Phys 512 Lecture 2+3 Polynomial Interpolation/ Integration

Admin

- Tutorials - Thursdays at 4 PM in the piano room
- TA office hours - Tuesdays 11-12
- Lecturer office hours - Tuesdays 1 PM

  Note - info now on the syllabus

Interpolation

- Let's say we have a function y_i tabulated at some values x_i but we want to know it at arbitrary x.

- If the new values are between current values, we call this interpolation. If outside, extrapolation. Extrapolation decidedly dodgier.

- How should we do this?

Simple Solutions

- Dumbest thing we can do is find the x_i nearest our target x, and take y_i. This is nearest neighbor. Is this going to be accurate? Is it ever going to go crazy?

- Next dumbest thing is find the interval x_i, x_{i+1} that x sits in. Then draw a line between (x_i,y_i) and (x_{i+1},y_{i+1}) and evaluate it at x. This is linear interpolation. Will this be accurate? Is it every going to go crazy?

- NB - by “crazy” we usually mean that the interpolated y is far from the spread covered by y_i and its neighbors.

- For linear, y will be between y_i and y_{i+1}, so it can’t go crazy.

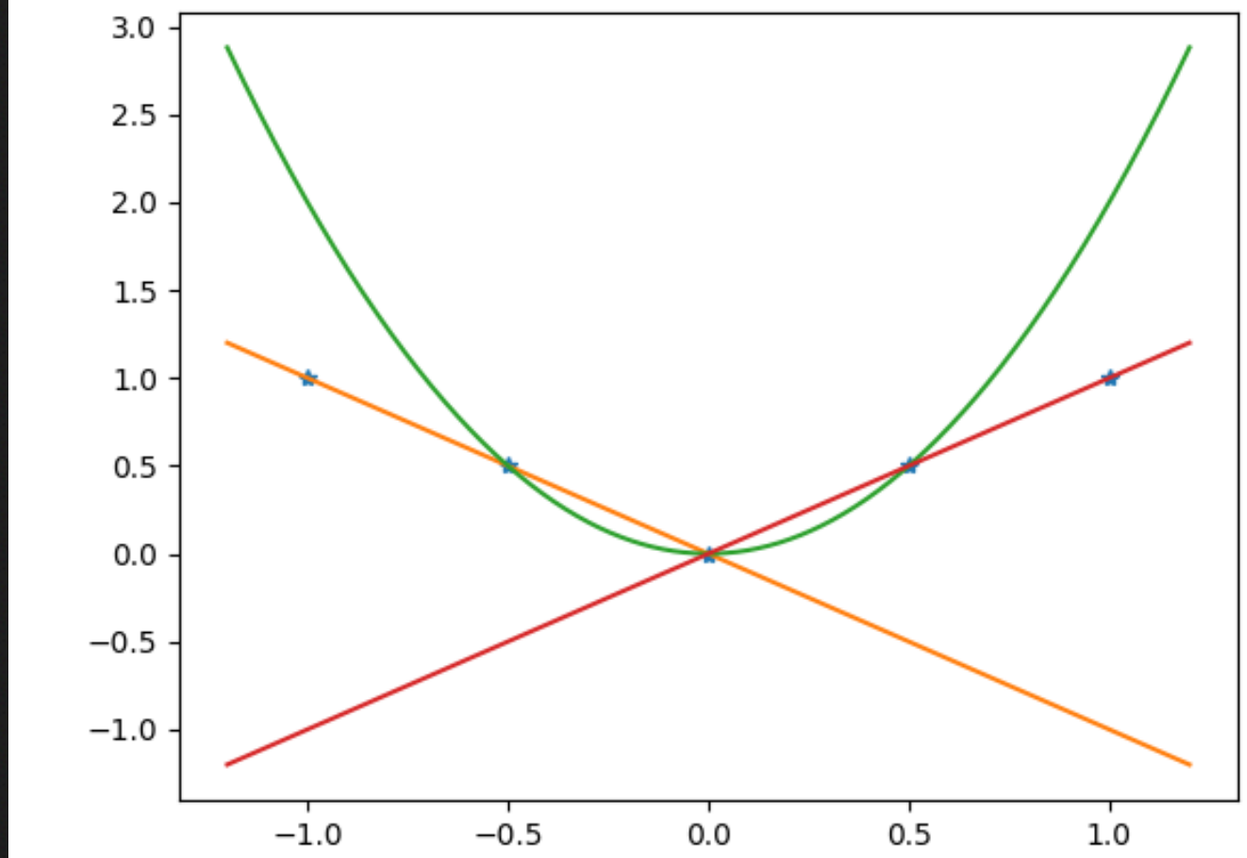
Higher Order

- If we want more accuracy ***AND*** our function is “smooth”, we can win by Taylor expanding.

- Next order is quadratic. We could take y_{i-1}, y_i, y_{i+1} and draw a parabola.

- This is almost certainly a bad idea! Why?

```python
import numpy as np
from matplotlib import pyplot as plt

x=np.linspace(-1,1,5)
y=np.sqrt(np.abs(x))
y=np.abs(x)

xx=np.linspace(-1.2,1.2,1000)

plt.clf()
plt.plot(x,y,'*')
for i in range(len(x)-2):
    pp=np.polyfit(x[i:i+3],y[i:i+3],2)
    yy=np.polyval(pp,xx)
    plt.plot(xx,yy)
plt.savefig('parabolas.png')
```

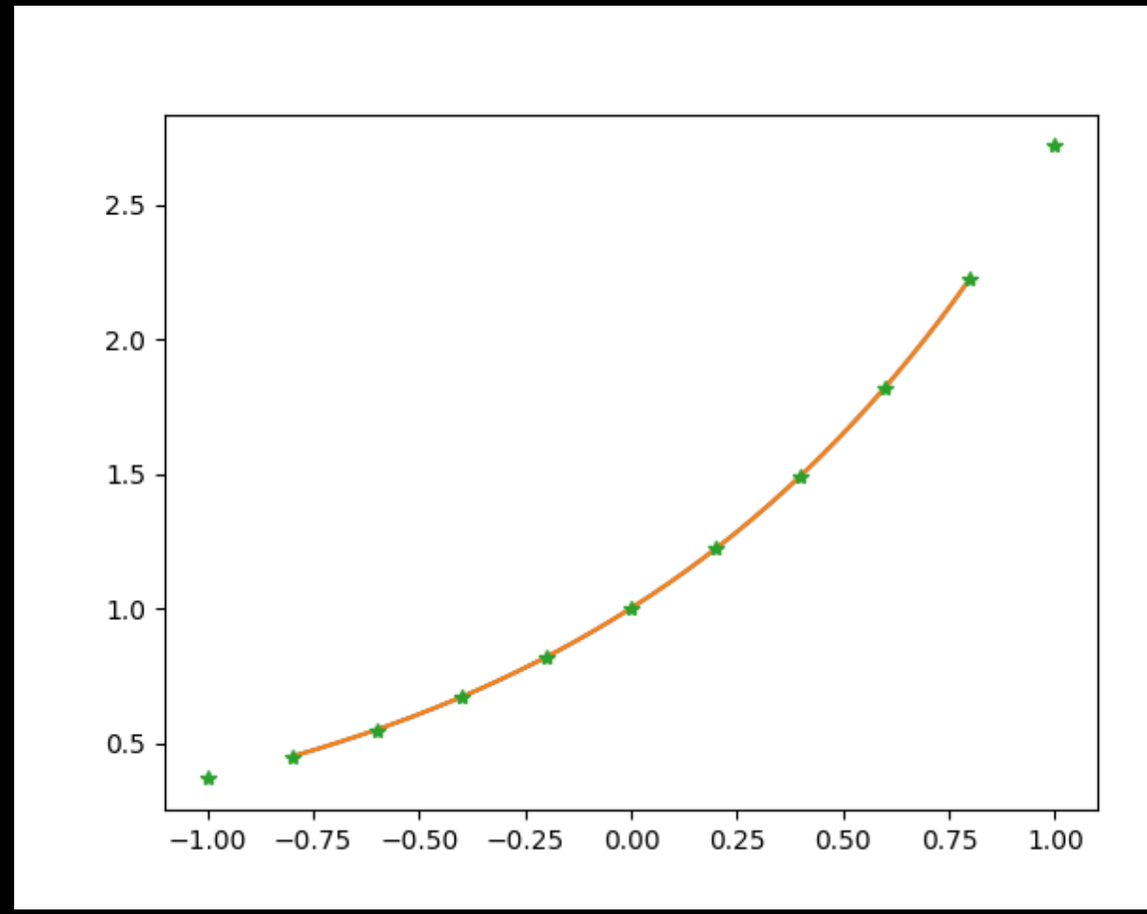


How would I decide on x_i? Natural thing is to take nearest neighbor.

