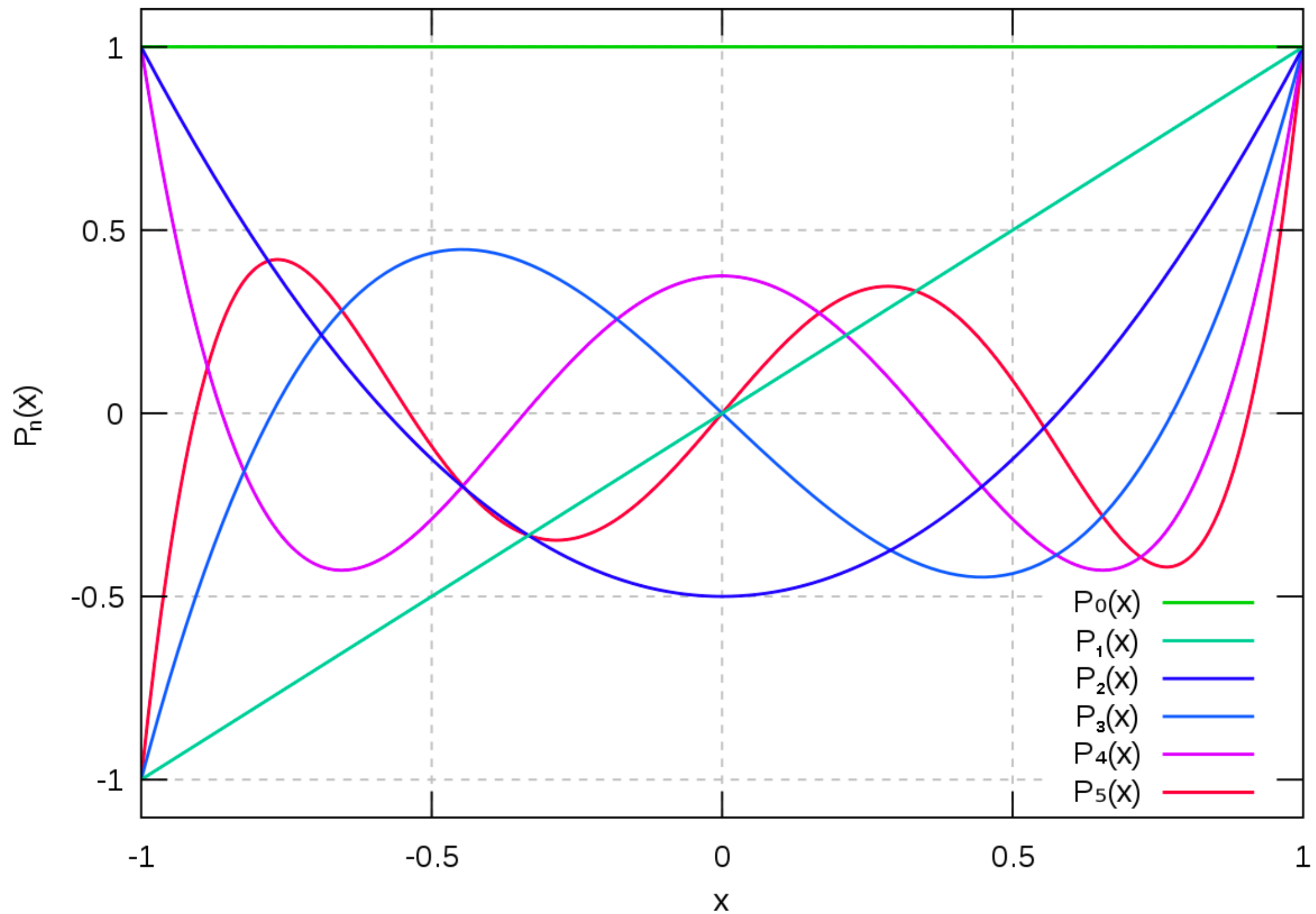


# Phys 512 Lecture 5

## Integration ctd./ODEs

legendre polynomials



# 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)))
```

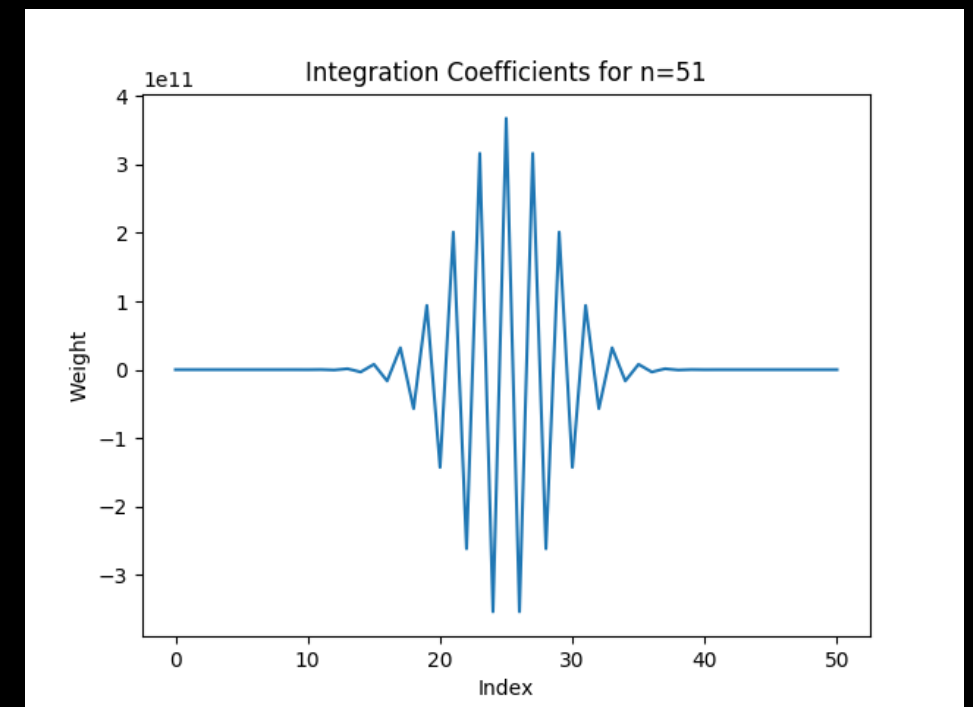
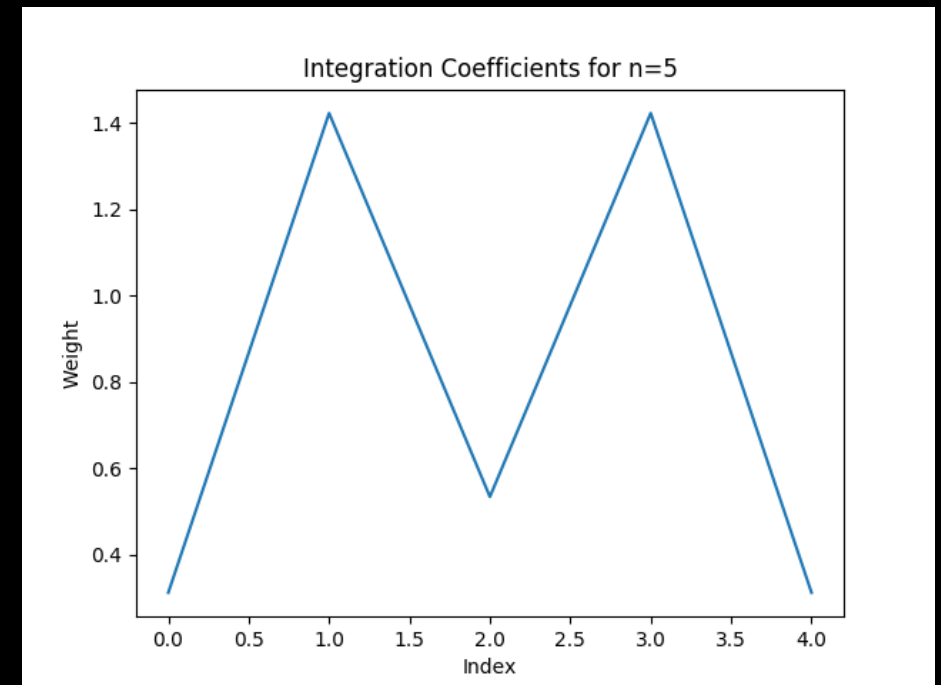
```
def lorentz(x):
    return 1.0/(1.0+x**2)
```

```
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
```

# 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.



# 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?

# 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
>>>
```



# 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  $(-20,20)$ ? How about  $(-25,15)$ ?
- Does using scipy's quad help us here?
- How can we fix things?

