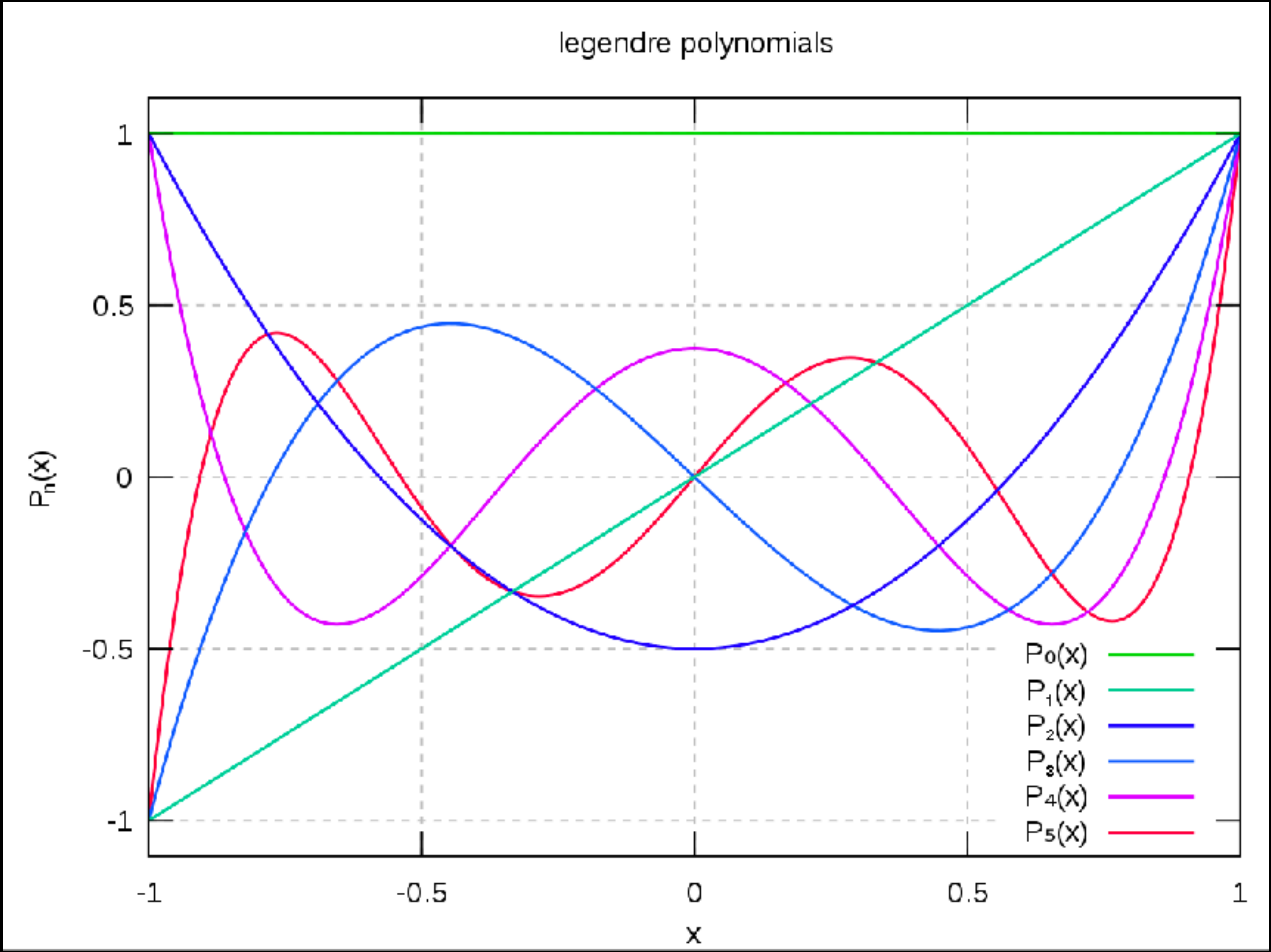
Phys 512 Lecture 3
Polynomial Integration ctd.