However, that changes halfway between points. I then move to a different parabola. In general, interpolation will be discontinuous. This is (almost certainly) bad!

NB - no guarantee anymore that y is bracketed by function values.

```python
fun=np.exp
xmin=-1
xmax=1
x=np.linspace(xmin,xmax,11)
dx=np.median(np.diff(x))
y=fun(x)

xx=np.linspace(x[1],x[-2]-1e-13,1001) #skip the first/last region
                                      #since we aren't double-bracketed

yy_true=fun(xx)
yy=0*yy_true
for i,myx in enumerate(xx):
    j=np.int((myx-xmin)/dx)
    pp=np.polyfit(x[j-1:j+3],y[j-1:j+3],3)
    yy[i]=np.polyval(pp,myx)
plt.clf();
plt.plot(xx,yy_true);
plt.plot(xx,yy);
plt.savefig('cubic_interp.png')
print('mean error is ' + repr(np.mean(np.abs(yy-yy_true))))
```



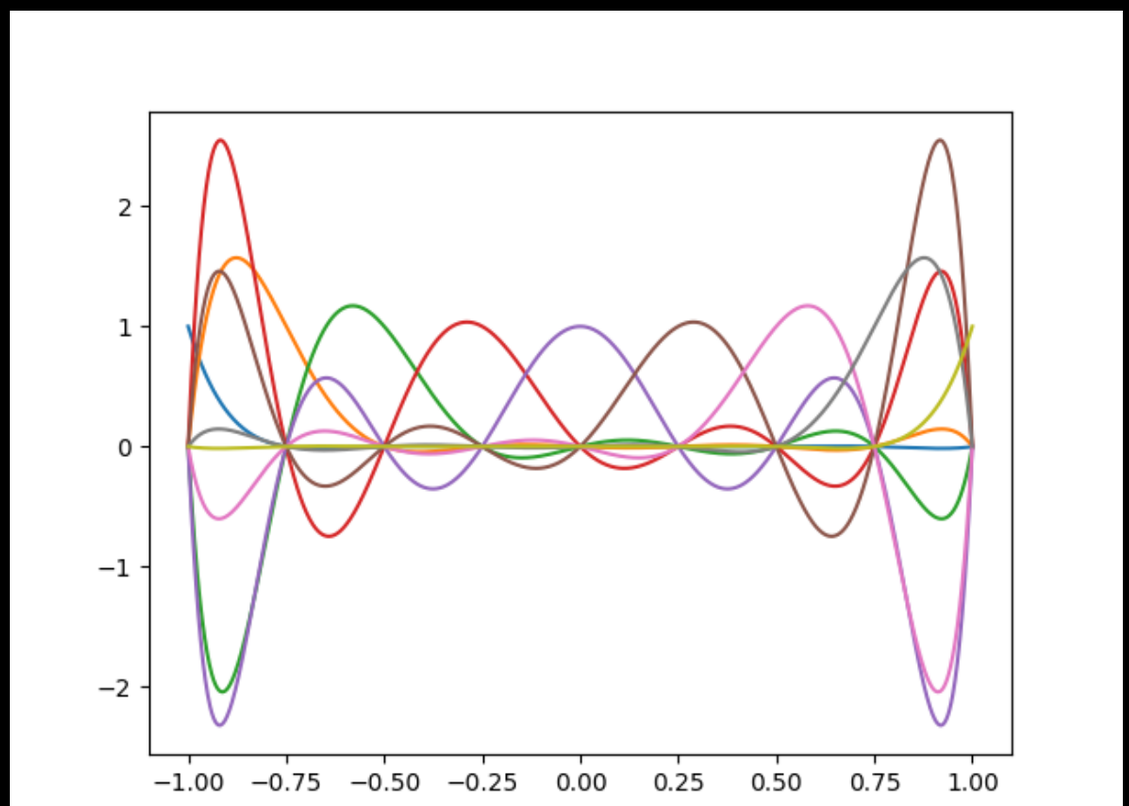
- Cubic polynomials don’t suffer from discontinuity since interval is well defined.
- Mean error has done pretty well! ~1e-5
- However - we are doing a polynomial fit for every point. This can be quite slow.

Simple Polynomial Construction

- How can I construct a polynomial that goes through *n* points?
- Well, I can construct a polynomial that is zero at *n*-1 points, say x_1 through x_{n-1}. NB - *n*-1 is because we zero index.
- $(x-x_1)(x-x_2)(x-x_3)..(x-x_{n-1})$ by construction is zero.
- What is it at x_0? well, $(x_0-x_1)(x_0-x_2)\ldots(x_0-x_{n-1})$.
- If I take the ratio, I have a polynomial P_0 that is 1 at x_0 and zero at all other x_i.
- I can now take $\sum P_i y_i$ to have a polynomial that goes exactly through all y_i.

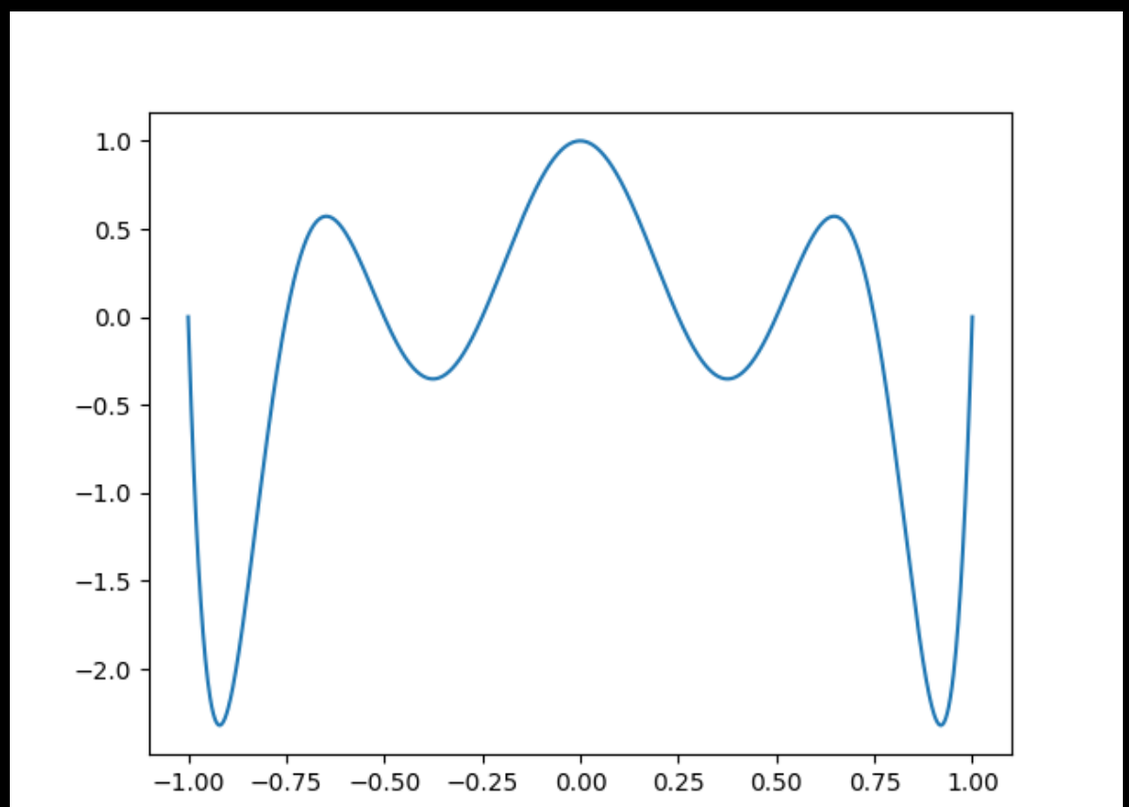
What Those Look Like

```
ord=8
x=np.linspace(-1,1,ord+1)
xx=np.linspace(x[0],x[-1],1001)
plt.clf()
for i in range(len(x)):
    x_use=np.append(x[:i],x[i+1:])
    x0=x[i]
    mynorm=np.prod(x0-x_use)
    p0=1.0
    for xi in x_use:
        p0=p0*(xi-xx)
    p0=p0/mynorm
    plt.plot(xx,p0)
    if i==4:
        bad_p0=p0.copy()

plt.savefig('delta_polys_out.png')
plt.clf()
plt.plot(xx,bad_p0)
plt.savefig('delta_polys_one.png')
```



- This should make you nervous. At high order, polys could jiggle quite a bit between points. OK iff your function is smooth, in the sense that (ord**dx*)^n f$^{(n)}$/*n*! is converging.

- What happens if your values have errors/noise?



Poly Interpolation ctd.

