12.010 Computational Methods of Scientific Programming

Tom Herring, tah@mit.edu

Chris Hill, cnh@mit.edu

Lecture 6: Solving problems numerically

Topics

- Numerical methods:
  - Differentiation
  - Integration
- Numpy Scipy modules
- ODE solution (using modules from internet)

- Questions on Homework #1?

Numerical methods: Differentiation

- In an earlier Python notebook, we simulated the time evolution of a simple diffusion equation

$$\frac{\partial \phi}{\partial t} = \kappa \frac{\partial^2 \phi}{\partial x^2}$$

We did this starting from some initial conditions $\phi = \phi_0$ for some starting time t=0 and integrating forward in time from t=0 to $t=t_{final}$

We had to numerically evaluate the spatial derivative $\frac{\partial^2 \phi}{\partial x^2}$ and numerically integrate forward in time

We will now look in a bit more detail at numerical differentiation and integration and then look at tools in SciPy that can be used for simulating differential equations of various sorts.

Numerical methods: Differentiation

- Types of algorithms
  - Mathematical/symbolic
    - $\frac{d}{dx}x^2 = \lim_{h\to 0}\frac{(x+h)^2-x^2}{h} = \lim_{h\to 0}\frac{x^2+2xh+h^2-x^2}{h} = \lim_{h\to 0}\frac{2xh+h^2}{h}$=2x

    or

    $\frac{d}{dx}x^n$=nx^{n-1}
  - Numerical approaches use a finite “h” (“mesh” size) to approximate derivative e.g

```
x=2;
h=0.001;
print( ( (x+h)**2.-x**2. )/h )
```

```
4.000999999999699
```

Numerical methods: Differentiation

- Error estimating
  - With step size of h=0.001

```
x=2;
h=0.001;
print( ( (x+h)**2.-x**2. )/h )
```

```
4.000999999999699
```

  - The error is

```
4.000999999999699-4
```

```
0.0009999999996992415
```

  - Can we just keep reducing step size to get more accuracy?

Numerical methods: Differentiation

- Lets try reducing step size

```
x=2;
h=1.e-3;
print( ( (x+h)**2.-x**2. )/h  - 4. )
```

```
0.0009999999996992415
```

```
x=2;
h=1.e-7;
print( ( (x+h)**2.-x**2. )/h  - 4. )
```

```
9.115331067732768e-08
```

```
x=2;
h=1.e-5;
print( ( (x+h)**2.-x**2. )/h  - 4. )
```

```
1.0000027032219805e-05
```

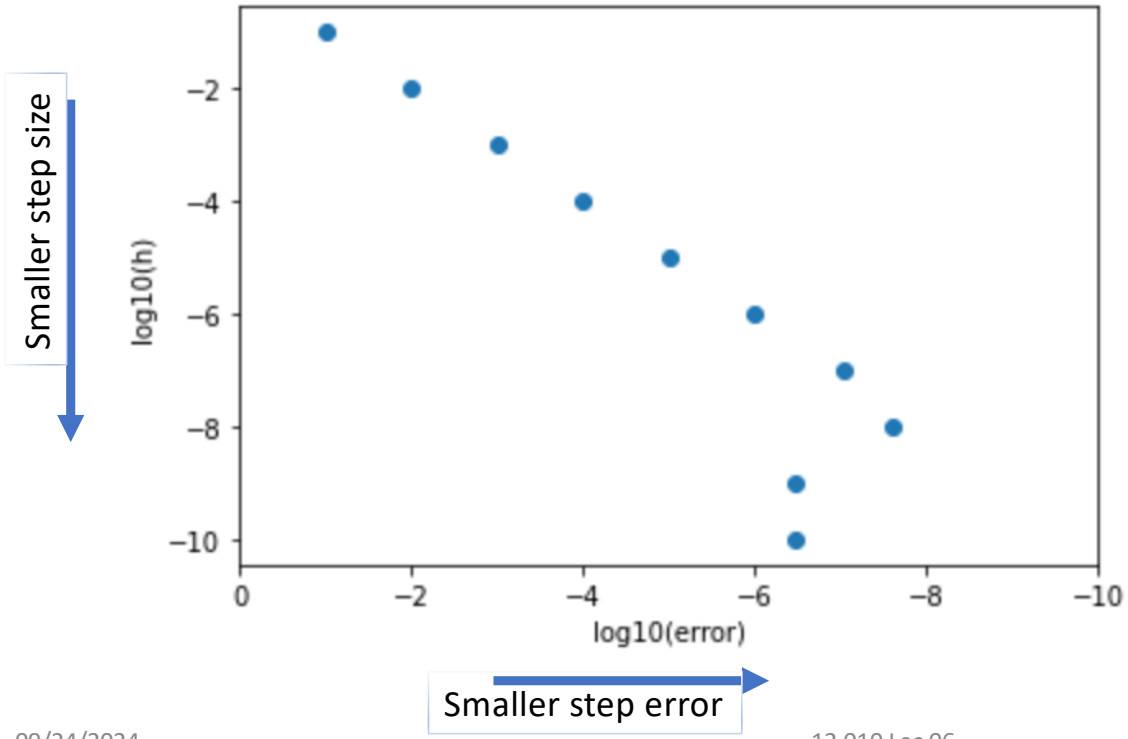
```
x=2;
h=1.e-15;
print( ( (x+h)**2.-x**2. )/h  - 4. )
```

```
-0.4472863211994995
```

