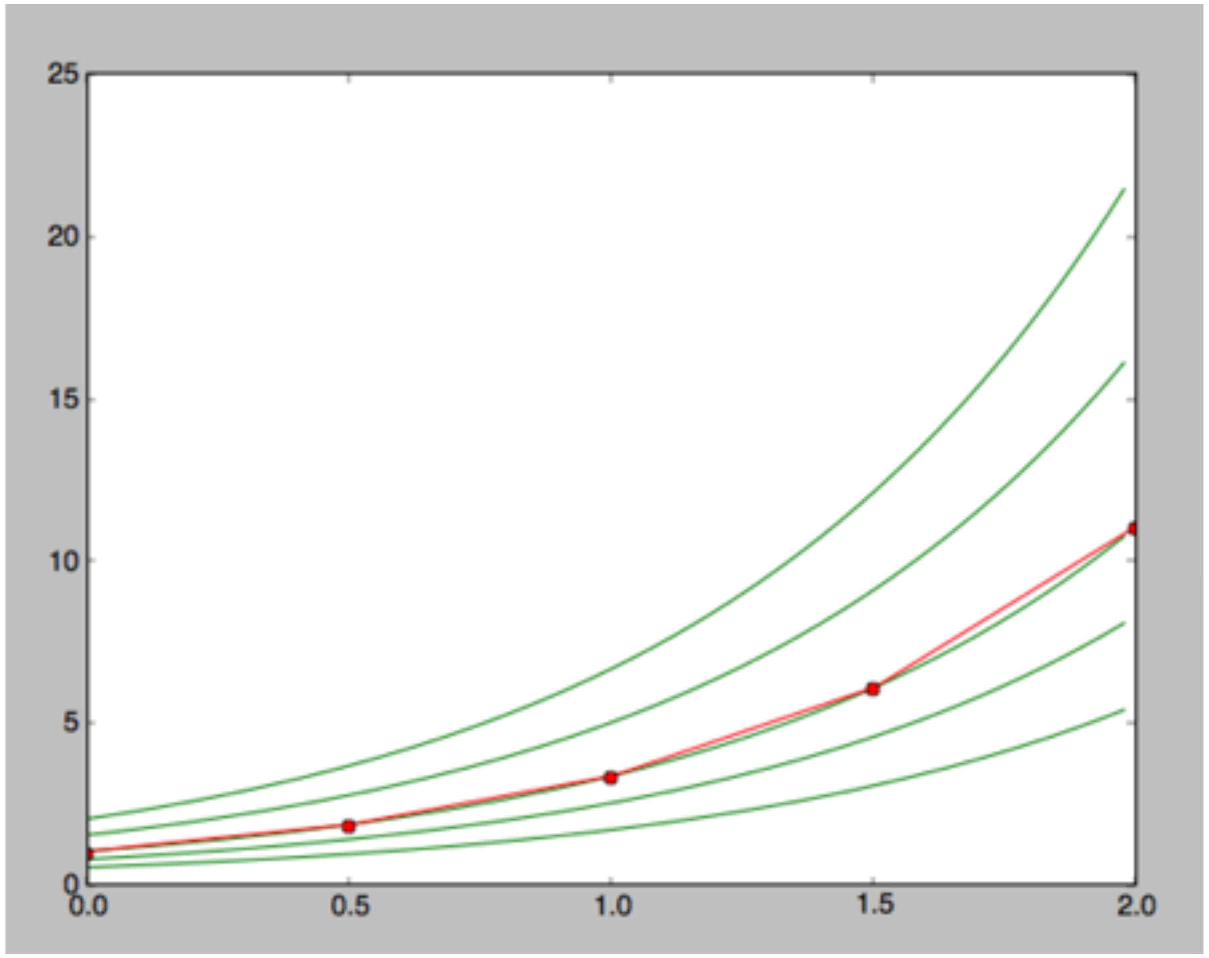
Initial Value ODEs

- Given some initial conditions and a differential equation, evolve the differential equation.
- Eg, given:

$$\mathbf{y}' = \mathbf{f}(\mathbf{y}, t)$$

$$(\mathbf{y}_0, t_0)$$

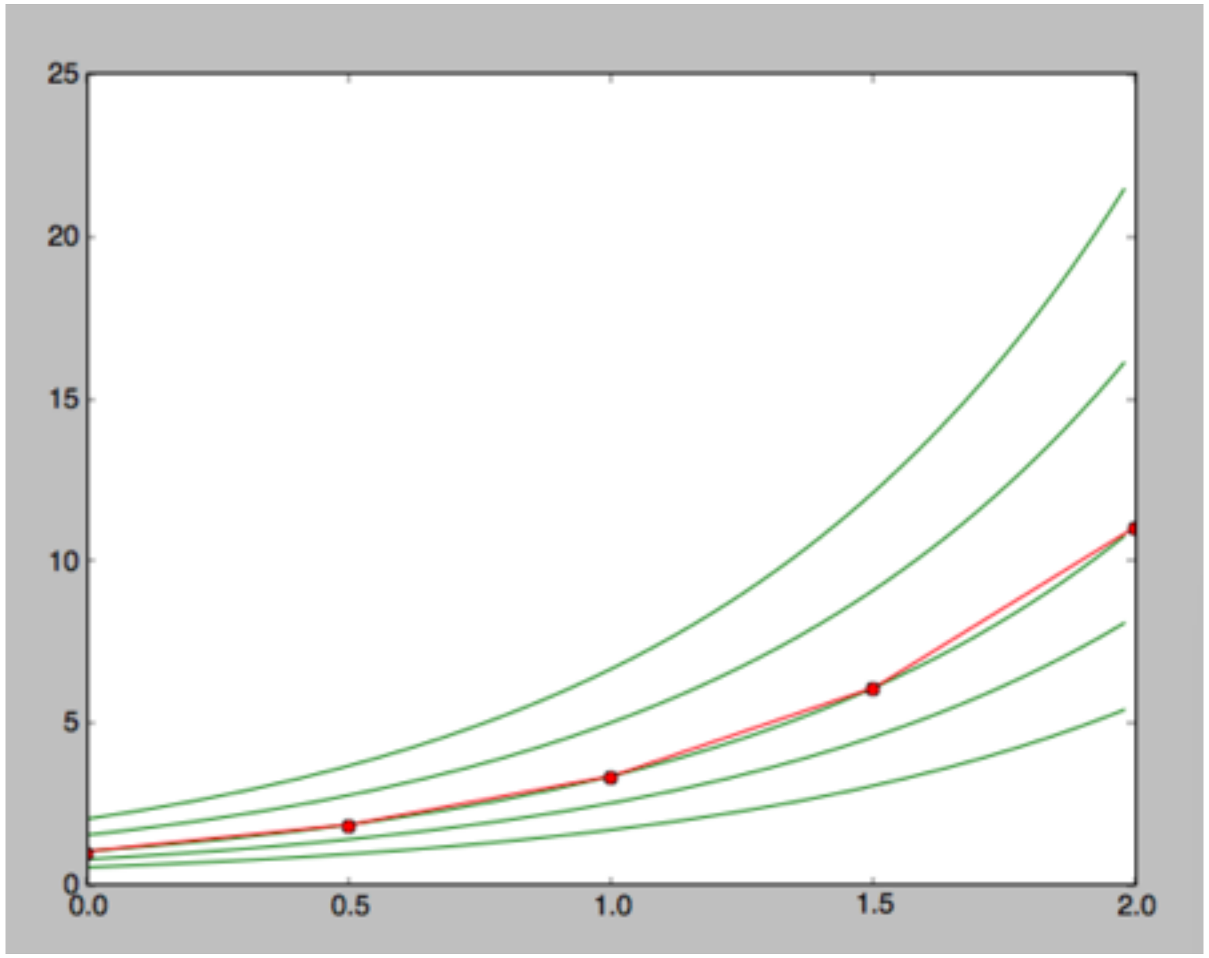
- evolve relevant y(t)




```
def f(y,t):
    return 1.2*y

ts = [0,.5,1,1.5,2]
ys = scipy.integrate.odeint(f, 1, ts)

plot(ts, ys, 'o-')
```

Initial Value ODEs

- If our **f** is Lipshitz continuous (differentiable), ∃ unique solution given ICs.
- However, that doesn't necessarily mean we can calculate it well.
- Stability of equation; stability of method; accuracy.