# Indefinite Integrals

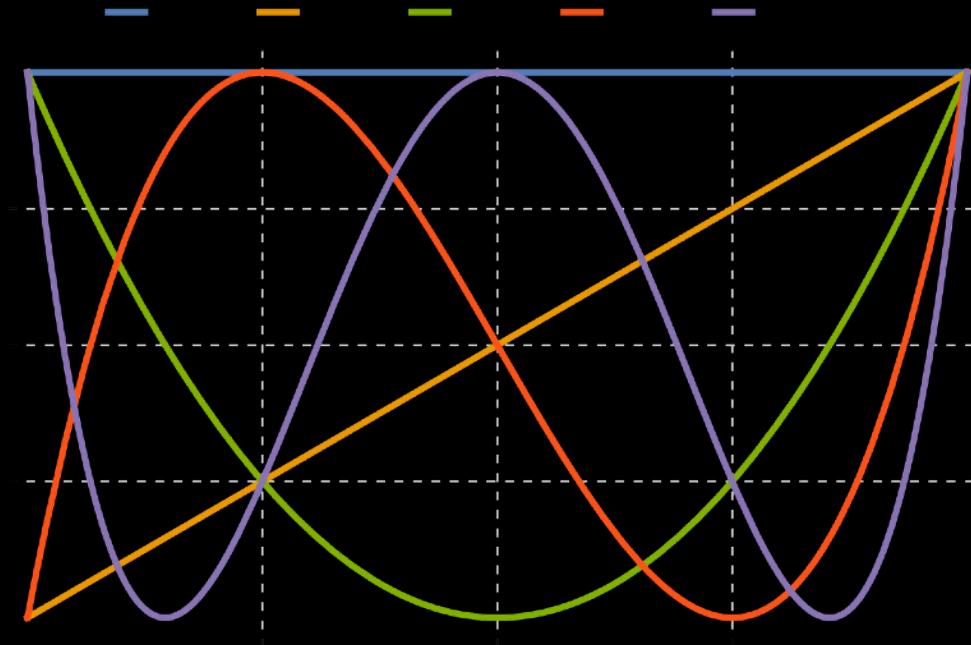
- Handy trick:  $\int_a^b f(x)dx = \int_{1/b}^{1/a} f(1/t)t^{-2}dt$  for  $t=1/x$
- Can now set e.g.  $b$  to  $\infty$ , and take integral happily since  $1/b=0$ .
- Happily, as long as function falls off quickly enough.

