

# Phys 512 Lecture 3

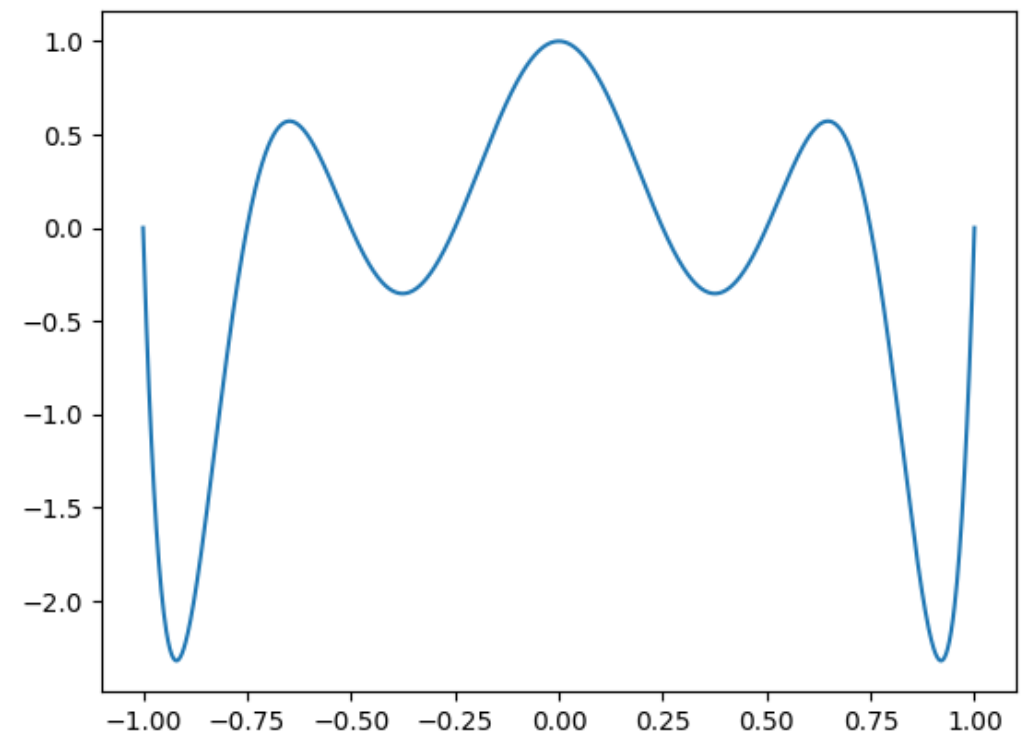
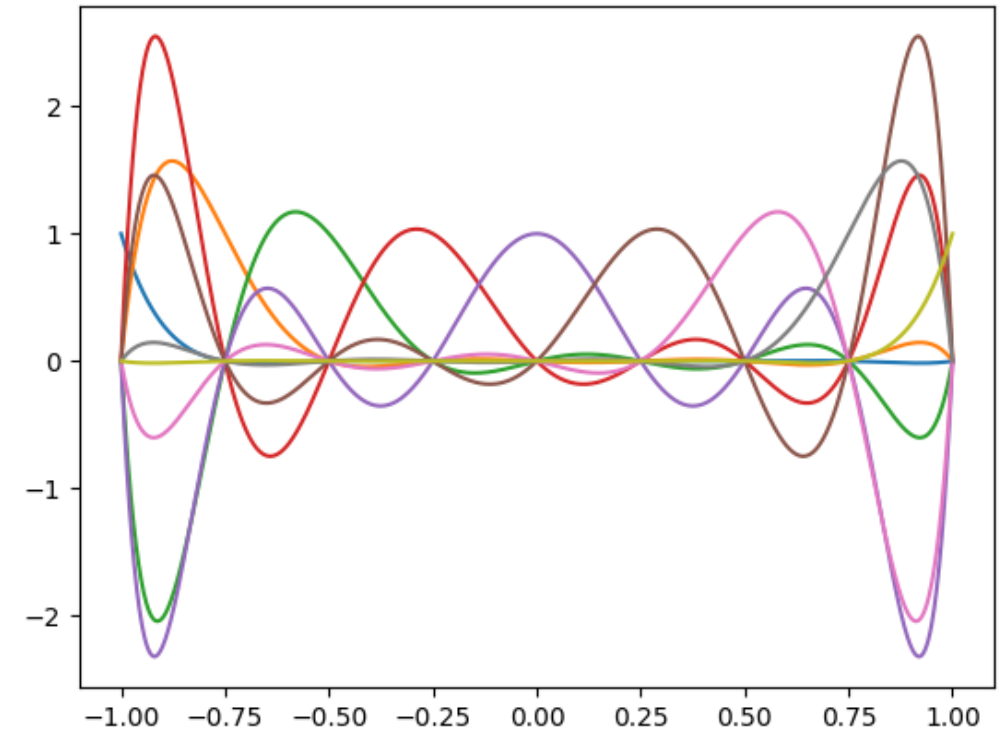
## Integration

# Reminder - poly interpolation

```
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  $(ord \cdot dx)^n f^{(n)}/n!$  is converging.