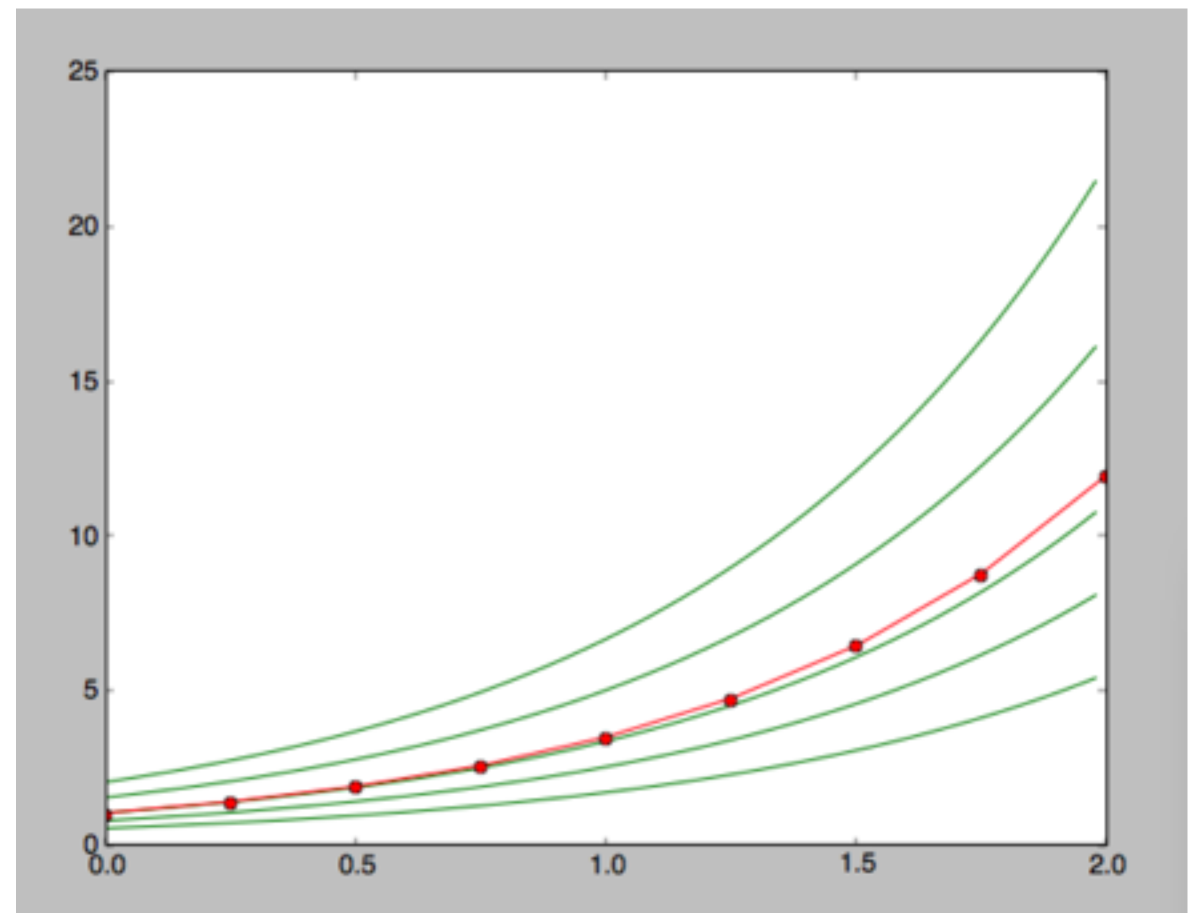
$$\mathbf{y}' = \mathbf{f}(\mathbf{y}, t)$$