- works up to a point, but then finite precision numbers start to limit

Numerical methods: Differentiation*

- We can test reducing step size in a notebook



Error gets smaller with step size, to a point. Then numerical truncation impacts.

Later we will look in libraries to help with this.

Numerical methods: Differentiation

- What about higher-order derivatives

  - $\frac{d^2\phi}{dx^2} \equiv \frac{d}{dx}\frac{d\phi}{dx}$ so we can apply our formula repeatedly

  this gives one discrete formula for higher-order derivatives e.g

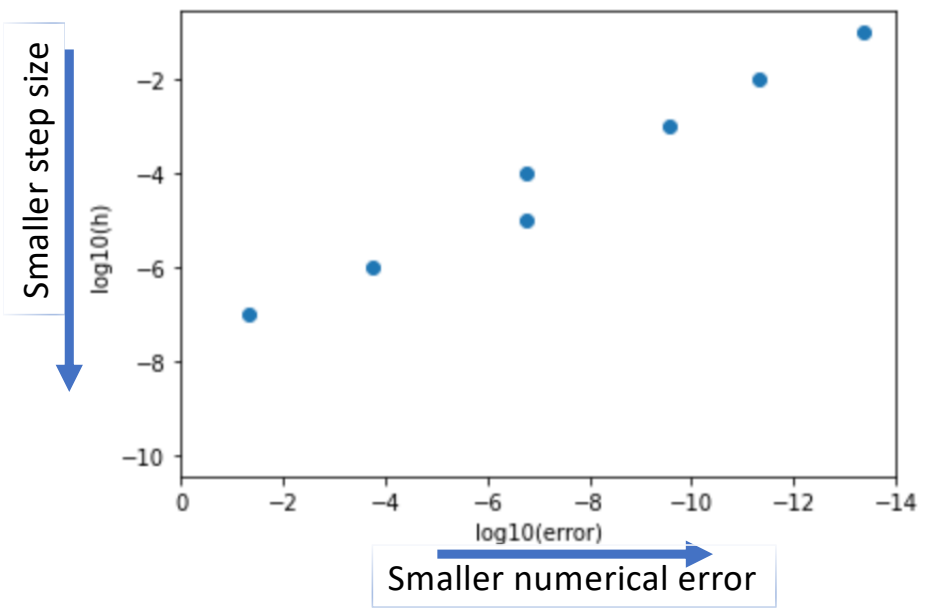
$$\frac{d^2 f(x)}{dx^2} \approx \frac{f(x-h) + f(x+h) - 2f(x)}{h^2}$$

Numerical methods: Differentiation

- Testing a second order derivative for $f(x)=x^2$ with x=2

```
def n_d2fdx2(x,h):
    return (f(x-h)+f(x+h)-2*f(x))/(h**2)
def a_d2fdx2(x):
    return 2
```

in this case we find smallest error (10^{-13}) is for largest step size (h=0.1)!