- Cubic interpolation from earlier look OK. At least was continuous.
- I didn't need to do the polynomial fit for each interpolated value. Rather, I could fit every interval and store coefficients.
- Then, when interpolating, look up the segment, take the relevant coefficients, and evaluate.
- Note that nowhere have we assumed the x_i are equally spaced. The lookup is faster when they are, but still only $\log_2(n)$ when they aren't.

Cubic Splines

- My old cubic interpolation was continuous. Were its derivatives?

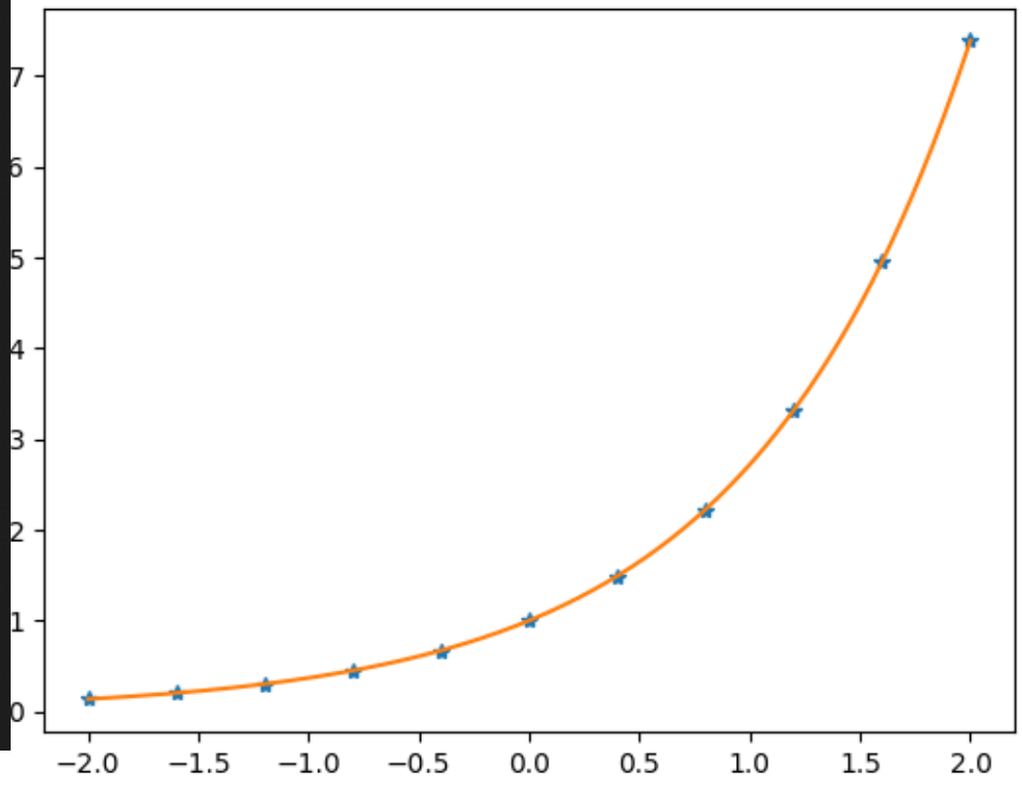
- An important class of interpolation forces the function and the first *n* derivatives to be continuous. These are called splines.

- By far most common is requirement that second derivate is continuous (so first deriv is guaranteed smooth). Specifying function values and derivatives at edges is 2+2=4 values, so we will need a cubic to fit them.

- Cubic splines are so common, we usually drop the cubic, just call them splines.

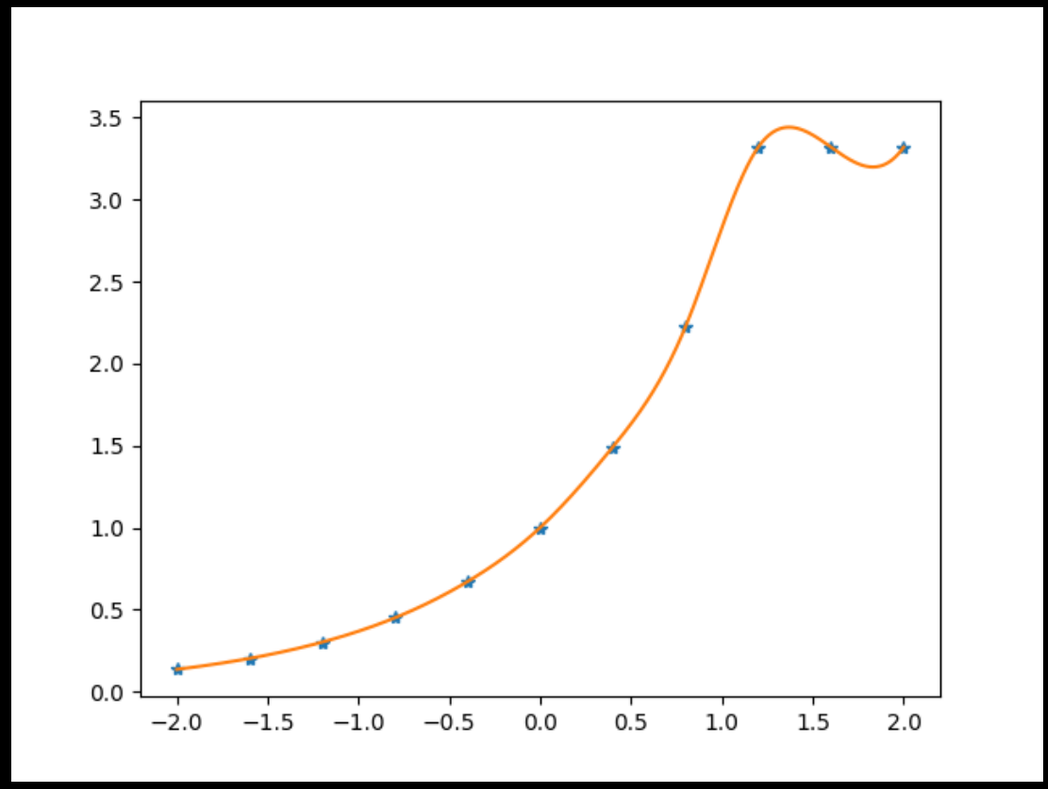
- My 2nd deriv has to match left and right neighbors. But what about at ends? You need to specify. Most common is set *f*” to zero on edges, but can also set *f*’ to specific values, or require periodic boundary conditions (right edge has to match left).

```
import numpy as np
from matplotlib import pyplot as plt
from scipy.interpolate import CubicSpline
from scipy import interpolate
plt.ion()

x=np.linspace(-2,2,11)
y=np.exp(x)
xx=np.linspace(x[0],x[-1],2001)
spln=CubicSpline(x,y)
yy=spln(xx)

plt.clf();
plt.plot(x,y,'*')
plt.plot(xx,yy)
```



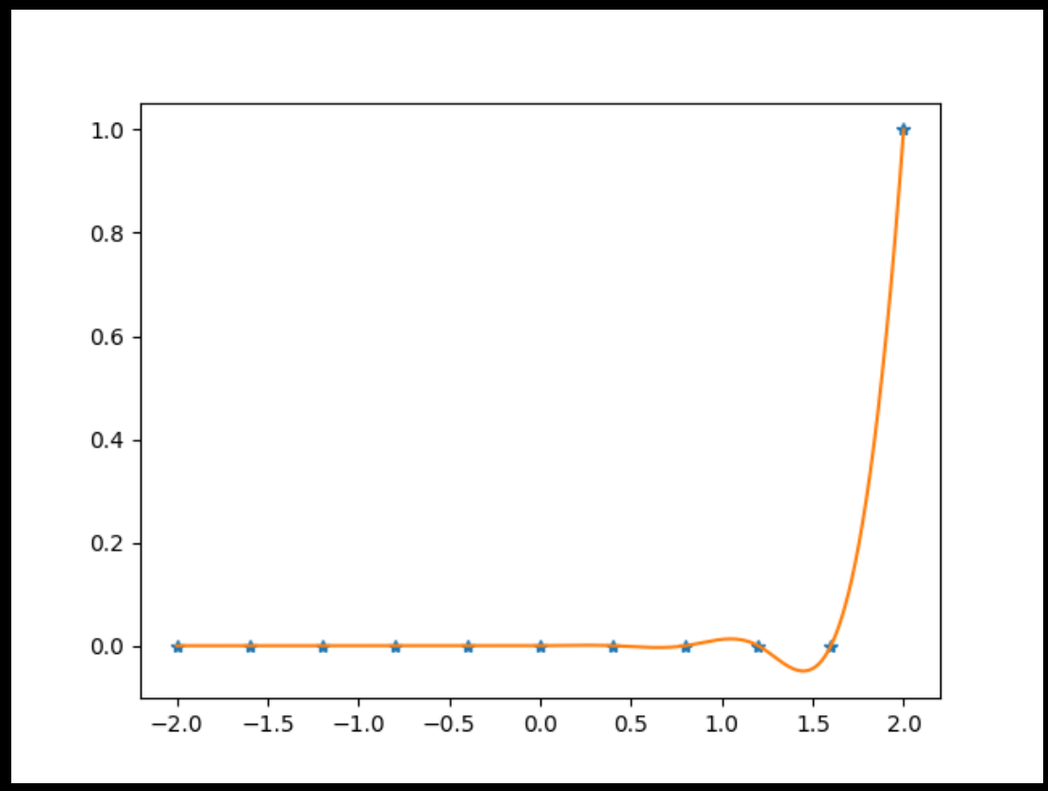
- Spline in use. Lives in scipy.interpolate
- First, set up the spline fit with splrep.
- Then evaluate with splev.
- Broader range of options in BSpline class (also in scipy.interpolate).

```
x=np.linspace(-2,2,11)
y=np.exp(x)
y[-2:]=y[-3]
xx=np.linspace(x[0],x[-1],2001)
spln=CubicSpline(x,y)
yy=spln(xx)
```



- Be aware - splines can ring near jumps/kinks. This might be OK, this might not be. But you should not be caught by surprise.