$$(\mathbf{y}_0, t_0)$$

SciNet compute • calcul CANADA

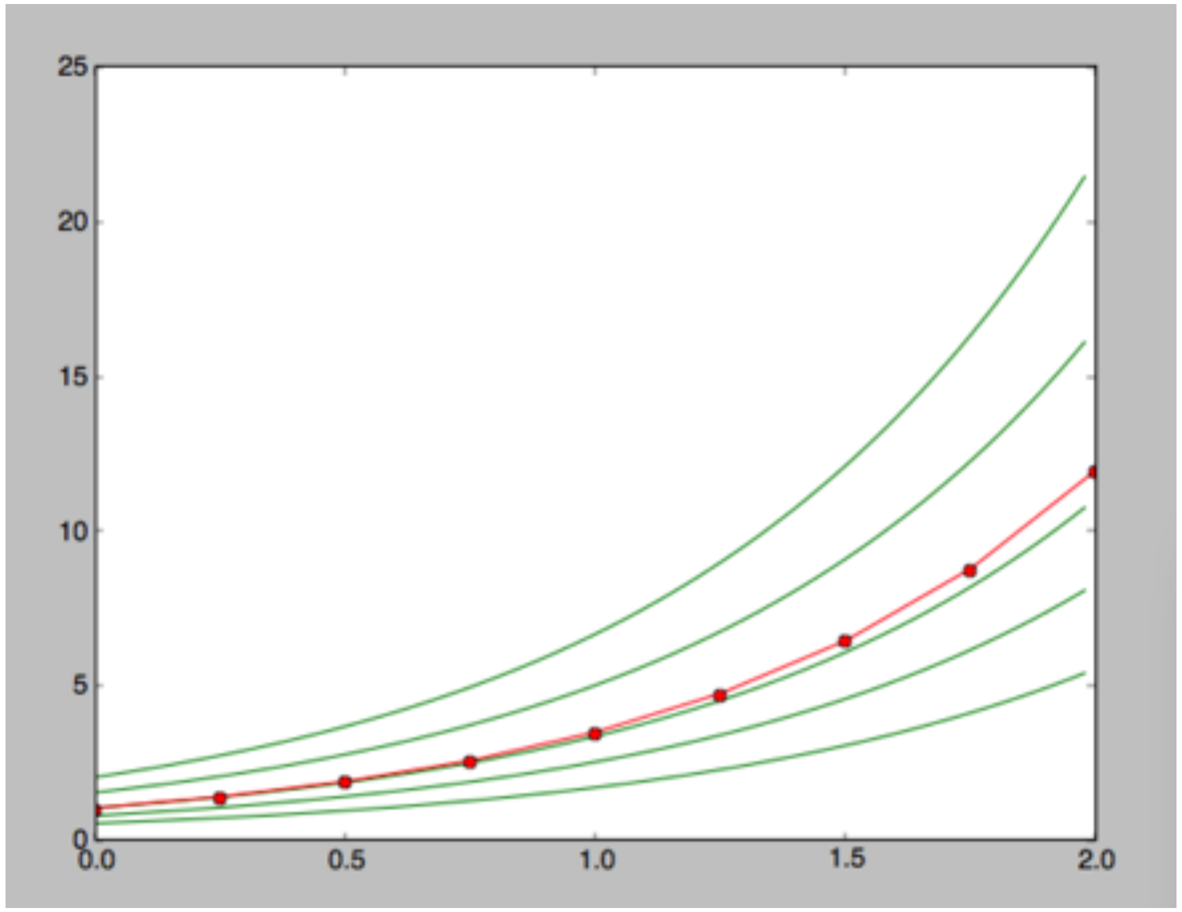
Equation stability

- Some systems are inherently challenging to integrate
- Eigenvalues > 1; small deviations pull you further away from solution
- Since small errors will always creep in (Part II), very challenging for correctness.



Equation stability

- Accuracy: how close to you stay to current solution?
- Stability: how do nearby solutions diverge from each other?