Numerical methods: Differentiation

- Try with notebook *Lec06-deriv.ipynb*

Numerical methods: Integration

- Symbolic rules e.g.

$$\int x^n dx = \frac{1}{n+1} x^{n+1} + C$$

➔ $\frac{1}{n+1} b^{n+1} - \frac{1}{n+1} a^{n+1}$

For definite integral $\int_a^b x^n dx$

- Like derivative numerical is discrete, so applies to definite integral

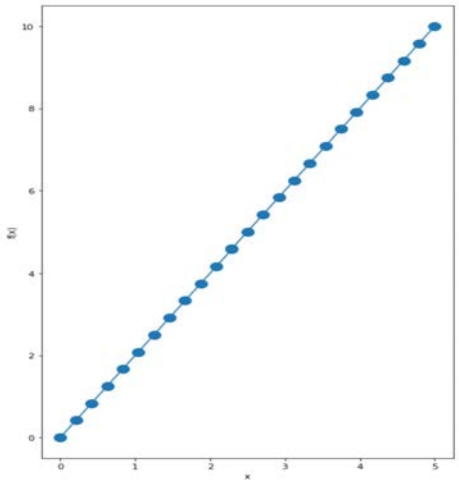
$$\int_{x=a}^{x=b} x^n dx \approx \sum_{i=0}^{i=N-1} \frac{1}{2} [(a + i\Delta)^n + (a + (i+1)\Delta)^n] \Delta$$

Numerical methods: Integration

- Basic Example 1

  Lets take a really simple function, $y = 2x$, where we know analytically the integral is simply x^2.

  We can evaluate this function at discrete points



```
import numpy as np
import matplotlib.pyplot as plt
def fx(x):
        return 2*x
xvals= np.linspace(0,5,25)
plt.figure(figsize=(16, 12))
plt.plot(xvals,[fx(x) for x in xvals],'o-',markersize=12);
```

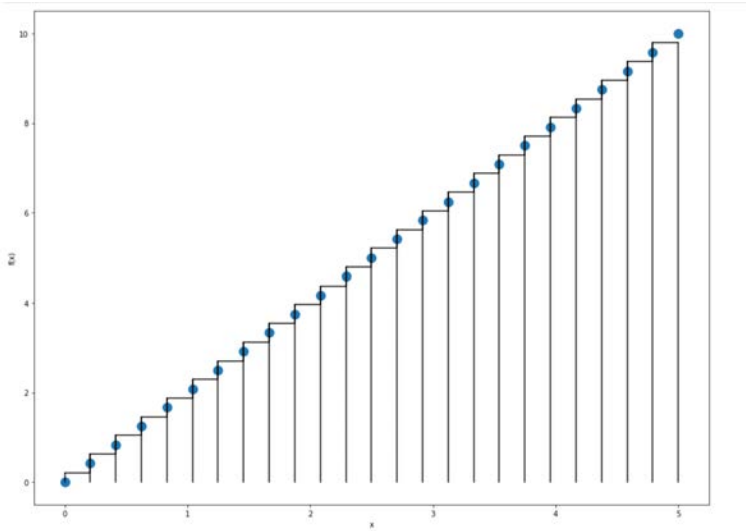
Numerical methods: Integration

- Basic Example 1

  Our simple discrete integral entails creating approximate areas
  $\int_{x=a}^{x=b} x^n dx \approx \sum_{i=0}^{i=N-1} \frac{1}{2}[(a+i\Delta)^n + (a+(i+1)\Delta)^n]\,\Delta$
  under our function and summing

  Each of the rectangles is of width Δ

  The height is the average of the two function evaluations bracketing e.g
  $\frac{1}{2}[(a+i\Delta)^n + (a+(i+1)\Delta)^n]$



Numerical methods: Integration

- Basic Example 1

  A simple integrator function would be

```
# Now lets make a very simple integrator function
def myint(fx, xs ):
    total_a=0
    dx=xs[1]-xs[0]
    for xval in xs[:-1]:
        midy=0.5*(fx(xval)+fx(xval+dx))
        a=midy*dx
        total_a=total_a+a
    return total_a
```

fx – is the function we defined earlier
xs – are pre-defined points to evaluate fx()
dx – is our step size Δ

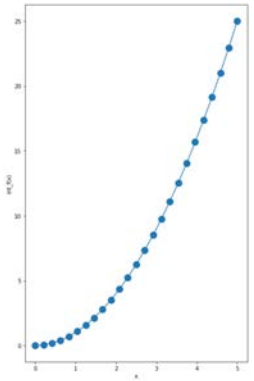
Numerical methods: Integration

- Basic Example 1

  Lets see how it works for f(x)=2x!