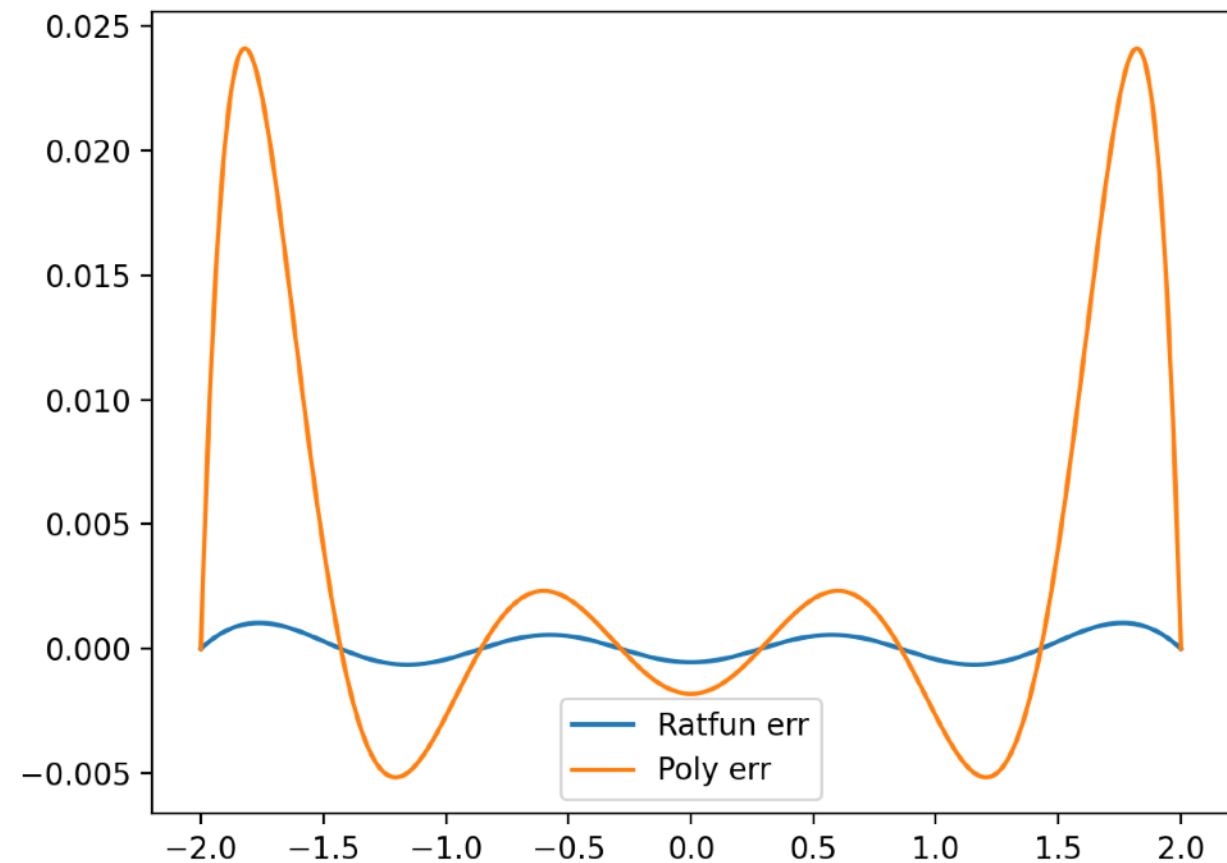
```
def rat_eval(p,q,x):
    top=0
    for i in range(len(p)):
        top=top+p[i]*x**i
    bot=1
    for i in range(len(q)):
        bot=bot+q[i]*x**(i+1)
    return top/bot

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
```

```
#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')
```

```
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)