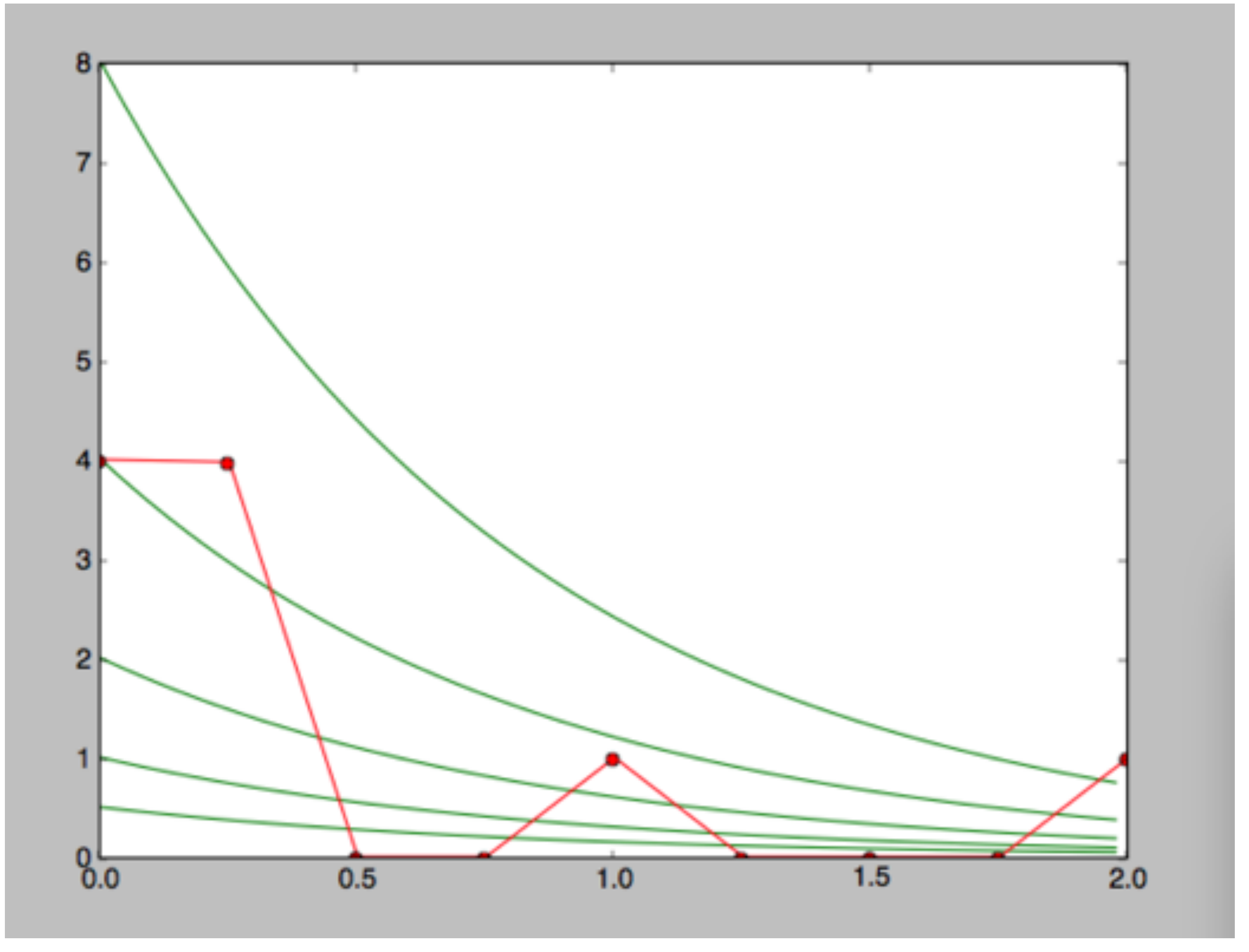


Method stability

- Even with perfectly well-behaved functions, some **methods** can be unstable

- Errors grow without bound

- Often see oscilatory behaviour



Euler's Method

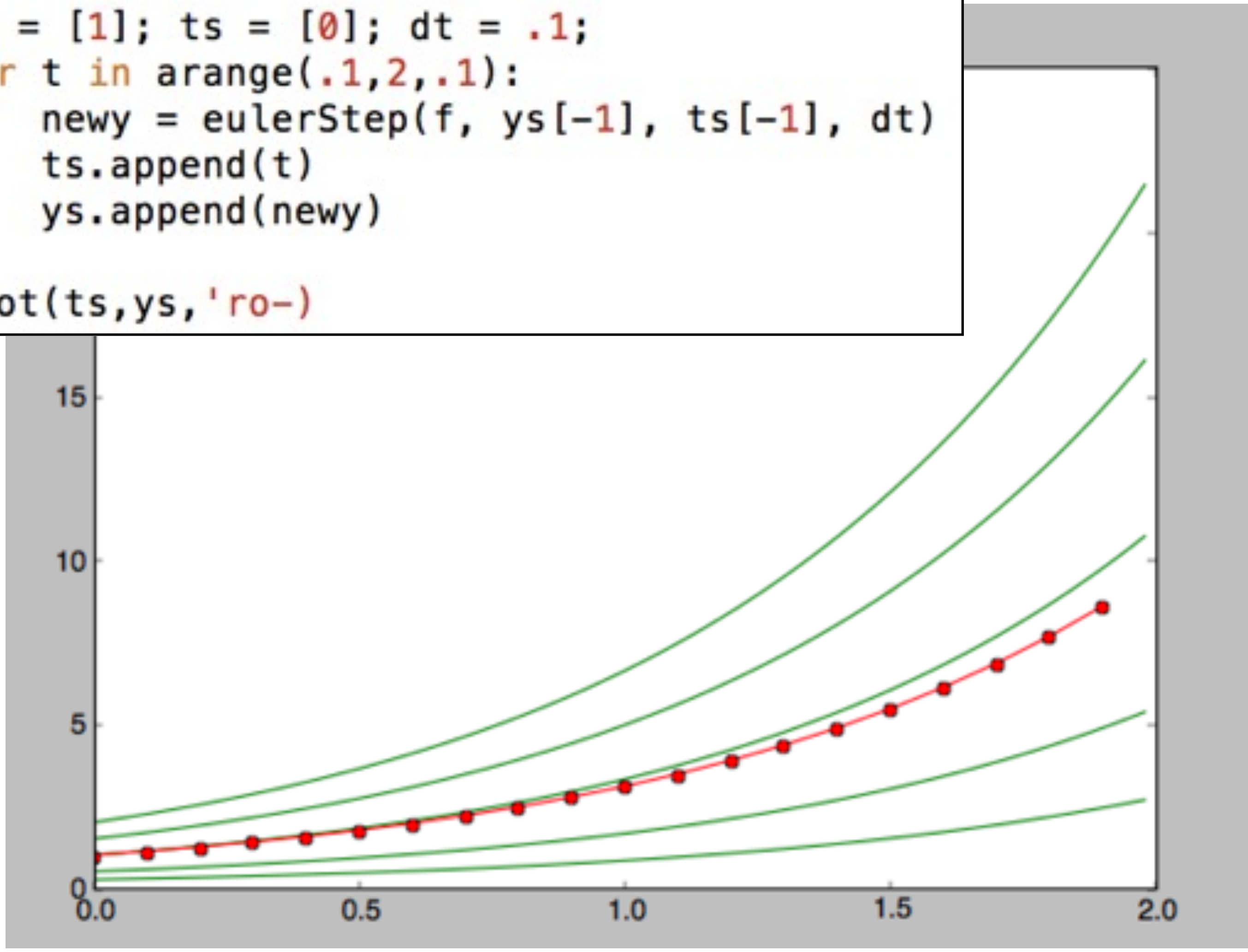
- Simplest possible integration method
- stepsize *h*
- Calculate local deriviative, and approximate (first term in Taylor's series):

$$\left.\frac{d\mathbf{y}}{dt}\right|_{(y_0,t_0)} = \mathbf{f}(\mathbf{y}_0, t_0)$$

$$\mathbf{y}(t_0 + h) \approx \mathbf{y}_0 + h\left.\frac{d\mathbf{y}}{dt}\right|_{(y_0,t_0)}$$

$$\approx \mathbf{y}_0 + h\mathbf{f}(\mathbf{y}_0, t_0)$$

```
def eulerStep(f, yo, to, dt):
    dydt = f(yo, to)
    return yo + dydt * dt

def f(y,t):
    return 1.2*y

xx = arange(0,2,.025)
for y in [.25,.5,1,1.5,2]:
    plot(xx,y*exp(1.2*xx),'g-')

ys = [1]; ts = [0]; dt = .1;
for t in arange(.1,2,.1):
    newy = eulerStep(f, ys[-1], ts[-1], dt)
    ts.append(t)
    ys.append(newy)

plot(ts,ys,'ro-)
```