# 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  $f(x)/(1-x^2)^{1/2}$ .



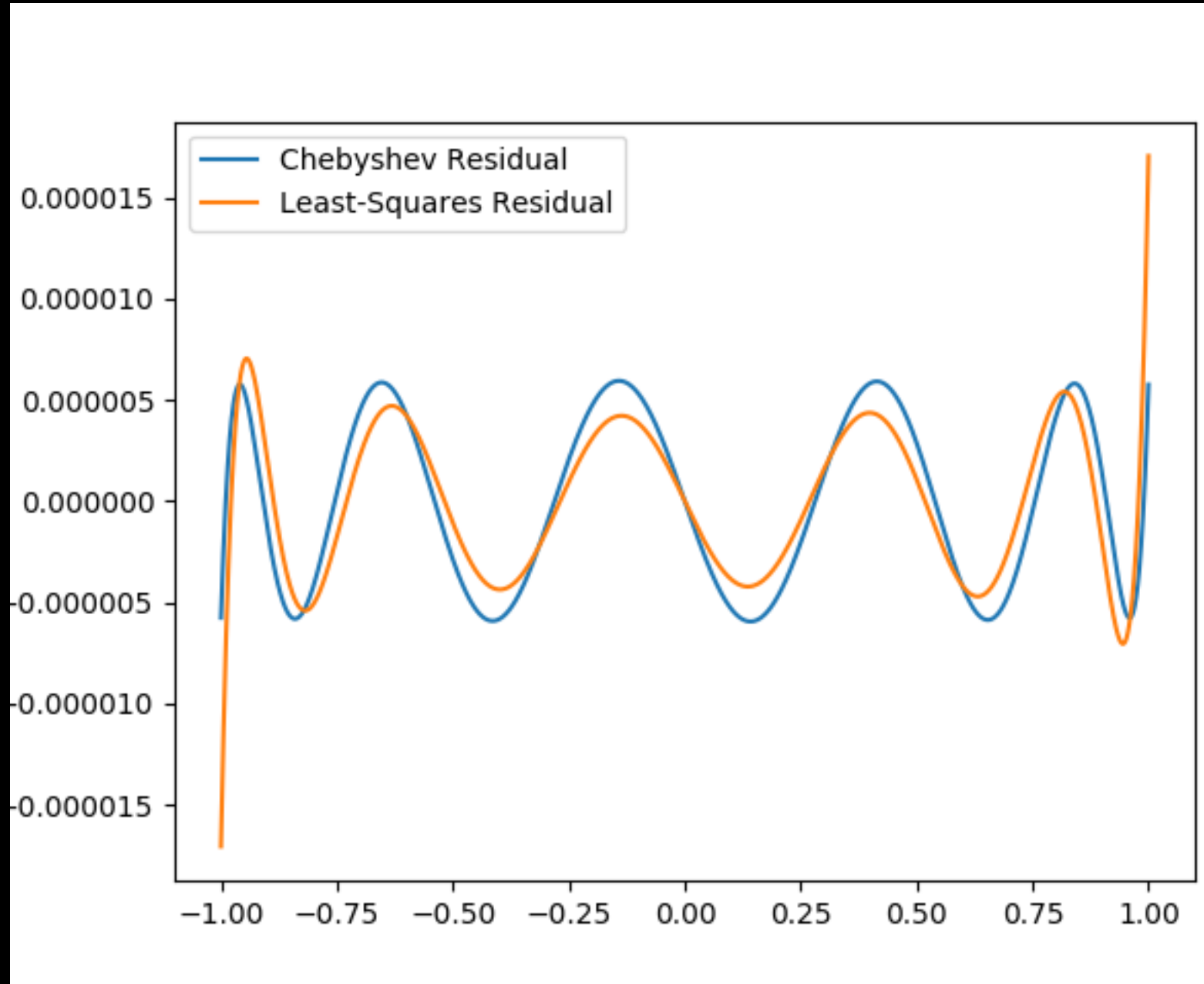


# Chebyshev Series

- 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
```

# ODEs

- usual case: we have  $y(x_0)$ , find  $y(x_0+h)$  given  $dy/dx=f(x,y)$
- Harder than integration, because we don't know how to evaluate  $dy/dx$  along the path - that depends on the (unknown) value of  $y$ .
- We can still apply many of the Taylor series tricks we've learned.
- NB - we could have a system of equations just as well as a single equation.
- NB 2 - we can also have higher order equations recast into a system. variables are then  $y, y', y'' \dots$

# 1st order

- We could just take  $y(x+h)=y(x)+h \, dy/dx = y(x)+hf$
- How will error scale with size of  $h$ ?
- This isn't very good...
- Can we do better?



# RK2

(see [https://web.mit.edu/10.001/Web/Course\\_Notes/Differential\\_Equations\\_Notes/node5.html](https://web.mit.edu/10.001/Web/Course_Notes/Differential_Equations_Notes/node5.html))