  We know the indefinite integral in this case, so we can create a function for that too and plot it.

```
def a_int_fx_lin(x):
        return x**2
```



```
plot(xvals,[a_int_fx(x) for x in xvals],'o-',markersize=12);
```

Numerical methods: Integration

- Basic Example 1

  Lets compare numerical and analytic integral

```
ni=myint(fx,xvals)
ai=( a_int_fx(xvals[-1]) + a_int_fx(xvals[0]) )
print("Numerical integral =", ni  )
print("Error = ", ni - ai )
print("Percent error =",(ni-ai)/(0.5*(ni+ai))*100,"%")
```

```
Numerical integral = 25.0
Error =  0.0
Percent error = 0.0 %
```

ni – is the numerical integral
ai – is the analytic integral (the first value, `a_int_fx(xvals[0])`, is the constant of integration – 0 in this case)

Works perfectly – is that odd?

Numerical methods: Integration

- Basic Example 1

  We can apply to different functions by redefining fx() e.g.

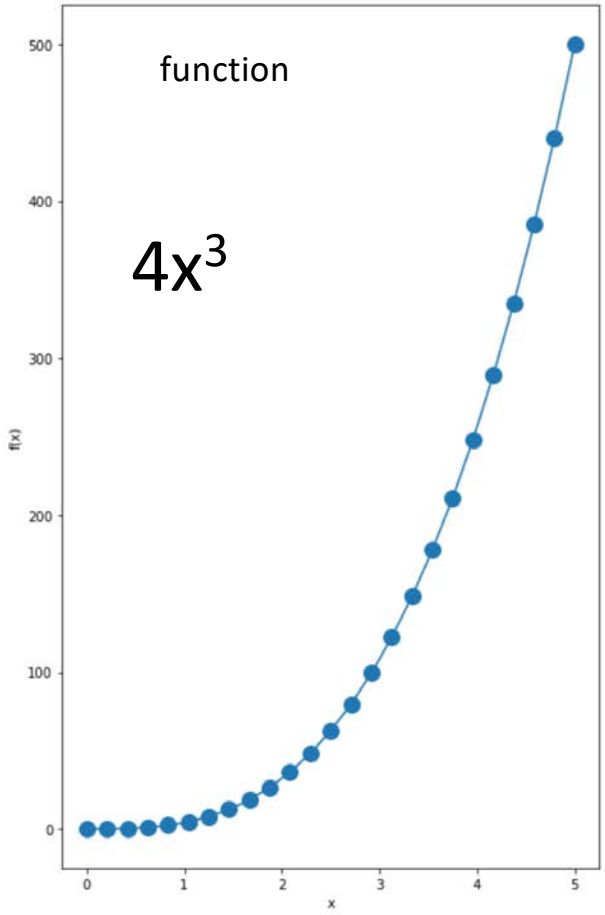
```
def fx_pow(x):
    # print("fx evlauted at x=",x)
    return 4.*x**3

# Lets define its analytic integral is w
def a_int_fx_pow(x):
    return x**4
```

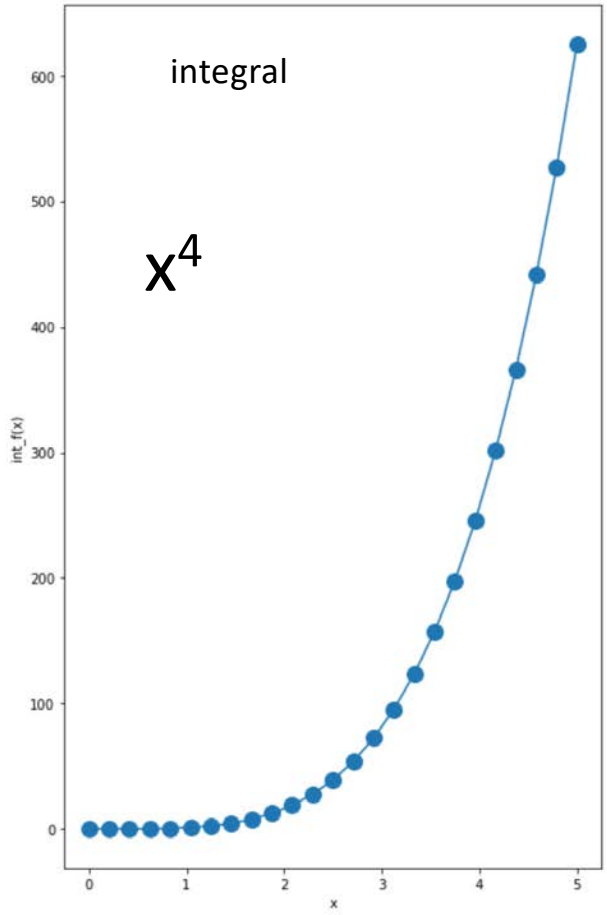
```
def fx_sin(x):
    import math
    # print("fx evlauted at x=",x)
    return math.sin(x)

# Lets define its analytic integral is
def a_int_fx_sin(x):
    import math
    return -math.cos(x)
```