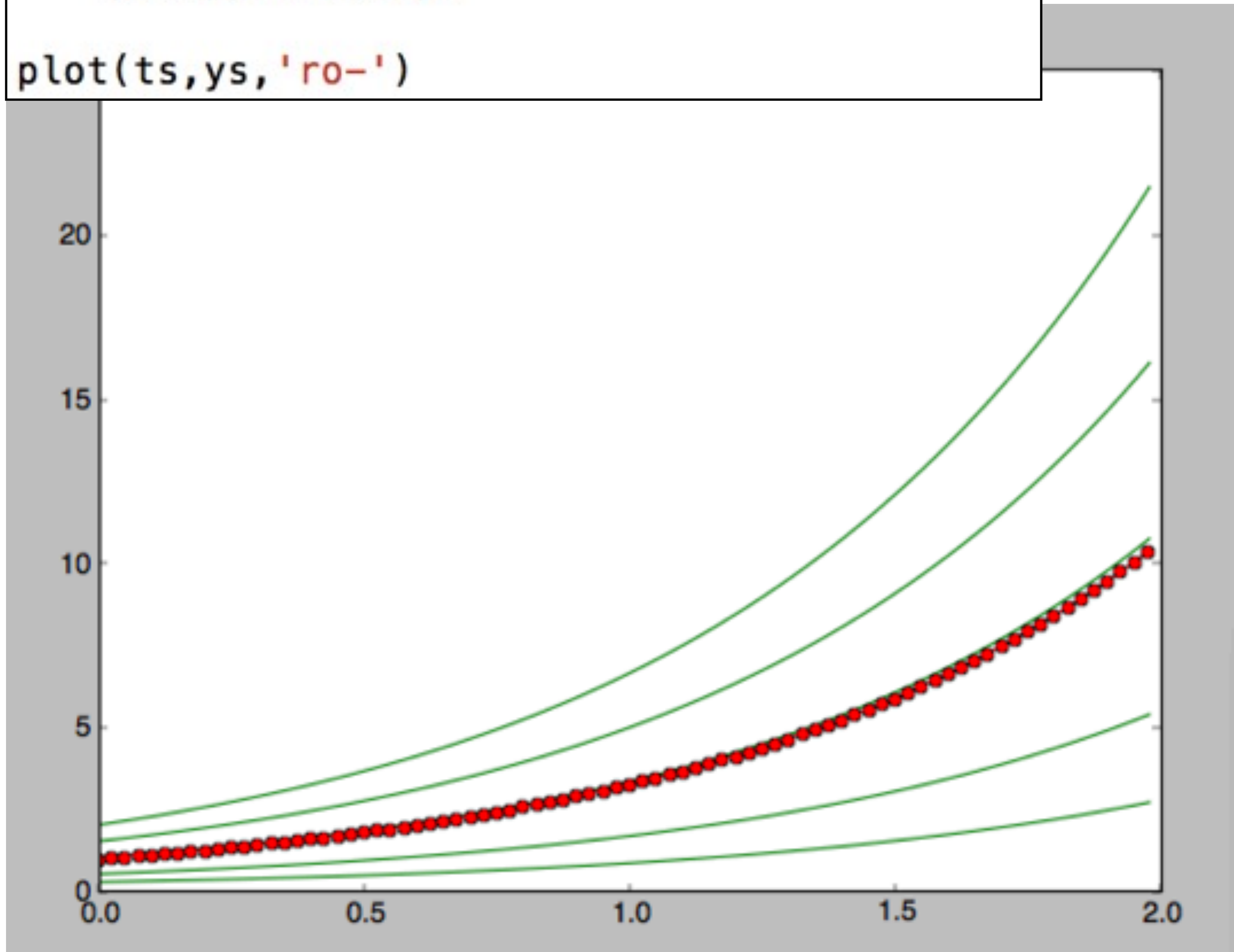
Accuracy

- Accuracy improves with smaller stepsize
- As with interpolation, error in a linear step from Taylor series is

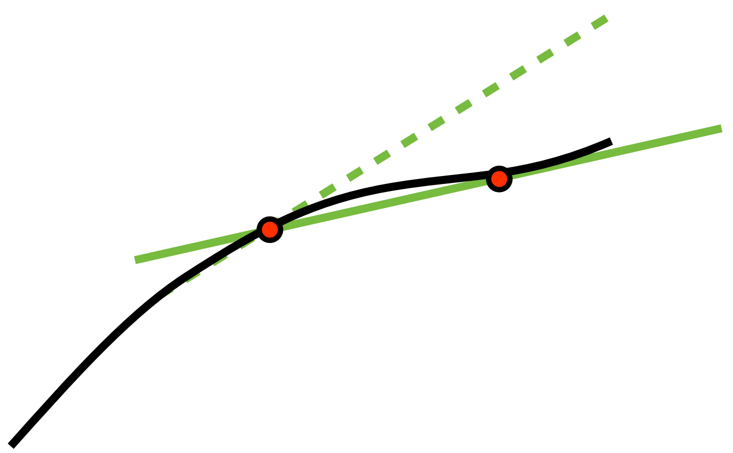
$$\mathcal{O}(h^2)$$

- "Too large" h - unstable.
- Also as with interpolation, can improve accuracy with higher-order methods.

```
ys = [1]; ts = [0]; dt = .025;

for t in arange(dt,2,dt):
    newy = eulerStep(f, ys[-1], ts[-1], dt)
    ts.append(t)
    ys.append(newy)

plot(ts,ys,'ro-')
```