- Another option is cubic Hermite polynomials - similar to spline, but $y(x \in (x_1,x_2)) \in (y_1,y_2)$ (scipy.interpolate.PchipInterpolator)



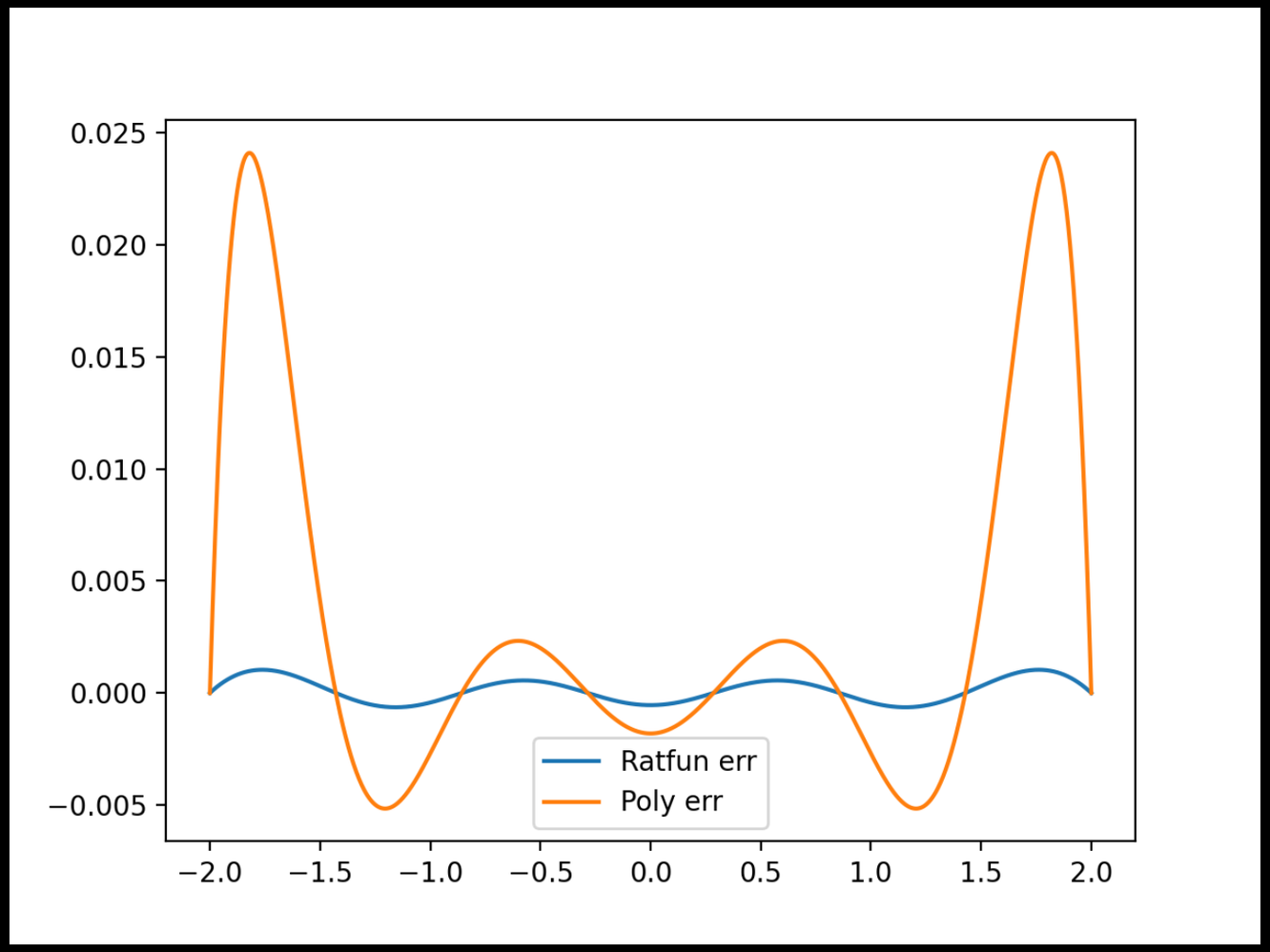
Rational Functions

- Another useful class is rational functions, ratios of two polynomials.
- Can be especially useful when your function has poles.
- Nominally non-linear: $p(x)/q(x)=y(x)$. Without loss, set $q(x)=1+ax+bx^2+\ldots = 1 + qq(x)$
- For simple case of number of degrees matches number of points can get around: $p(x)=q(x)y(x) = y(x)+qq(x)y(x)$
- Matrix: $[1\ x\ x^2\ x^3\ldots\ -yx\ -yx^2\ -yx^3\ldots][p;q]=y$

```python
def rat_eval(p,q,x):
    top=0
    for i in range(len(p)):
        top=top+p[i]*x**i
    bot=1
    for i in range(len(q)):
        bot=bot+q[i]*x**(i+1)
    return top/bot

def rat_fit(x,y,n,m):
    assert(len(x)==n+m-1)
    assert(len(y)==len(x))
    mat=np.zeros([n+m-1,n+m-1])
    for i in range(n):
        mat[:,i]=x**i
    for i in range(1,m):
        mat[:,i-1+n]=-y*x**i
    pars=np.dot(np.linalg.inv(mat),y)
    p=pars[:n]
    q=pars[n:]
    return p,q
```

```python
n=4
m=5
x=np.linspace(-2,2,n+m-1)
y=np.exp(-0.5*x**2)
p,q=rat_fit(x,y,n,m)
pred=rat_eval(p,q,x)


xx=np.linspace(-2,2,1001)
yy=np.exp(-0.5*xx**2)
yy_interp=rat_eval(p,q,xx)
plt.ion()
plt.clf()
plt.plot(xx,yy_interp-yy)
```

```python
#we can use numpy's polynomial fitter to see how that does
pp=np.polyfit(x,y,n+m) #use same number of terms
yy_poly=np.polyval(pp,xx)
plt.plot(xx,yy_poly-yy)
plt.savefig('ratfit_vs_poly.png')
```



Integration

- Interpolation & numerical integration are closely coupled
- Interpolation schemes are often easy to integrate analytically.
- Usually end up as a set of coefficients times function values, where coefficients are set by scheme.
- Can think of this as finding the “average” value in a region, based on some interpolation scheme.