- I could take a trial step, then evaluate derivative. Maybe combine with first derivative to do better?
- Taylor expand  $y(x+h)$  to 2nd order.
- Take  $k_1=hf(x,y)$ ,  $k_2=hf(x+\alpha h, y+\beta k_1)=hf(x+\alpha h, y+\beta fh)$
- Let answer be  $y(x+h)=ak_1+bk_2$ .
- Solve for  $a, b, \alpha, \beta$  to make agree with Taylor.
- Answer underconstrained - have freedom to pick. Usual is  $\alpha=\beta=1$ ,  $a=b=1/2$ , or  $\alpha=\beta=1/2$  and  $a=0, b=1$ .

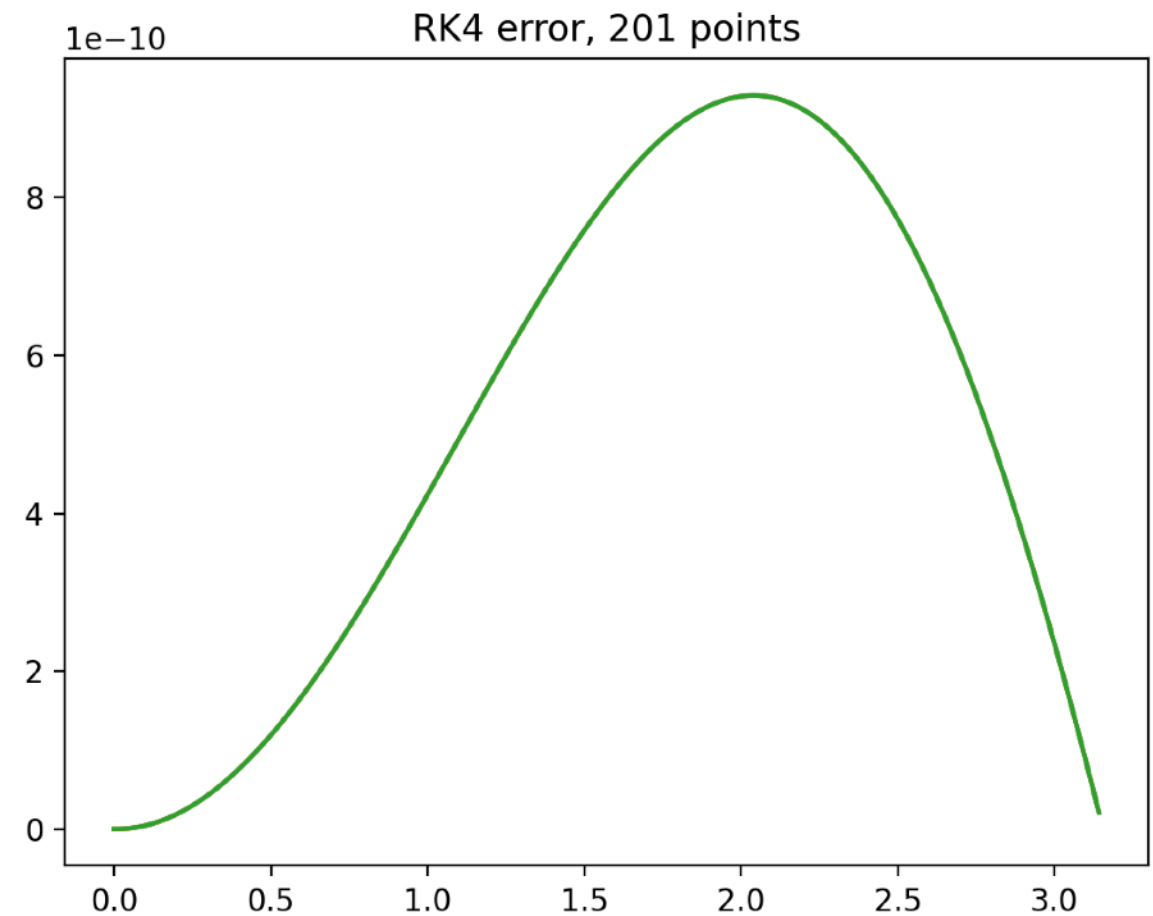
# RK4

- Can extend to 4<sup>th</sup> order. Mathy, but not very illustrative.
- $k_1 = hf(x, y)$   
 $k_2 = hf(x + h/2, y + k_1/2)$   
 $k_3 = hf(x + h/2, y + k_2/2)$   
 $k_4 = hf(x + h, y + k_3)$   
 $y(x + h) = y(x) + (k_1 + 2k_2 + 2k_3 + k_4)/6$
- Accurate to 4th order - ODE equivalent of Simpson's rule.
- Adaptive stepsize *highly* recommended.

```
def f(x,y):
    dydx=np.asarray([y[1],-y[0]])
    return dydx
```

```
def rk4(fun,x,y,h):
    k1=fun(x,y)*h
    k2=h*fun(x+h/2,y+k1/2)
    k3=h*fun(x+h/2,y+k2/2)
    k4=h*fun(x+h,y+k3)
    dy=(k1+2*k2+2*k3+k4)/6
    return y+dy
```