Backward Euler



- Solve for step implicitly
- Take slope approximation as slope at **new** point
- Same *accuracy* as forward Euler, better *stability*

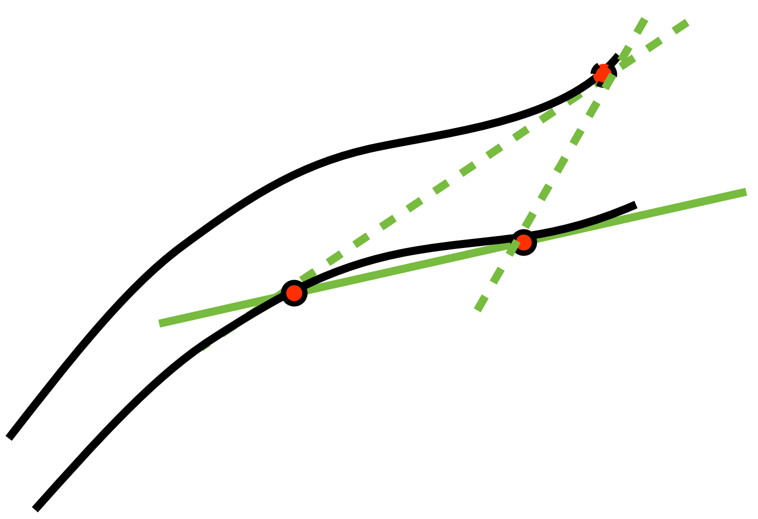
$$\left.\frac{d\mathbf{y}}{dt}\right|_{(y_0+\Delta y, t_0+h)} = \mathbf{f}(\mathbf{y}_0 + \Delta y, t_0 + h)$$

$$\mathbf{y}(t_0 + h) \approx \mathbf{y}_0 + h \left.\frac{d\mathbf{y}}{dt}\right|_{(y_0+\Delta y, t_0+h)}$$

$$\mathbf{y_0} + \Delta\mathbf{y} \approx \mathbf{y}_0 + h\mathbf{f}(\mathbf{y}_0 + \Delta y, t_0 + h)$$

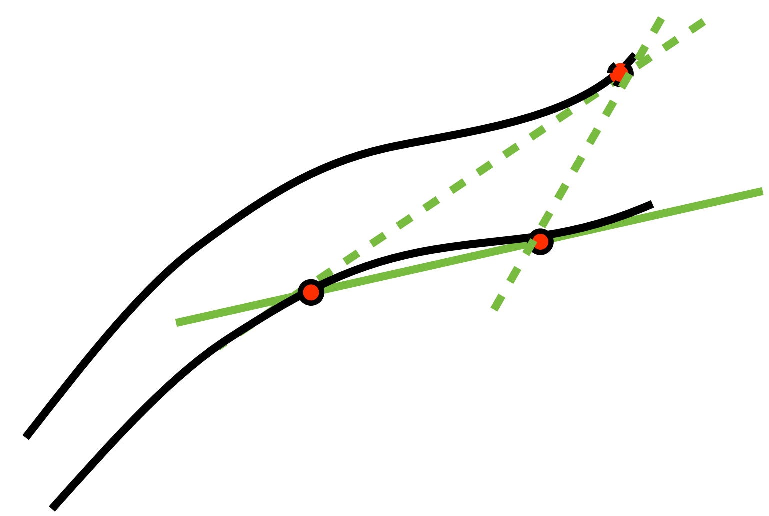
Predictor-Corrector

- As with interpolation, can get higher accuracy by using more points
- Can evaluate **f** anywhere
- Predictor-corrector: take forward Euler step, use f value there to improve estimate.



Error estimation

- Note! With multiple function evaluations, one can use different combinations of them to derive different estimates.
- Can use higher- and lower- order methods, and use difference to infer error in estimate.
- This allows adaptive stepsizing to satisfy an error tolerance. Redo with smaller step if error too large.
- Without error estimate, all one can do is say whether solution "looks good" or not.