```



# Integration

- Frequently need to numerically integrate functions
- 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

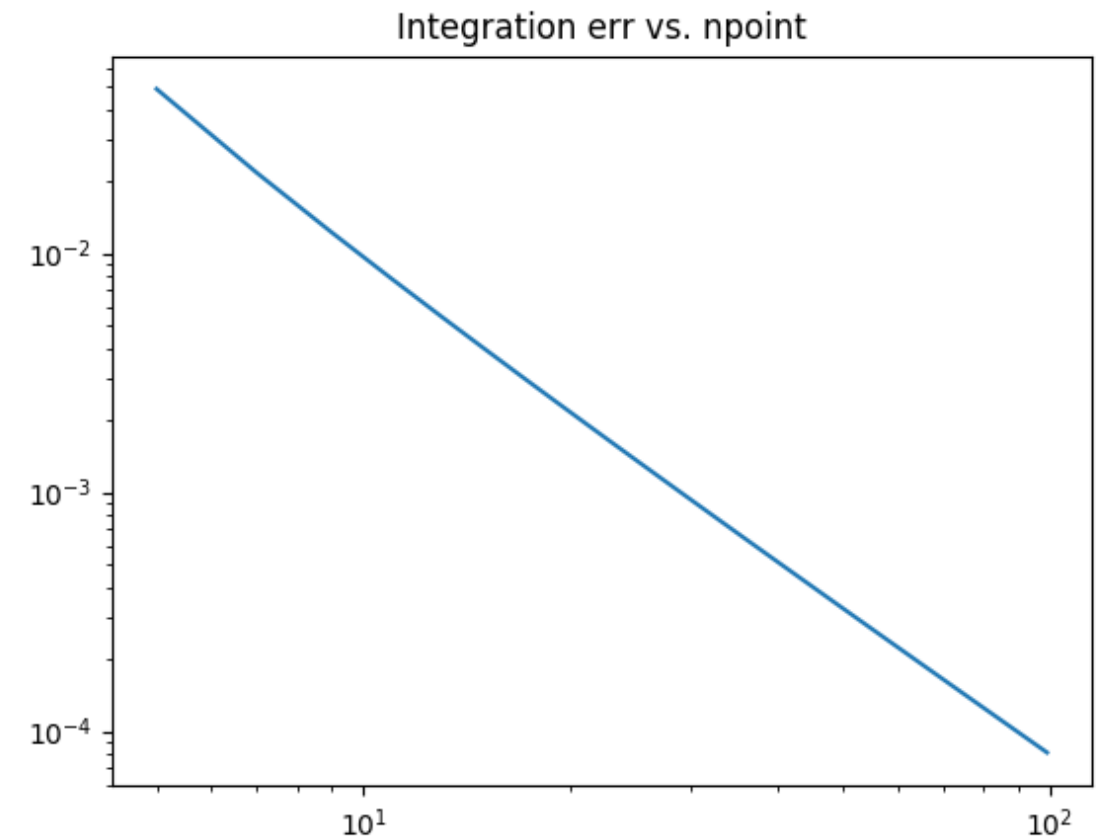
- Break function into series of regions, draw a line between points.
- What is the average value of  $y$  if it is based on a line between  $y_i$  and  $y_{i+1}$ ? just  $0.5(y_i + y_{i+1})$ .