```
npt=201
x=np.linspace(0,np.pi,npt)
y=np.zeros([2,npt])
y[0,0]=1 #starting conditions
y[1,0]=0 #if I start at peak, then first derivative =0
for i in range(npt-1):
    h=x[i+1]-x[i]
    y[:,i+1]=rk4(f,x[i],y[:,i],h)
truth=np.cos(x)
print(np.std(truth-y[0,:]))
```



# Bulirsch-Stoer

- We saw we could take multiple not-very accurate integrals, and combine to get high accuracy by estimating error terms.
- Can do the same with ODEs - Romberg equivalent is called Bulirsch-Stoer.
- If you need high accuracy, have smooth function, try this!



# Stability

- let  $y' = -cy$ . Answer should be  $\exp(-cx)$
- Solve the stupid way:  $y(x+h) = y(x) + hf(x,y) = y(x) - hcy(x)$
- $y(x+h) = y(x)(1-hc)$ .  $y(x+nh) = y(x)(1-hc)^n$ .
- What happens if  $h$  is too large? For  $|1-hc| > 1$ , this *grows* exponentially. Large steps have made our answer *unstable*.

# Stiff Equations

- Very often in systems, one equation (or eigenmode) has a large  $c$ , other has a small  $c$ .
- Solution is large  $c$  converges very quickly, and stays at solution while we wait for small  $c$  to evolve.
- If we aren't careful and take steps for small  $c$ , then large  $c$  becomes unstable, and solution blows up.
- Shrinking time step enough to track large  $c$  may be impracticable.
- Such systems are called *stiff*. Solutions presented here are simplistic, but knowing you have a stiff set is most of the battle.

# U238 Decay

- Radioactive decay common situation.
- U238 takes 4 billion years to go to Th234. Po214 takes 160 microseconds to go to Pb210.
- To solve naively, takes (4 billion years/160 microseconds) =  $10^{21}$  timesteps.

|                  | Half-Life | Time unit        | Emitter |
|------------------|-----------|------------------|---------|
| Uranium-238      | 4,468     | billion of years | alpha   |
| Thorium-234      | 24,10     | days             | beta -  |
| Protactinium-234 | 6,70      | hours            | beta -  |
| Uranium-234      | 245 500   | years            | alpha   |
| Thorium-230      | 75380     | years            | alpha   |
| Radium-226       | 1 600     | years            | alpha   |
| Radon-222        | 3,8235    | days             | alpha   |
| Polonium-218     | 3,10      | minutes          | alpha   |
| Plomb-214        | 26,8      | minutes          | beta -  |
| Bismuth-214      | 19,9      | minutes          | beta -  |
| Polonium-214     | 164,3     | microseconds     | alpha   |
| Plomb-210        | 22,3      | years            | beta    |
| Bismuth-210      | 5,015     | years            | beta    |
| Polonium-210     | 138,376   | days             | alpha   |
| Plomb-206        | Stable    |                  |         |

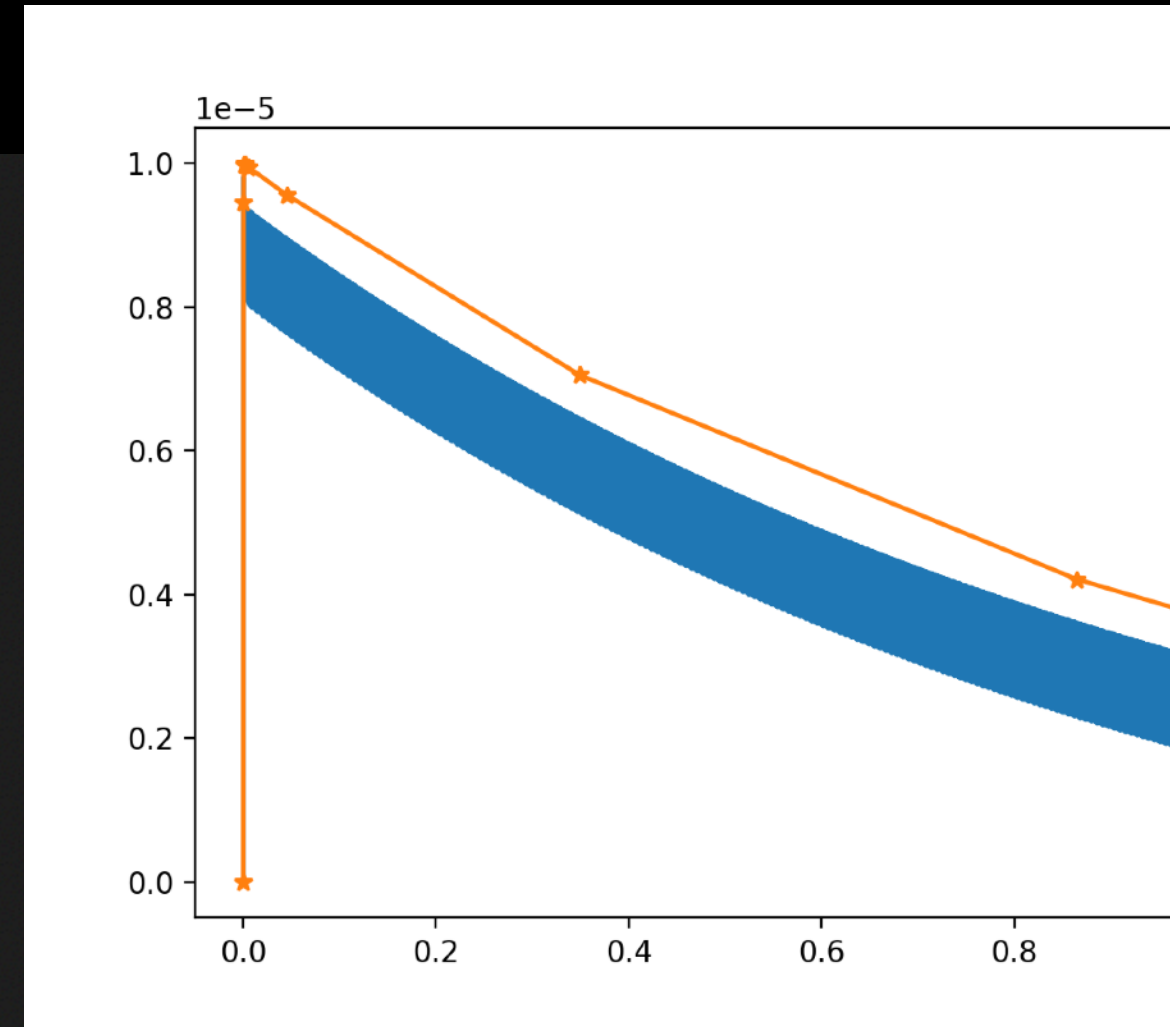
# Implicit

- Common solution - use derivative at end of interval.
- In constant coefficient case  $y_{n+1}=y_n-hcy_{n+1}$ .
- Solve:  $y_{n+1}(1+hc)=y_n$ ,  $y_{n+1}=y_n/(1+hc)$ . (reminder - old way was  $y_{n+1}=y_n(1-hc)$ . Agrees to 1<sup>st</sup> order but not 2<sup>nd</sup>. )
- I can now make  $h$  very large and remain stable. I may not be accurate, but I can crank up  $h$ .
- This is just an introduction, much better ways to handle stiff equations exist, but at least you know to look for them...

# Scipy ODE Driver