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 1.0769163338864018e-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 0.010977677697237 01
For order 6 error is 0.0026212732363117 83
For order 8 error is 0.018377038078451 59
For order 10 error is 0.005032084054994446
For order 12 error is 0.001118349714313016
For order 14 error is 0.00039648653766 55524
```

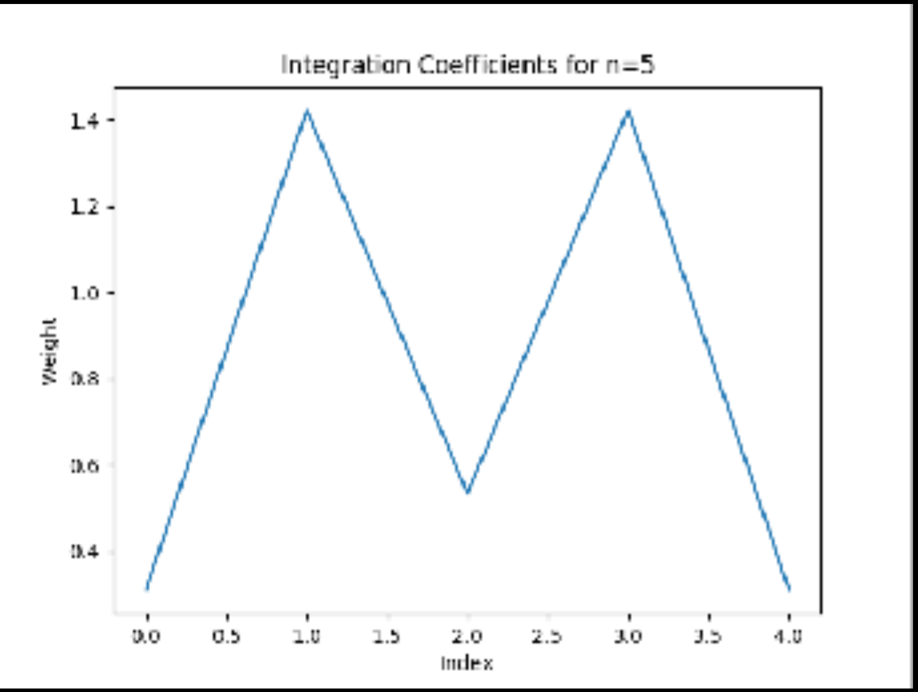
What Happened?

- 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!

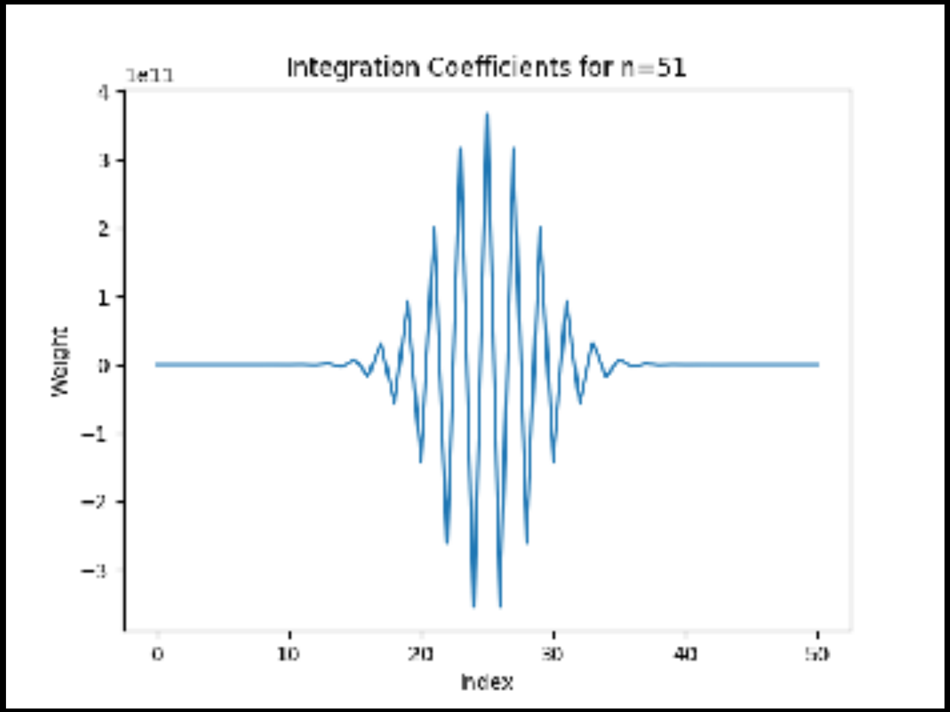
Error Estimate

- 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.