- If points are  $dx$  apart, then  $\text{area}/dx = 0.5(y_0 + y_1) + 0.5(y_1 + y_2) \dots + 0.5(y_{n-2} + y_{n-1}) = 0.5(y_0 + y_{n-1}) + \sum y_i, i=1..n-2$ .
- 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?
- $f(dx) = f_{\text{true}} + a dx^2 + \dots$     $f(2dx) = f_{\text{true}} + a(2dx)^2 + \dots$
- $4f(dx) - f(2dx) = 4f_{\text{true}} - f_{\text{true}} + \dots = 3f_{\text{true}} + \dots$  or  $f_{\text{true}} = (4f(dx) - f(2dx))/3$
- 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.
- Note - sum of coeffs should be length of region. In this case, we have  $2 dx$ , so sum of coeffs must be 2 (otherwise a constant function would be wrong)

# Equivalent

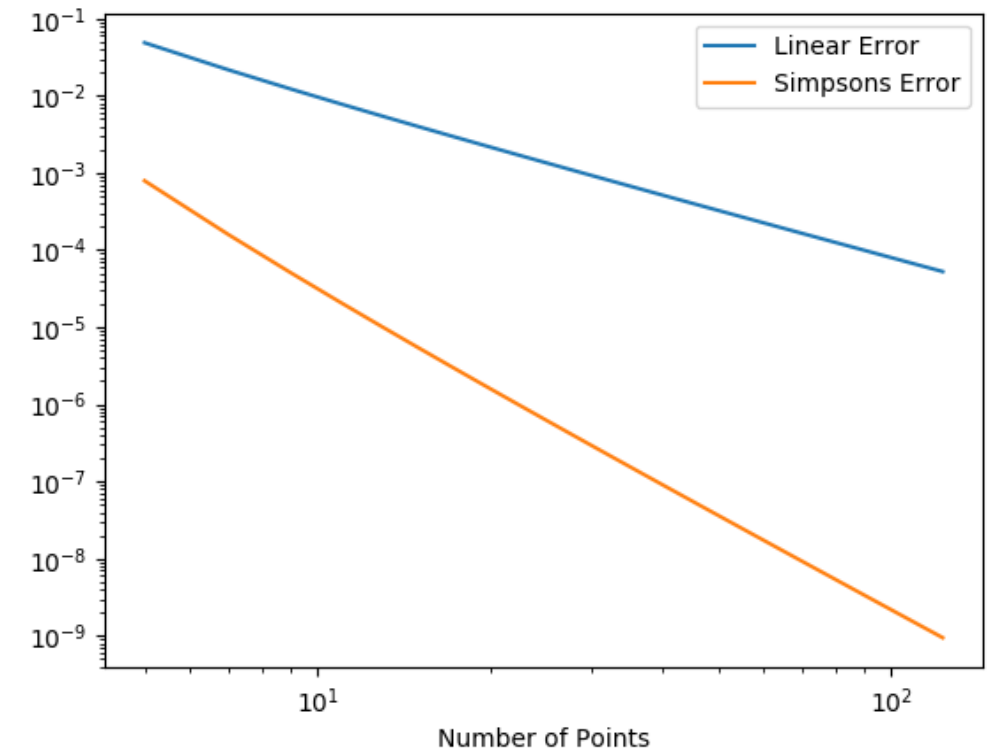
- We could also fit a parabola to 3 points. For simplicity take  $x=-1,0,1$ . There's a trapezoid going through  $y_0$  and  $y_2$ , with a parabola that goes through  $y_1$  minus  $(y_0+y_2)/2$ .
- Parabola is  $(y_1-(y_0+y_2)/2)(1-x^2)$ . Integral is  $(x-x^3/3)$ , evaluates to  $4/3$  over  $2dx$ , 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  $x^3$ ?