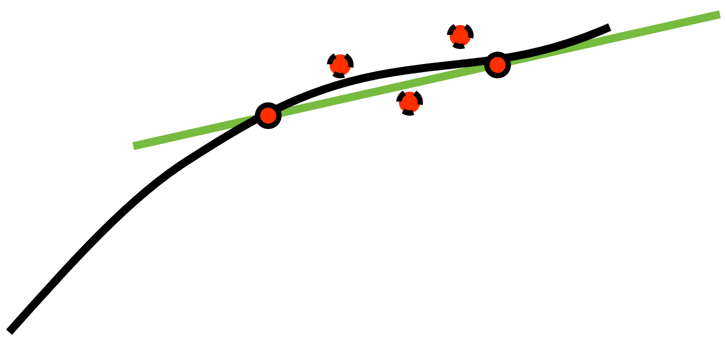
Multi-step methods

- More complex approaches are tradeoffs between stability, accuracy, and cost (function evaluation or nonlinear solves)

- Take multiple function evaluations between t and t+Δt, and use the combination of those to get next value

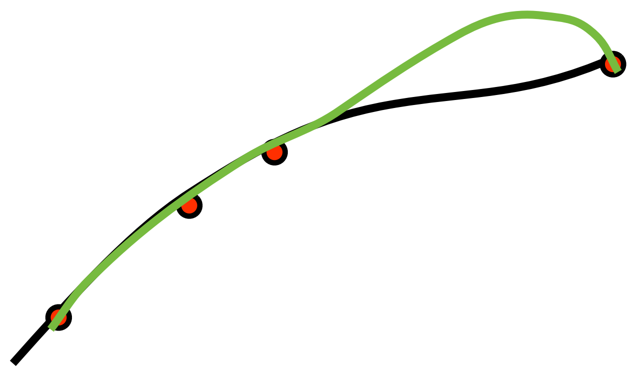
- Runge-Kutta methods are classics of these kinds.

- Again, can return error estimates.



Multi-stage methods

- Multiple function evaluations "for free"; use previous evaluations!
- Require something special to start.



Don't Repeat Yourself (Or Others)

- ODEs, interpolation *very* common
- Very well established techniques, code, for doing this.
- Except for most trivial cases, do **not** code yourself. Libraries will do this for you.
- GSL (gnu scientific library) ubiquitious, has several methods for both.
- Allows you to easily experiment with different methods without rewriting code.

Interpolation

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_spline.h>

int
main (void)
{
  int i;
  double xi, yi, x[10], y[10];

  printf ("#m=0,S=2\n");

  for (i = 0; i < 10; i++)
    {
      x[i] = i + 0.5 * sin (i);
      y[i] = i + cos (i * i);
      printf ("%g %g\n", x[i], y[i]);
    }

  printf ("#m=1,S=0\n");

  {
    gsl_interp_accel *acc
      = gsl_interp_accel_alloc ();
    gsl_spline *spline
      = gsl_spline_alloc (gsl_interp_cspline, 10);

    gsl_spline_init (spline, x, y, 10);

    for (xi = x[0]; xi < x[9]; xi += 0.01)
      {
        yi = gsl_spline_eval (spline, xi, acc);
        printf ("%g %g\n", xi, yi);
      }
    gsl_spline_free (spline);
    gsl_interp_accel_free (acc);
  }
  return 0;
```

SciNet
compute • calcul
CANADA

ODE Integration

```c
int
func (double t, const double y[], double f[],
      void *params)
{
  double mu = *(double *)params;
  f[0] = y[1];
  f[1] = -y[0] - mu*y[1]*(y[0]*y[0] - 1);
  return GSL_SUCCESS;
}

int
jac (double t, const double y[], double *dfdy,
     double dfdt[], void *params)
{
  double mu = *(double *)params;
  gsl_matrix_view dfdy_mat
    = gsl_matrix_view_array (dfdy, 2, 2);
  gsl_matrix * m = &dfdy_mat.matrix;
  gsl_matrix_set (m, 0, 0, 0.0);
  gsl_matrix_set (m, 0, 1, 1.0);
  gsl_matrix_set (m, 1, 0, -2.0*mu*y[0]*y[1] - 1.0);
  gsl_matrix_set (m, 1, 1, -mu*(y[0]*y[0] - 1.0));
  dfdt[0] = 0.0;
  dfdt[1] = 0.0;
  return GSL_SUCCESS;
}

int
main (void)
{
  double mu = 10;
  gsl_odeiv2_system sys = {func, jac, 2, &mu};

  gsl_odeiv2_driver * d =
    gsl_odeiv2_driver_alloc_y_new (&sys, gsl_odeiv2_step_rk8pd,
                                   1e-6, 1e-6, 0.0);
  int i;
  double t = 0.0, t1 = 100.0;
  double y[2] = { 1.0, 0.0 };

  for (i = 1; i <= 100; i++)
    {
      double ti = i * t1 / 100.0;
      int status = gsl_odeiv2_driver_apply (d, &t, ti, y);

      if (status != GSL_SUCCESS)
	{
	  printf ("error, return value=%d\n", status);
	  break;
	}

      printf ("%.5e %.5e %.5e\n", t, y[0], y[1]);
    }

  gsl_odeiv2_driver_free (d);
  return 0;
}
```