Integration with Linear

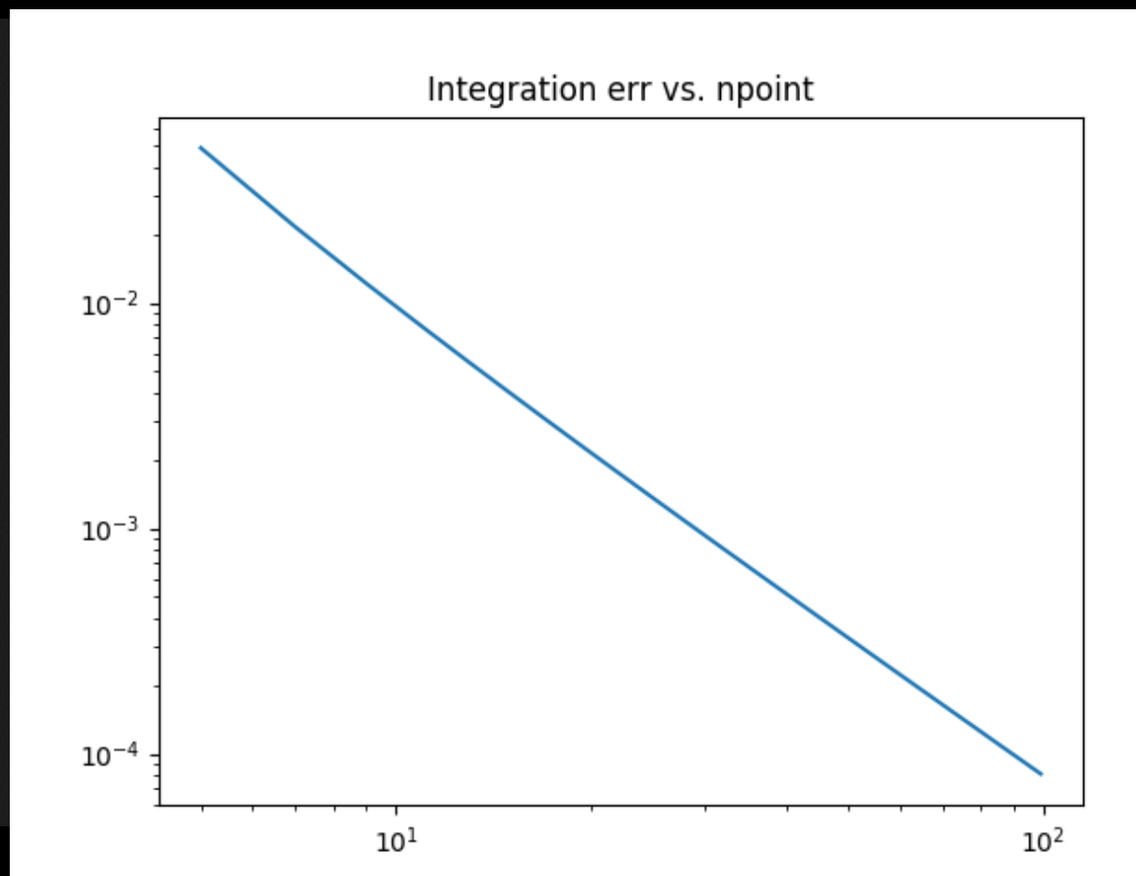
- Break function into series of regions, draw a line between points.

- What is the average value of y if it is based on a line between y_i and y_{i+1}? just $0.5(y_i+y_{i+1})$.

- If points are dx apart, then area/dx=$0.5(y_0+y_1)+0.5(y_1+y_2)\ldots+0.5(y_{n-2}+y_{n-1})=0.5(y_0+y_{n-1})+\sum y_i$, i=1..n-2.

- How should error scale with # of points?

```
x0=-1
x1=1
nn=np.arange(5,101,2)
errs=np.zeros(nn.size)
for i,npt in enumerate(nn):
    x=np.linspace(x0,x1,npt)
    y=np.exp(x)
    dx=np.median(np.diff(x))
    myint=0.5*(y[0]+y[-1])+np.sum(y[1:-1])
    myint=myint*dx

    targ=np.exp(x1)-np.exp(x0)
    errs[i]=np.abs(myint-targ)
plt.loglog(nn,errs)
plt.title('Integration err vs. npoint')
plt.savefig('linear_integral_errs.png')
pp=np.polyfit(np.log10(nn),np.log10(errs),1)
print('error is scaling as step size to the power ' + repr(pp[0]))
```



- Scaling is going as step size squared, as expected.

But Wait!

- If error is going like dx^2, can I use that for fun and profit?
- $f(dx)=f_{true}+adx^2+\ldots$ $f(2dx)=f_{true}+a(2dx)^2+\ldots$
- $4f(dx)-f(2dx)=4f_{true}-f_{true}+\ldots=3f_{true}+\ldots$ or $f_{true}=(4f(dx)-f(2dx))/3$
- 3 points:$f(dx)=(y_0+2y_1+y_2)dx/2$. $f(2dx)=(y_0+y_2)dx$ (NB - dx went to 2 dx, cancelling usual factor of 2)
- $4f(dx)-f(2dx)=(2y_0+4y_1+2y_2-y_0-y_2)dx/3=(y_0+4y_1+y_2)dx/3$
- We have cancelled 2nd order error term. Should be more accurate.

Equivalent

- We could also fit a parabola to 3 points. For simplicity take x=-1,0,1. There's a trapezoid going through y_0 and y_2, with a parabola that goes through y_1 minus $(y_0+y_2)/2$.

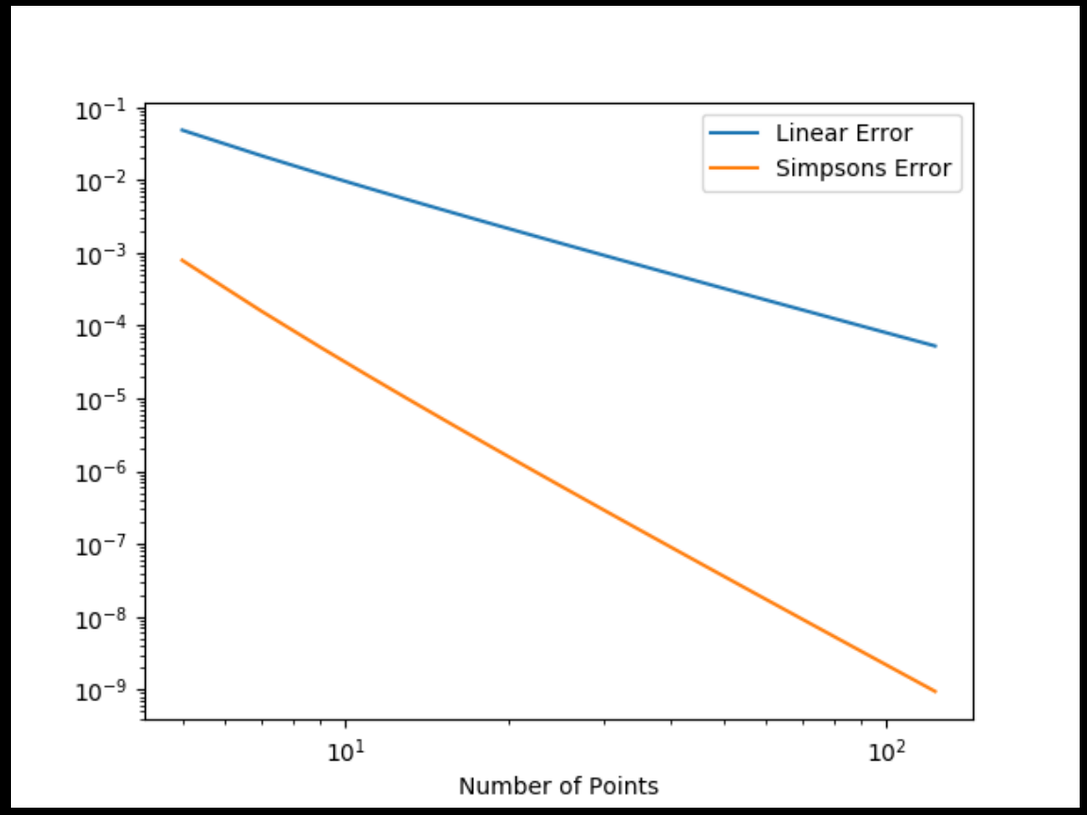
- Parabola is $(y_1-(y_0+y_2)/2)(1-x^2)$. Integral is $(x-x^3/3)$, evaluates to 4/3 over 2dx, so average is 2/3. Leaves $(y_0+y_2)/2*(2dx)+2/3(y_1-(y_0+y_2)/2)2dx=dx(y_0+4y_1+y_2)/3$.

- This is exactly what we had before! Cancelling 2nd order error term is identical to fitting quadratic & integrating.

- This second-order scheme is called Simpson's rule.

- How should error scale? Hint - what is the average value of x^3?

```python
nn=np.arange(5,125,2)
x0=-1
x1=1
ints_lin=np.zeros(len(nn))
ints_quad=np.zeros(len(nn))
for i,npt in enumerate(nn):
    x=np.linspace(x0,x1,npt)
    y=np.exp(x)
    dx=np.median(np.diff(x))
    ints_lin[i]=dx*(0.5*y[0]+0.5*y[-1]+np.sum(y[1:-1]))
    ints_quad[i]=dx/3.0*(y[0]+y[-1]+4*np.sum(y[1::2])+2*np.sum(y[2:-1:2]))
targ=np.exp(x1)-np.exp(x0)
plt.clf();
errs_lin=np.abs(ints_lin-targ)
errs_quad=np.abs(ints_quad-targ)
pp=np.polyfit(np.log(nn),np.log(errs_quad),1)
print('Simpsons scaling is ' + repr(pp[0]))
plt.loglog(nn,errs_lin)
plt.loglog(nn,errs_quad)
plt.xlabel('Number of Points')
plt.legend(['Linear Error','Simpsons Error'])
plt.savefig('simpson_errs.png')
```



- Simpsons is far more accurate than trapezoid.