```

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')

```

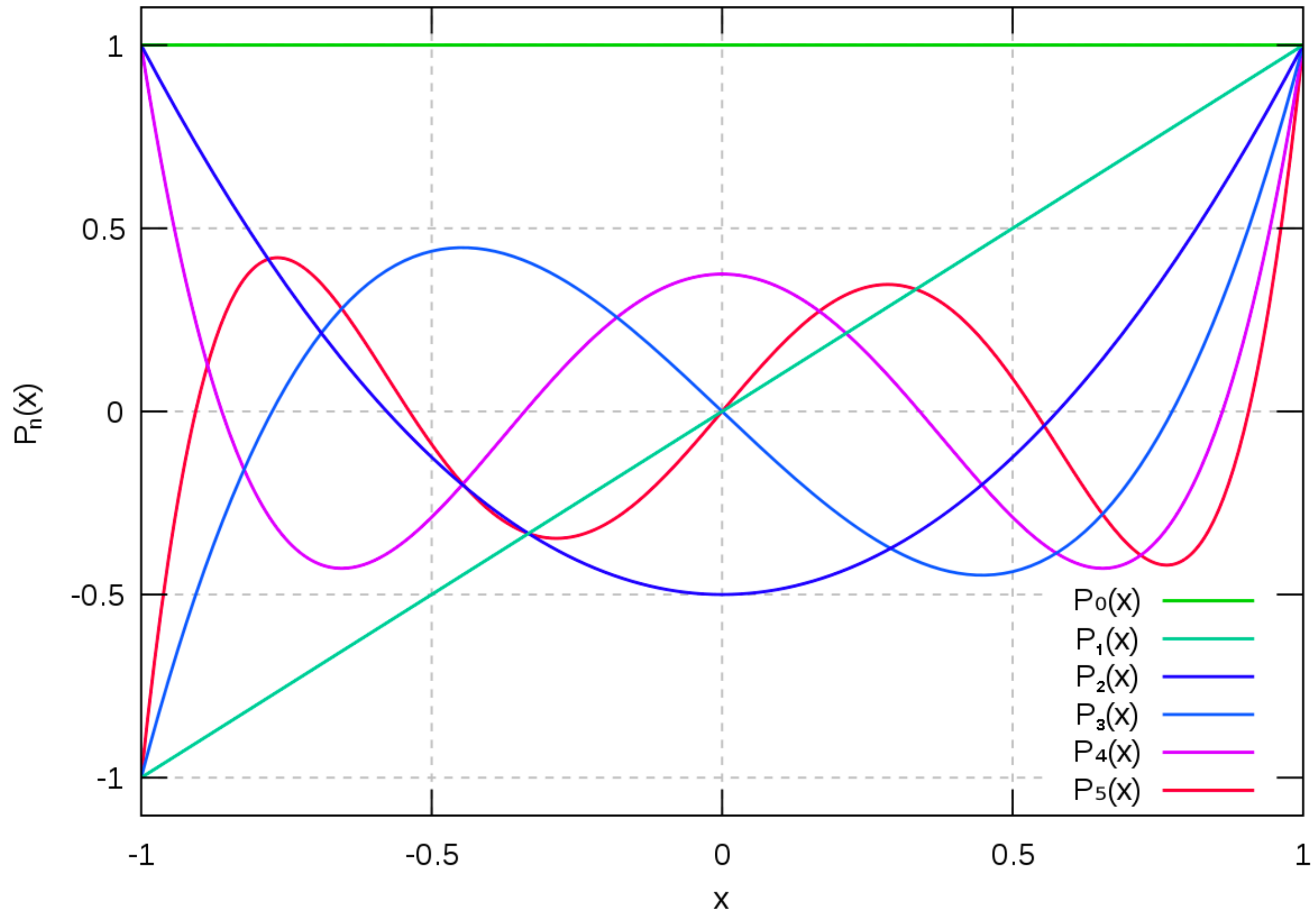


- Simpsons is far more accurate than trapezoid.
- Error goes like  $dx^4$ , since  $x^3$  term integrates to zero.
- I didn't have to make any more function calls, just used the ones I had more cleverly.
- NB - would this work with an even number of  $y_i$ ?