- Alternatively, compare f(2x) against f(x). Difference is another error estimate.
- If you beat hard enough, eventually Taylor wins out (usually).

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, 3.140184917367551e-15)
[>>> print np.sqrt(8)
 2.828427124746190З
 >>>
```

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.
- Variabel step size integration can easily save factors of ~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?

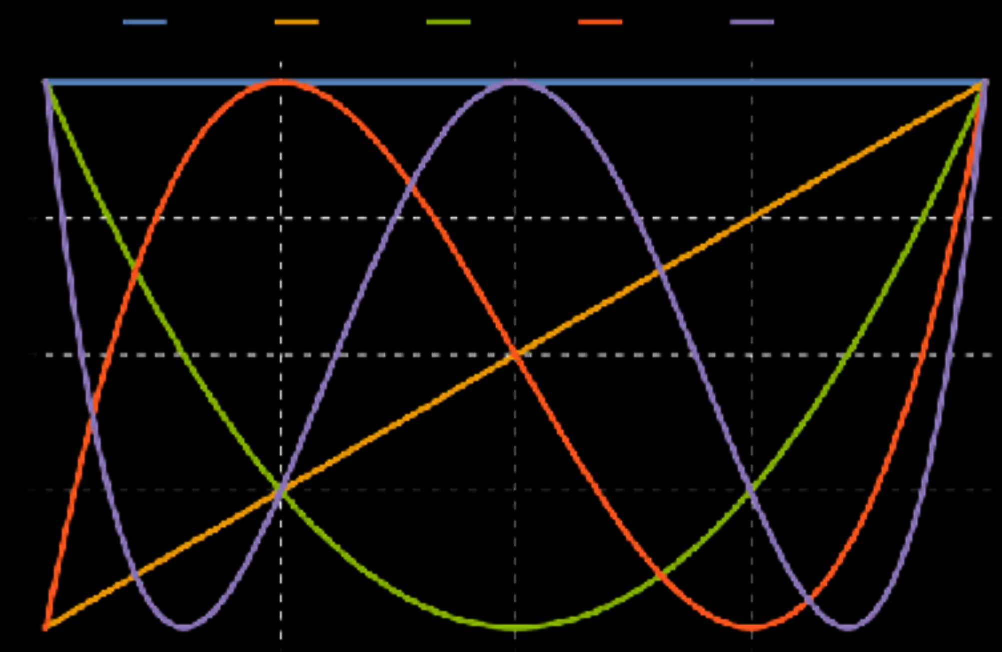
Cautionary Tale

- Let’s integrate $f(x)=1+exp(-0.5*(x/0.1)^2)$ from a (<<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 were on the right track with high order and orthogonal polynomials.
- Problem: polynomials not *quite* orthogonal on evenly spaced points.