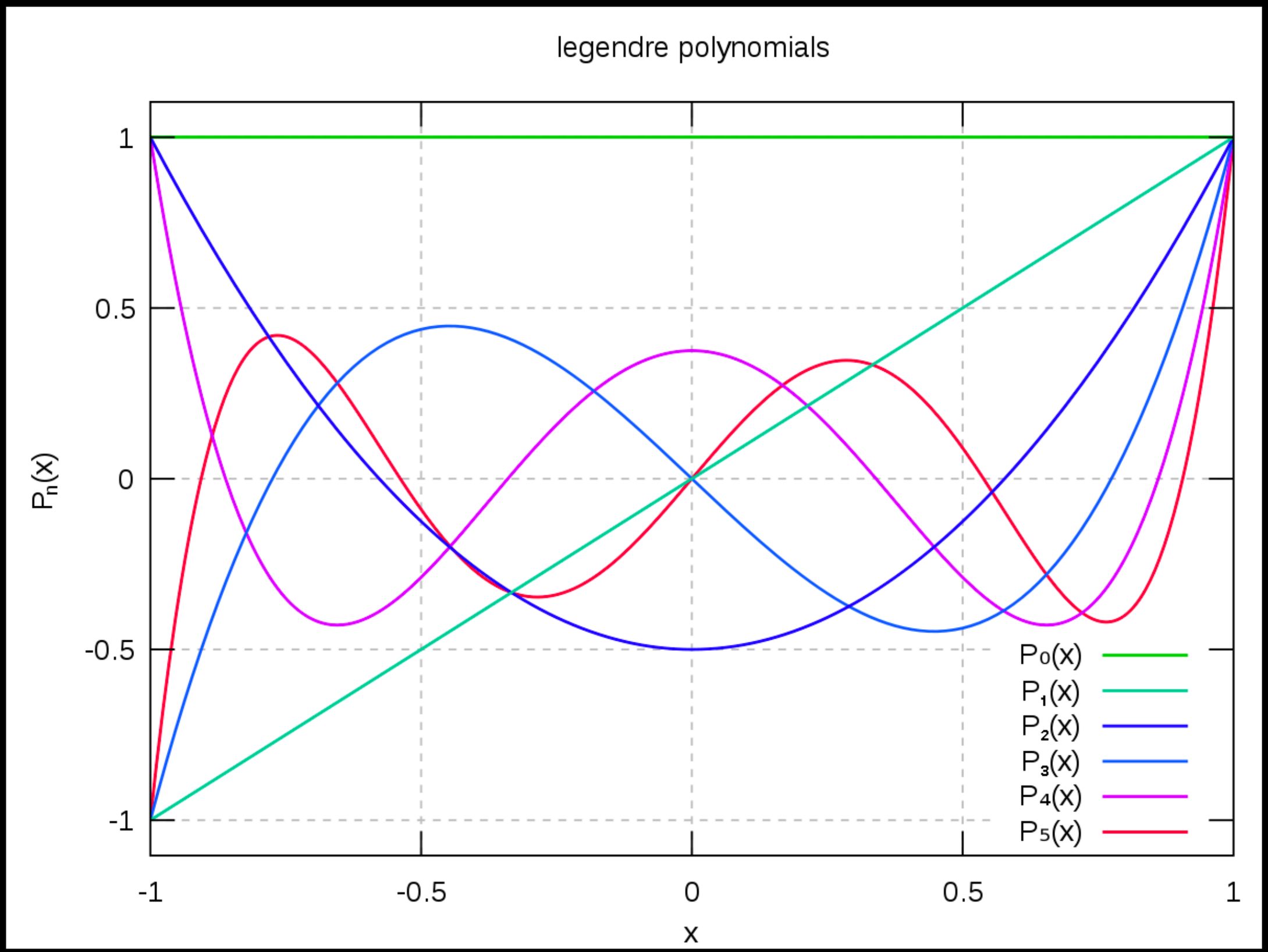
- Error goes like dx^4, since x^3 term integrates to zero.

- I didn’t have to make any more function calls, just used the ones I had more cleverly.

- NB - would this work with an even number of y_i?

Can We Turn it up to 11?

- Yes! Regular polynomials ($x, x^2, x^3 \ldots$) are not what we want, though.
- Legendre polynomials are an important class. They arise in separation of variables in spherical coordinates.
- We could think of regular polynomials as defined by a recurrence relation. $F_{n+1}=xF_n$.
- Legendre polynomials best generated with a different recurrence relation. $(n+1)P_{n+1}=(2n+1)xP_n-nP_{n-1}$. $P_0=1$, $P_1=x$.


legendre polynomials
1
0.5
0
-0.5
-1
$P_n(x)$
-1
-0.5
0
0.5
1
x
$P_0(x)$
$P_1(x)$
$P_2(x)$
$P_3(x)$
$P_4(x)$
$P_5(x)$

Legendre ctd.

- They have the important property that they are orthogonal on the interval (-1,1). $\int P_n P_m \propto \delta_{nm}$

- Given this, what is $\int P_n dx$ from -1 to 1?

- 2 for P_0, and zero for everything else.

- I can fit Legendre polynomials to a set of data, and integral will just be P_0 coefficient(!).

- How do I get this?

Legendre Fitting

- $y_i=\sum c_j P_j(x_i)$. If I can figure out the c_j then I'm in business.
- But this is just a matrix equation! $y=Pc$. If we have as many polynomials as we have points, then the matrix P is square, and we can just get $c=P^{-1}y$. Pull c_0, and then we're done.
- But, c_0 is just $\sum P^{-1}_{0,k} y_k$, so I can just take the first column of P^{-1}. This gives me my weights.
- Now I can integrate to whatever order I want (making sure I have a suitable number of y_i for my chosen order).

Code to Make Coeffs

```
def legendre_mat(npt):
    #Make a square legendre polynomial matrix of desired dimension
    x=np.linspace(-1,1,npt)
    mat=np.zeros([npt,npt])
    mat[:,0]=1.0
    mat[:,1]=x
    if npt>2:
        for i in range(1,npt-1):
            mat[:,i+1]=((2.0*i+1)*x*mat[:,i]-i*mat[:,i-1])/(i+1.0)
    return mat

def integration_coeffs_legendre(npt):
    #Find integration coefficients using
    #square legendre polynomial matrix
    mat=legendre_mat(npt)
    mat_inv=np.linalg.inv(mat)
    coeffs=mat_inv[0,:]
    coeffs=coeffs/coeffs.sum()*(npt-1.0)
    return coeffs
```

Code to Integrate Stuff

```
def integrate(fun,xmin,xmax,dx_targ,ord=2,verbose=False):
    coeffs=legendre.integration_coeffs_legendre(ord+1)
    if verbose: #should be zero
        print("fractional difference between first/last coefficients is "+repr(coeffs[0]/coeffs[-1]-1))

    npt=np.int((xmax-xmin)/dx_targ)+1
    nn=(npt-1)%(ord)
    if nn>0:
        npt=npt+(ord-nn)
    assert(npt%(ord)==1)

    x=np.linspace(xmin,xmax,npt)
    dx=np.median(np.diff(x))
    dat=fun(x)

    #we could have a loop here, but note that we can also reshape our data, then som along columns, and only then
    #apply coefficients.  Some care is required with the first and last points because they only show up once.
    mat=np.reshape(dat[:-1],[(npt-1)/(ord),ord]).copy()
    mat[0,0]=mat[0,0]+dat[-1] #as a hack, we can add the last point to the first
    mat[1:,0]=2*mat[1:,0] #double everythin in the first column, since each element appears as the last element in the previous row

    vec=np.sum(mat,axis=0)
    tot=np.sum(vec*coeffs[:-1])*dx
    return tot
```

Code to Call it +Output

```
if True:
    print("Integrating sin")
    fun=np.sin
    xmin=0
    xmax=np.pi
    targ=2.0
    dx_targ=0.1
else:
    print("Integrating Lorentzian")
    fun=lorentz
    xmin=-5
    xmax=5
    targ=np.arctan(xmax)-np.arctan(xmin)
    dx_targ=0.5


for ord in range(2,16,2):
    val=integrate(fun,xmin,xmax,dx_targ,ord)
    print('For order ' + repr(ord) + ' error is ' + repr(np.abs(val-targ)))
```

```
def lorentz(x):
    return 1.0/(1.0+x**2)
```

```
Integrating sin
For order 2 error is 1.0333694131503535e-06
For order 4 error is 3.809155213474469e-09
For order 6 error is 7.276845792603126e-12
For order 8 error is 1.0769163388864018e-13
For order 10 error is 0.0
For order 12 error is 1.9984014443252818e-15
For order 14 error is 2.6645352591003757e-15
```

```
Integrating Lorentzian
For order 2 error is 0.003893516371427985 2
For order 4 error is 0.010977677697237 01
For order 6 error is 0.002621273236311783
For order 8 error is 0.01837703807845159
For order 10 error is 0.005032084054994446
For order 12 error is 0.001118349714313016
For order 14 error is 0.0003964865376655524
```

What Happened?