# Can We Turn it up to 11?

- Yes! Regular polynomials ( $x, x^2, x^3 \dots$ ) 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.  $(n+1)P_{n+1} = (2n+1)xP_n - nP_{n-1}$ .  $P_0 = 1$ ,  $P_1 = x$ .
- NB - be careful about definition of 'dx'. In analysis, natural to think of it as width of high-order region, but in practice we think of it as spacing between points. They are not the same.

legendre polynomials



# 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 1.0769163338864018e-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 0.0038935163714279852
For order 4 error is 0.01097767769723701
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?

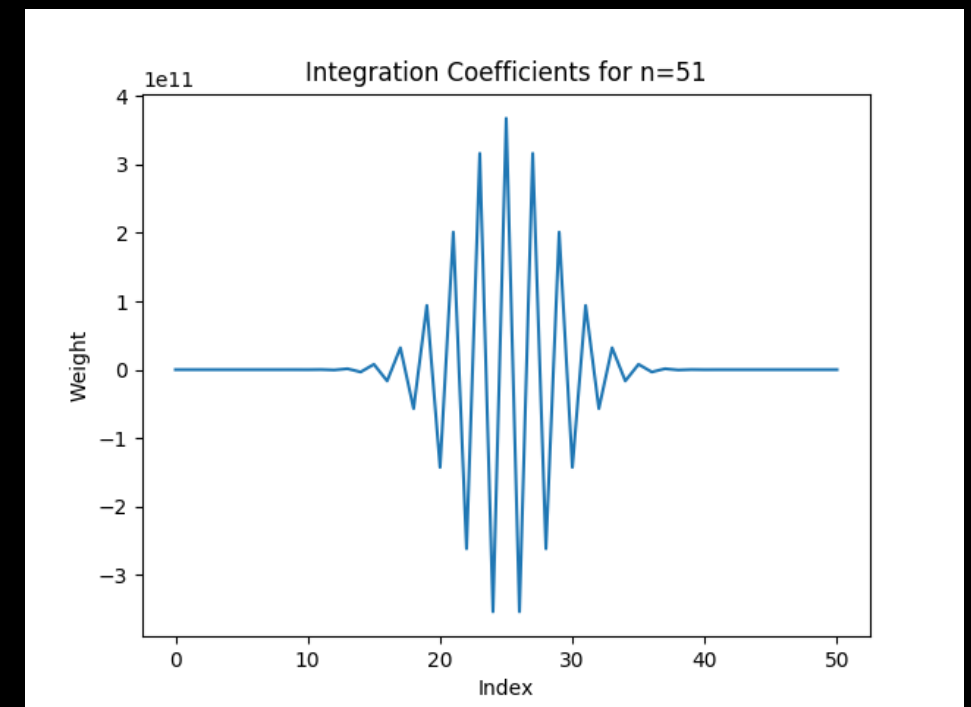
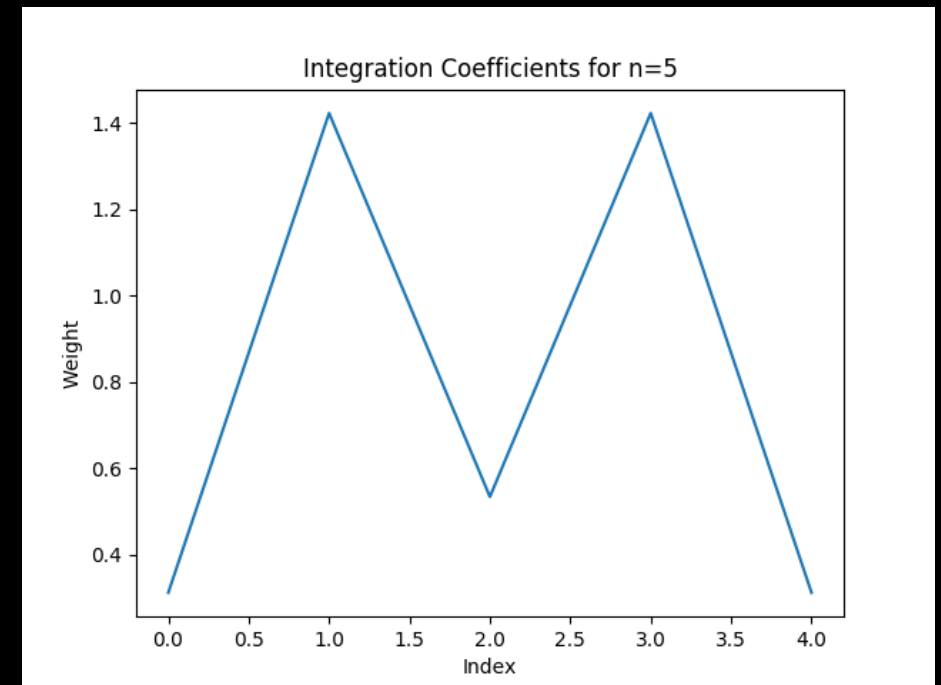
- In all cases, we used roughly the same number of function evaluations.