these are not straight lines – how do they work

Numerical methods: Integration

- $f(x) = 4x^3$





Numerical methods: Integration

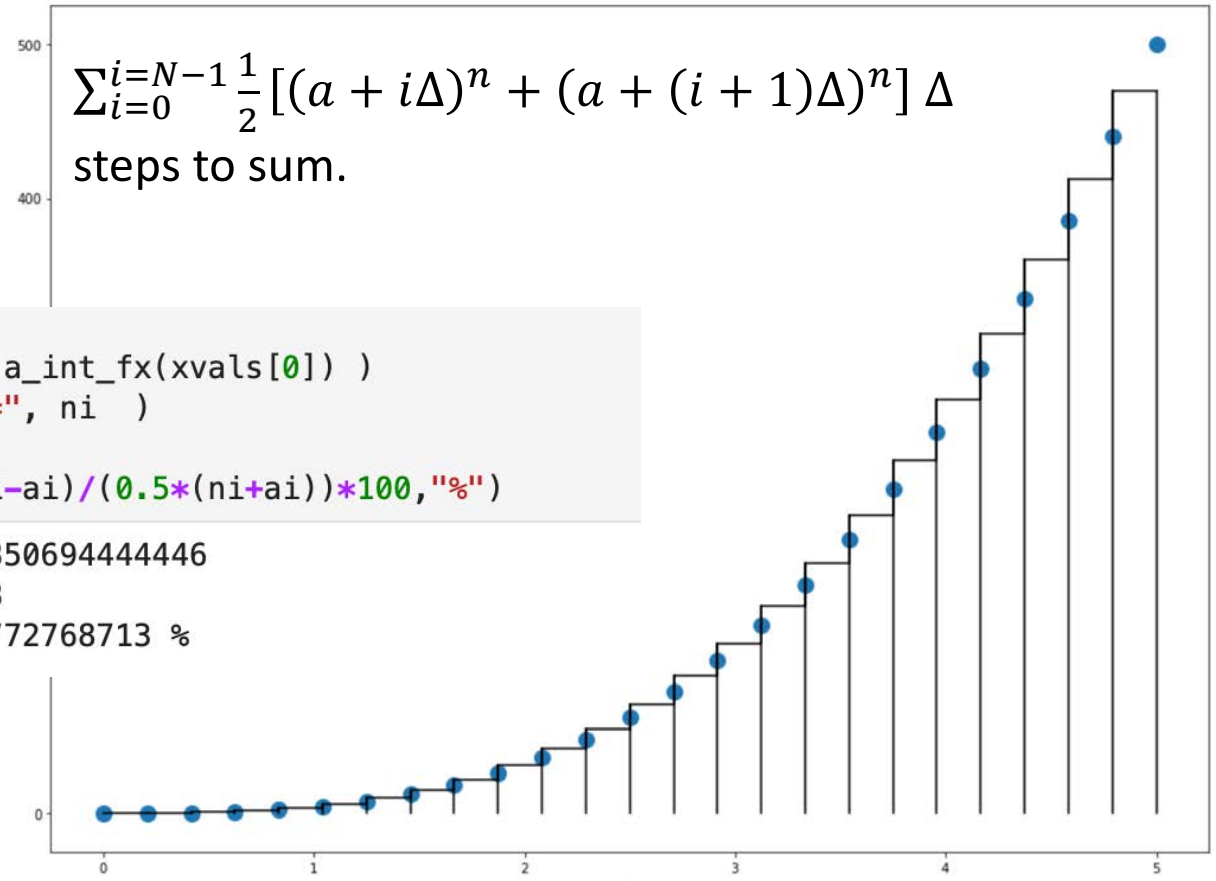
- f(x) = $4x^3$

$\sum_{i=0}^{i=N-1} \frac{1}{2}[(a+i\Delta)^n + (a+(i+1)\Delta)^n]\,\Delta$

steps to sum.