- Gaussian quadrature: if we can pick x positions (instead of evenly spaced), can also cancel higher order terms, keep polynomials orthogonal. 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



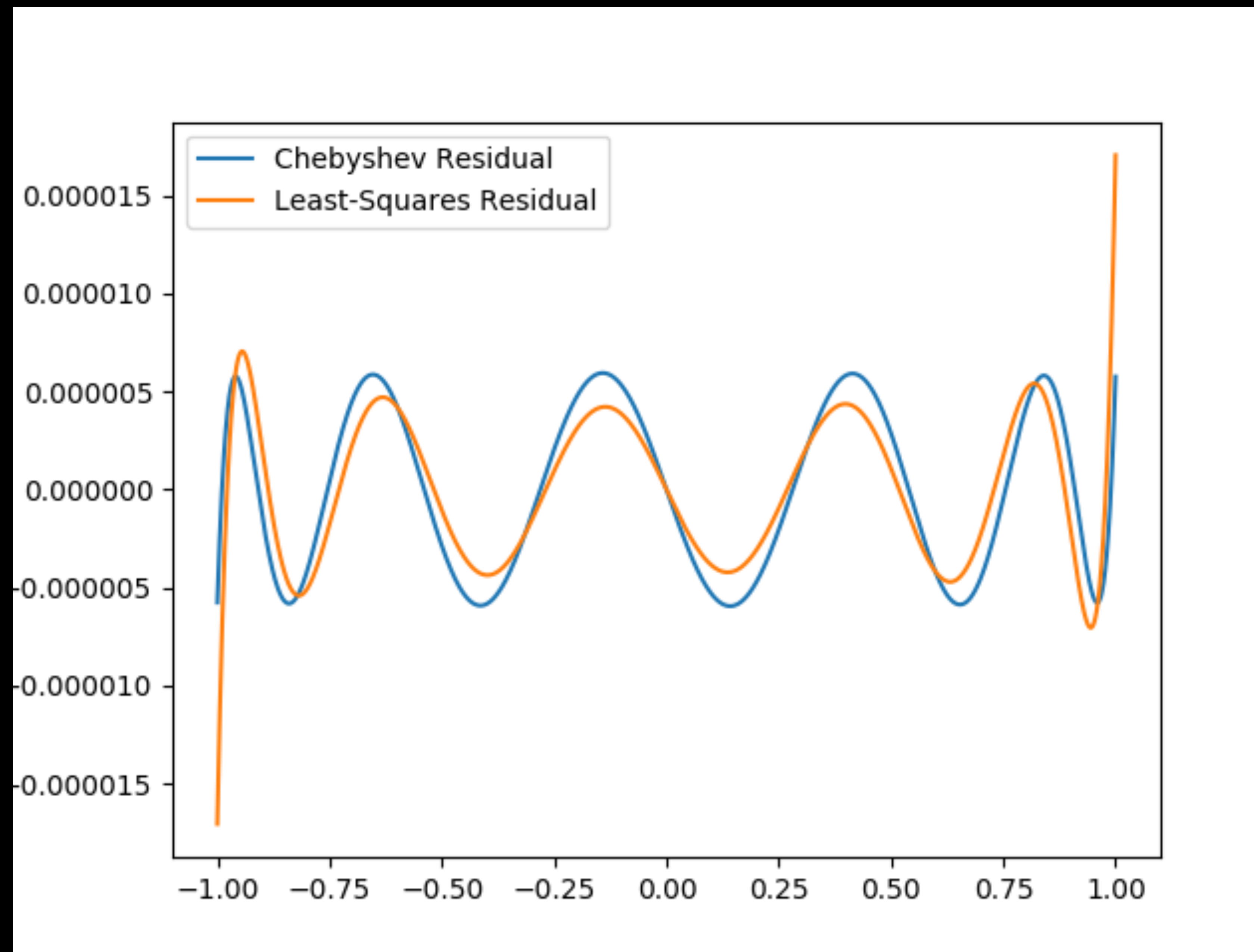
- $T_n = \cos(n \operatorname{acos}(x))$, $-1 \leq x \leq 1$
- $T_0 = 1$, $T_1 = x$.
- Recurrence relation: $T_{n+1} = 2xT_n - T_{n-1}$.
- Bounded by +/-1
- Orthogonal under weight: $\int T_n T_m / (1-x^2)^{1/2} dx = 0$ $(i \neq j)$, π $(i=j=0)$ or $\pi/2$ $(i=j>0)$.
- Make a natural way of doing Gaussian 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  3.665331384217385 7e-06  with max error  1.7044661309315215e-05
rms error for chebyshev     is  4.155908892551404e-06  with max error  5.942793386615186e-06
```