- In all cases, we used roughly the same number of function evaluations.

- Sin is a smooth function. Its error rapidly went to zero as we increased the order. If you know you're integrating a sin, crank away!

- Lorentzian is not a smooth function since it has poles at +/-i. Its power series expanded at zero does not converge for |x| >1. Integral is less accurate, and does not improve very rapidly as we increase order.

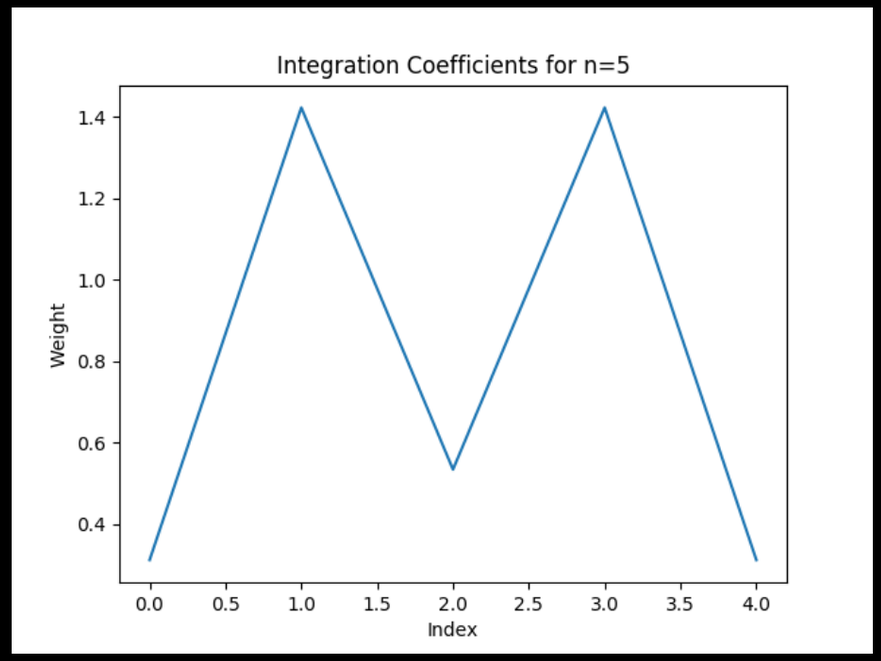
- Lorentzian sometimes gets worse at higher order!

Error Estimate

- Usual scheme is to pick an order, then vary step size until accuracy is good.
- How do I know accuracy? If I'm in happy Taylor regime, errors predictable. Compare f(4dx) & f(2dx) against f(2dx) & f(dx). Did differences shrink as expected? If not, try smaller step size.
- If you beat hard enough, eventually Taylor wins out.

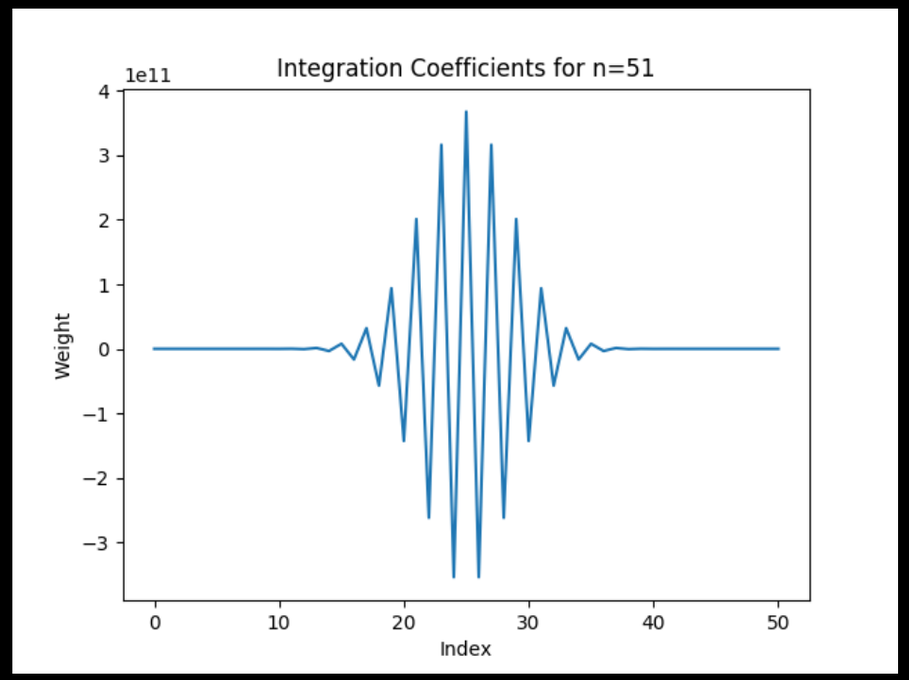
Interpolation Coefficients to High Order

- Top: 5th order polynomial integration weights.



- Bottom: 51st order polynomial integration weights.

- Do you want to go to (very) high order this way?



Romberg Integration

- Another way to get to high order.
- If I integrate from -*a* to *a*, only even terms survive in integral.
- If I have *n* estimates of area with varying d*x*, I could kill off n terms in *even* error series, giving accuracy of dx^{2n}.
- More stable than high order polynomial weights.

Scipy Romberg

- In scipy.integrate have 2 options:
  scipy.integrate.romb = integral from pre-evaluated points
  scipy.integrate.romberg = integral from function

```
for k=1 and 3 function calls, error is 0.011651369255893052
for k=2 and 5 function calls, error is 6.851628176995916e-05
for k=3 and 9 function calls, error is 1.0674648986963575e-07
for k=4 and 17 function calls, error is 4.2089887131169235e-11
for k=5 and 33 function calls, error is 4.440892098500626e-15
for k=6 and 65 function calls, error is 8.881784197001252e-16
for k=7 and 129 function calls, error is 4.440892098500626e-16
for k=8 and 257 function calls, error is 0.0
for k=9 and 513 function calls, error is 0.0
Romberg integration of <function vfunc at 0x11b243140> from [-1, 1]

 Steps  StepSize   Results
     1  2.000000  3.086161
     2  1.000000  2.543081  2.362054
     4  0.500000  2.399166  2.351195  2.350471
     8  0.250000  2.362631  2.350453  2.350404  2.350402
    16  0.125000  2.353462  2.350406  2.350402  2.350402  2.350402
    32  0.062500  2.351167  2.350403  2.350402  2.350402  2.350402  2.350402

The final result is 2.350402387287607 after 33 function evaluations.
```

```python
import numpy as np
from scipy import integrate

a=-1
b=1
for k in range(1,10):
    n=1+2**k
    dx=(b-a)/(n-1.0)
    x=np.linspace(a,b,n)
    y=np.exp(x)
    pred=np.exp(b)-np.exp(a)
    f=dx*integrate.romb(y)
    print('for k=' + repr(k) + ' and ' + repr(n) + ' function calls, error is ' + repr(np.abs(f-pred)))
f=integrate.romberg(np.exp,a,b,show=True)
```

Indefinite Integrals

- Handy trick: $\int_a^b f(x)dx=\int_{1/b}^{1/a} f(1/t)t^{-2}dt$ for $t=1/x$
- Can now set e.g. b to ∞, and take integral happily since 1/b=0.
- Happily, as long as function falls off quickly enough.

scipy quad

- Quad is the general purpose routine for integrating.
- Supports indefinite integrals, integrals against integrable singularities:

```
>>> ans=integrate.quad(np.exp,-np.inf,-1)
>>> print([ans[0]-np.exp(-1),ans[1]])
[-5.551115123125783e-17, 2.1493749551987453e-11]
>>>
```

```
>>> def fun(x):
...     return 1.0/np.sqrt(x)
...
>>> integrate.quad(fun,0.0,2)
(2.8284271247461907, 3.1401849173675 51e-15)
>>> print np.sqrt(8)
2.8284271247461903
>>>
```

Variable Step Size

- For Lorentzian, areas well away from poles should integrate nicely. Only around |*x*|<~1 is problematic.

- If I keep track, I will be able to see that away from the origin I converge, but less well at origin.

- I can find regions that behave, and not shrink dx when their errors are small.

- Regions that do not behave: shrink dx by a factor of 2, and try again.