- Sin is a smooth function. Its error rapidly went to zero as we increased the order. If you know you're integrating a sin, crank away!
- Lorentzian is not a smooth function since it has poles at  $\pm i$ . Its power series expanded 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: 5<sup>th</sup> order polynomial integration weights.
- Bottom: 51<sup>st</sup> 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  $dx$ , 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]
```

```
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)
```

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.

# 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  $\infty$ , 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, 3.140184917367551e-15)
[>>> print np.sqrt(8)
2.8284271247461903
>>>
```

# Variable Step Size

- For Lorentzian, areas well away from poles should integrate nicely. Only around  $|x| < \sim 1$  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  $\sim$ hundred.
- Let's write one.



# 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

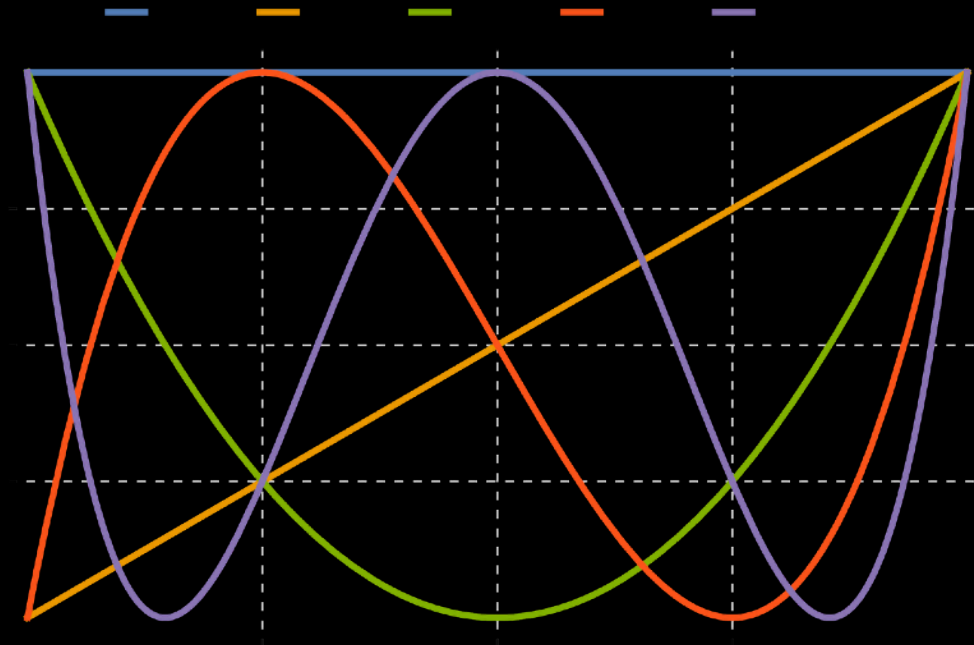
- Let's integrate  $f(x)=1+\exp(-0.5*(x/0.1)^2)$  from  $a$  ( $\ll 0$ ) to  $b$  ( $\gg 0$ ).
- What should the answer be?
- What do we get from  $(-20,20)$ ?