```
import numpy as np
from scipy import integrate
def fun(x,y,half_life=[1,1e-5]):
    #let's do a 2-state radioactive decay
    dydx=np.zeros(len(half_life)+1)
    dydx[0]=-y[0]/half_life[0]
    dydx[1]=y[0]/half_life[0]-y[1]/half_life[1]
    dydx[2]=y[1]/half_life[1]
    return dydx

y0=np.asarray([1,0,0])
x0=0
x1=1
ans_rk4=integrate.solve_ivp(fun,[x0,x1],y0)
ans_stiff=integrate.solve_ivp(fun,[x0,x1],y0,method='Radau')
print('took ',ans_rk4.nfev,' evaluations to solve with RK4.')
print('took ',ans_stiff.nfev,' evaluations to solve implicitly')
print('final values were ',ans_rk4.y[0,-1],' and ',ans_stiff.y[0,-1],'
```



```
[>>> exec(open('stiff.py').read())
took 212618 evaluations and 3.127746820449829 seconds to solve with RK4.
took 72 evaluations and 0.0035657882690429688 seconds to solve implicitly
final values were 0.3678794411714445 and 0.3678803705878816 with truth 0.367879441171442
[>>> plt.clf();plt.plot(ans_rk4.t,ans_rk4.y[1,:]);plt.plot(ans_stiff.t,ans_stiff.y[1,:],'*-')
```