- Life experience: Bad functions are usually bad in a small piece.

- Variable step size integration can easily save factors of ~hundred.

Side Note: Recursion

- A recursive function calls itself.
- In this case, we'll evaluate function across interval. If error small enough, we're done.
- Otherwise integral is integral of left half + integral of right half. Just call ourselves twice.
- If you don't have good stopping point, recursion can run away on you, easily crash computer.
- Good practice to think how stopping might go wrong.

Let’s play with our integrator

- Throw out some functions where you know the analytic integral. How do we do?

- If we shrink the input tolerance, does our error get more accurate?

- What’s a (finite, integrable function) with a spike? Does our integrator do lots of work around the spike and little elsewhere?

Cautionary Tale

- Let's integrate $f(x)=1+\exp(-0.5*(x/0.1)^2)$ from a (<<0) to b (>>0).
- What should the answer be?
- What do we get from (-20,20)?
- Does using scipy's quad help us here?
- How can we fix things?

Gaussian Quadrature

- We did well with high order and orthogonal polynomials. We might be able to do even better.

- Problem: polynomials not *quite* orthogonal on evenly spaced points.

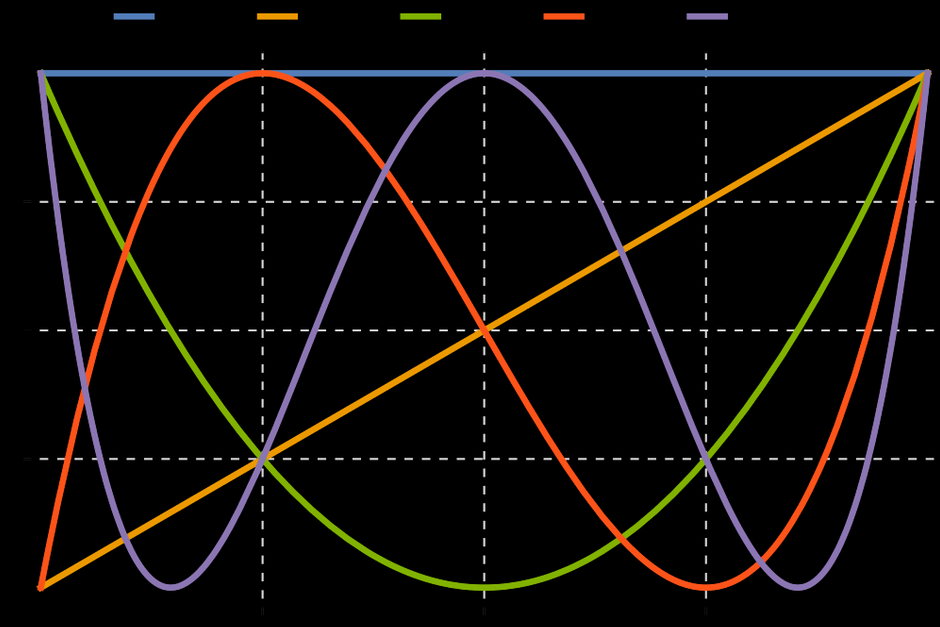
- Gaussian quadrature: if we can pick x positions (instead of evenly spaced), we can make points orthogonal to odd polynomials. By only fitting even, can go to twice the order. Weights depend on positions.

- Unexpected bonus - this works well for integrating w(x)f(x) for fixed w. One way to integrate over singularities.

- Example: integrate f(x)/sqrt(x) - we calculate quadrature positions, weights for w=1/sqrt(x), then use that to integrate f(x).

- Many weight function have already been generated - if you need this, have a look.

Chebyshev Polynomials



- $T_n=\cos(n\arccos(x))$, $-1\le x\le 1$
- $T_0=1$, $T_1=x$.
- Recurrence relation: $T_{n+1}=2xT_n-T_{n-1}$.
- Bounded by +/-1
- Orthogonal under weight: $\int T_n T_m/(1-x^2)^{1/2}dx=0$ $(i\neq j)$, π $(i=j=0)$ or $\pi/2$ $(i=j>0)$.
- Make a natural way of doing Gaussian quadrature (Gauss-Chebyshev quadrature) of $f(x)/(1-x^2)^{1/2}$.

Chebyshev Series

- Let's say we want to make a polynomial expansion for some function with the smallest *maximum* errors.

- Common case when, say, trying to write code for evaluating functions.

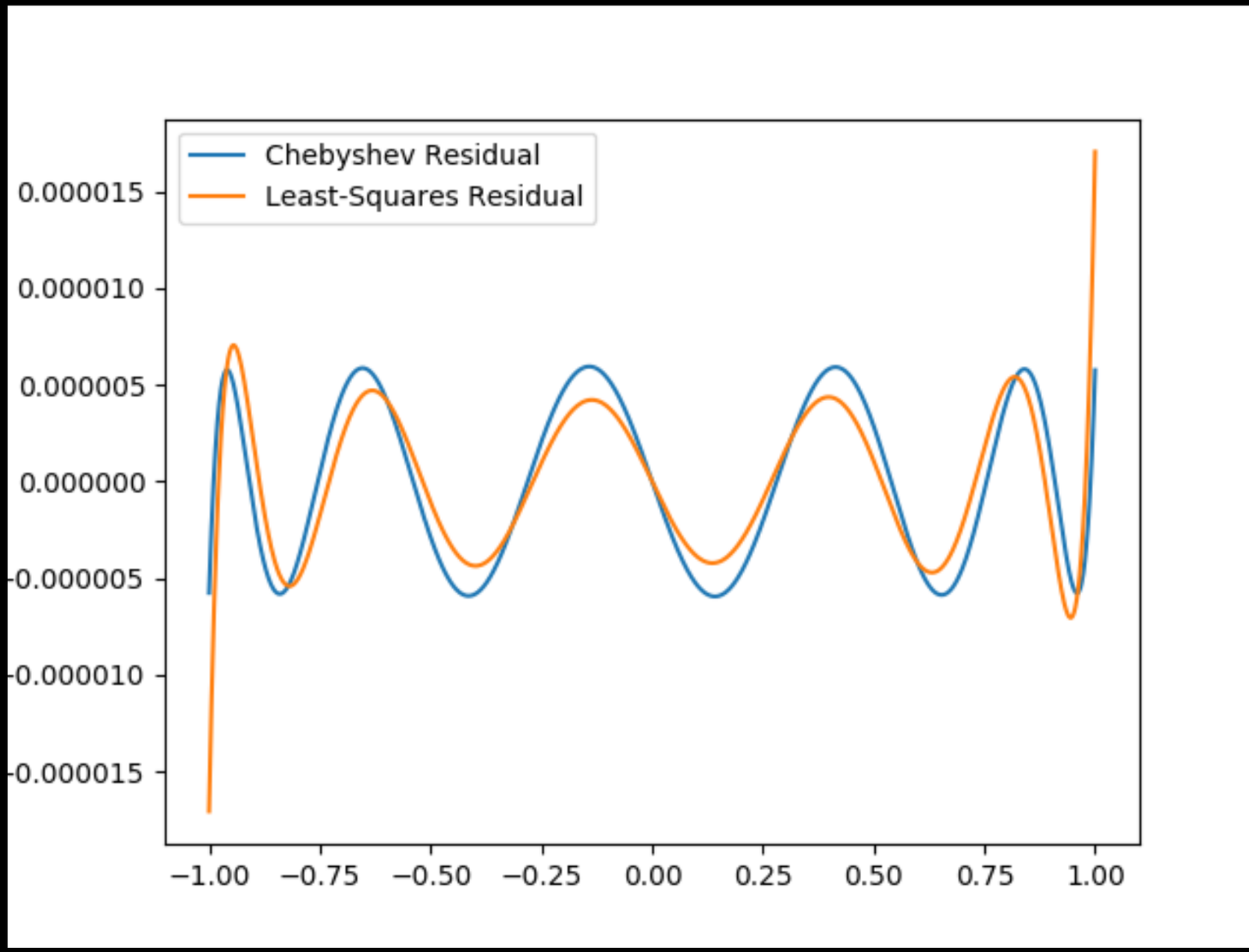
- For smooth functions, Chebyshev coefficients tend to drop smoothly.

- Because T_n are bounded, max error is $\sum$ of cut coefficients.

- If you want to have fast functions at possibly relaxed precision over possibly restricted range, T_n are very useful.

How This Looks

- Look at cheb_expand.py
- Fits chebyshev and least-squares to same order.




```
[Jonathans-MacBook-Pro:lecture_3 sievers$ python3 cheb_expand.py
 rms error for least-squares is  3.6653313842173857e-06  with max error  1.7044661309315215e-05
 rms error for chebyshev     is  4.155908892551404e-06  with max error  5.9427933866151 86e-06
```