- Does using scipy's quad help us here?
- How can we fix things?

# 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

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

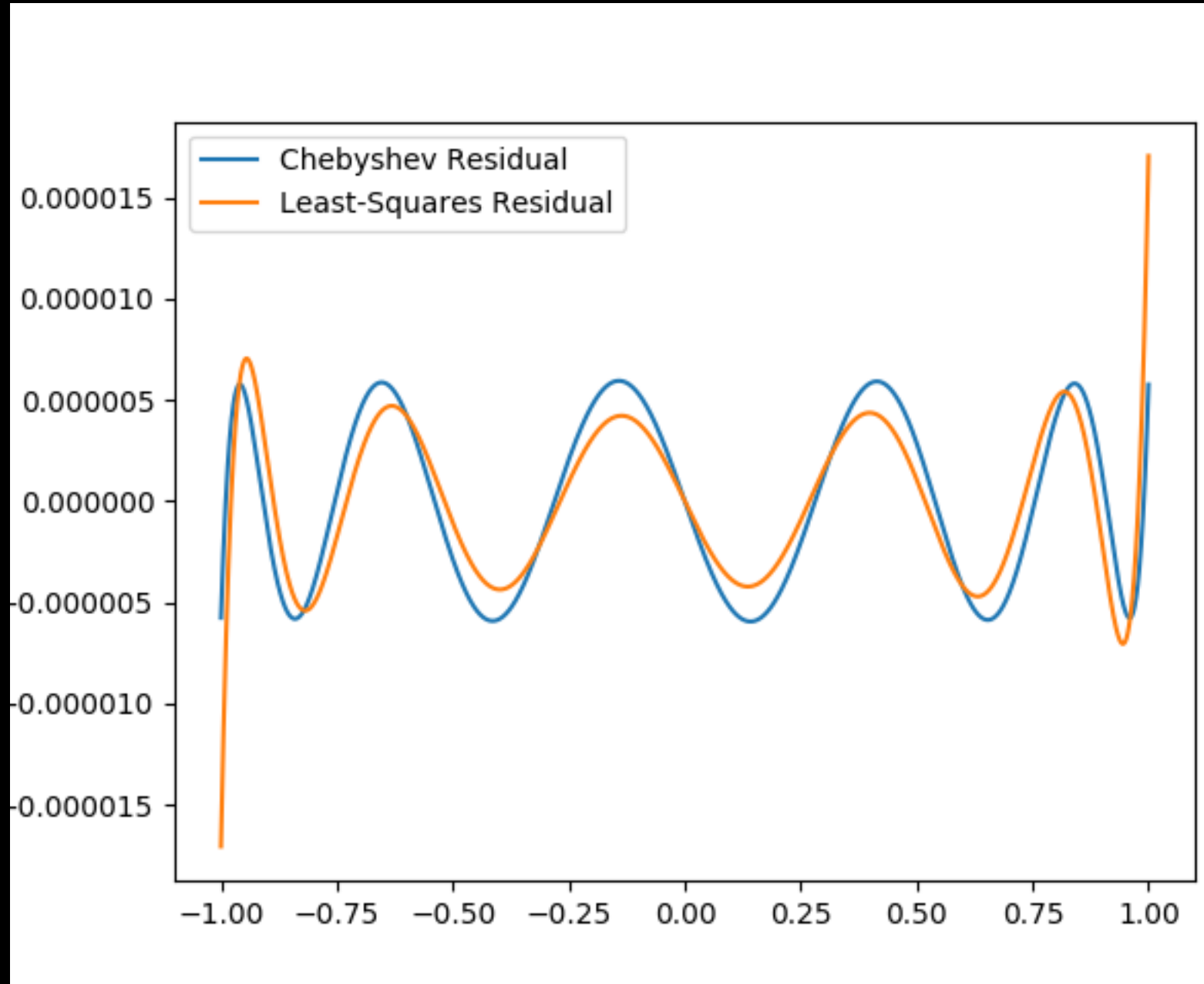


# Chebyshev Series

- 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 5.942793386615186e-06
```