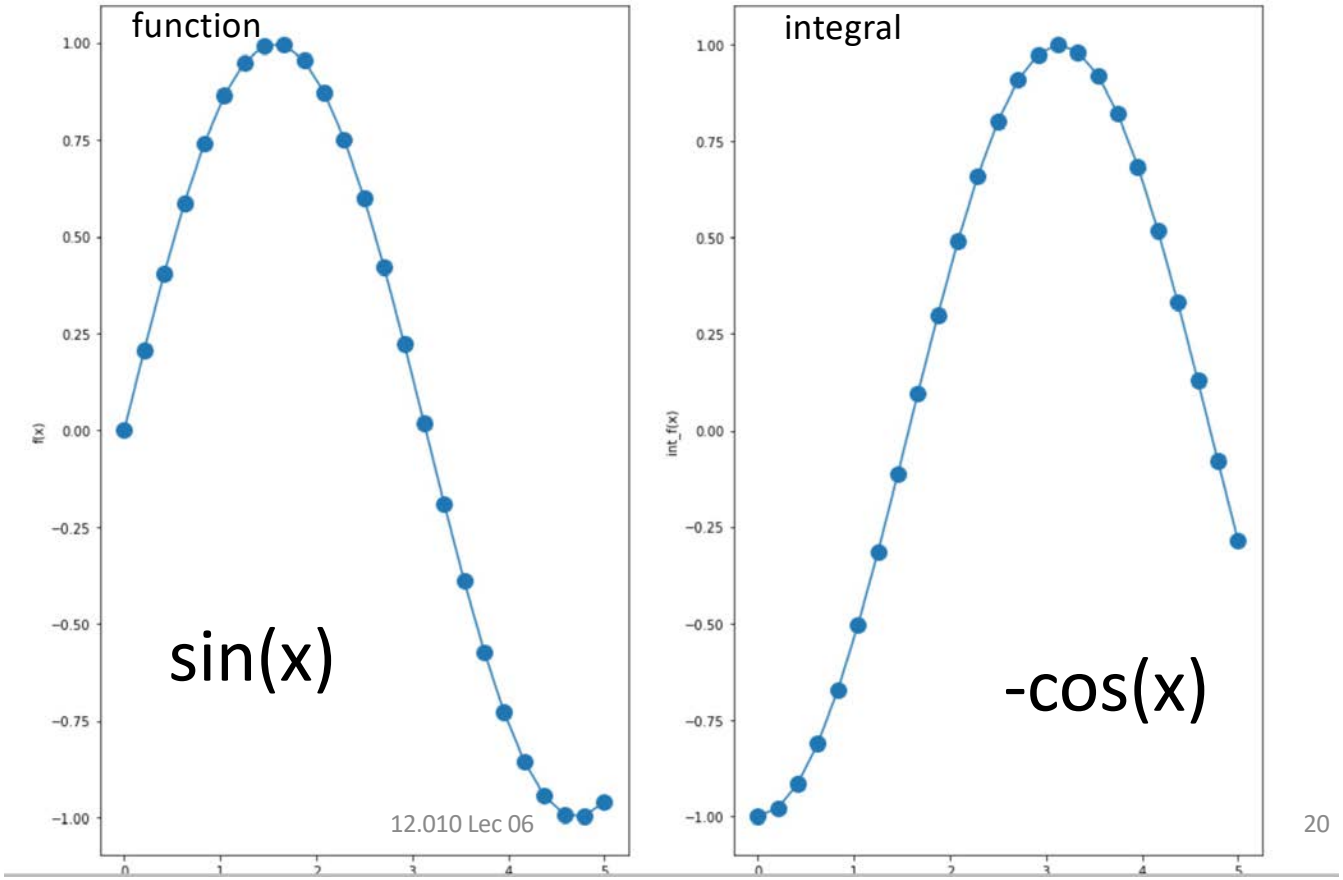
```
ni=myint(fx,xvals)
ai=( a_int_fx(xvals[-1]) - a_int_fx(xvals[0]) )
print("Numerical integral =", ni  )
print("Error = ", ni - ai )
print("Percent error =",(ni-ai)/(0.5*(ni+ai))*100,"%")
```

```
Numerical integral = 626.0850694444446
Error =  1.0850694444445708
Percent error = 0.17346053772768713 %
```



Numerical methods: Integration

- f(x) = sin(x)

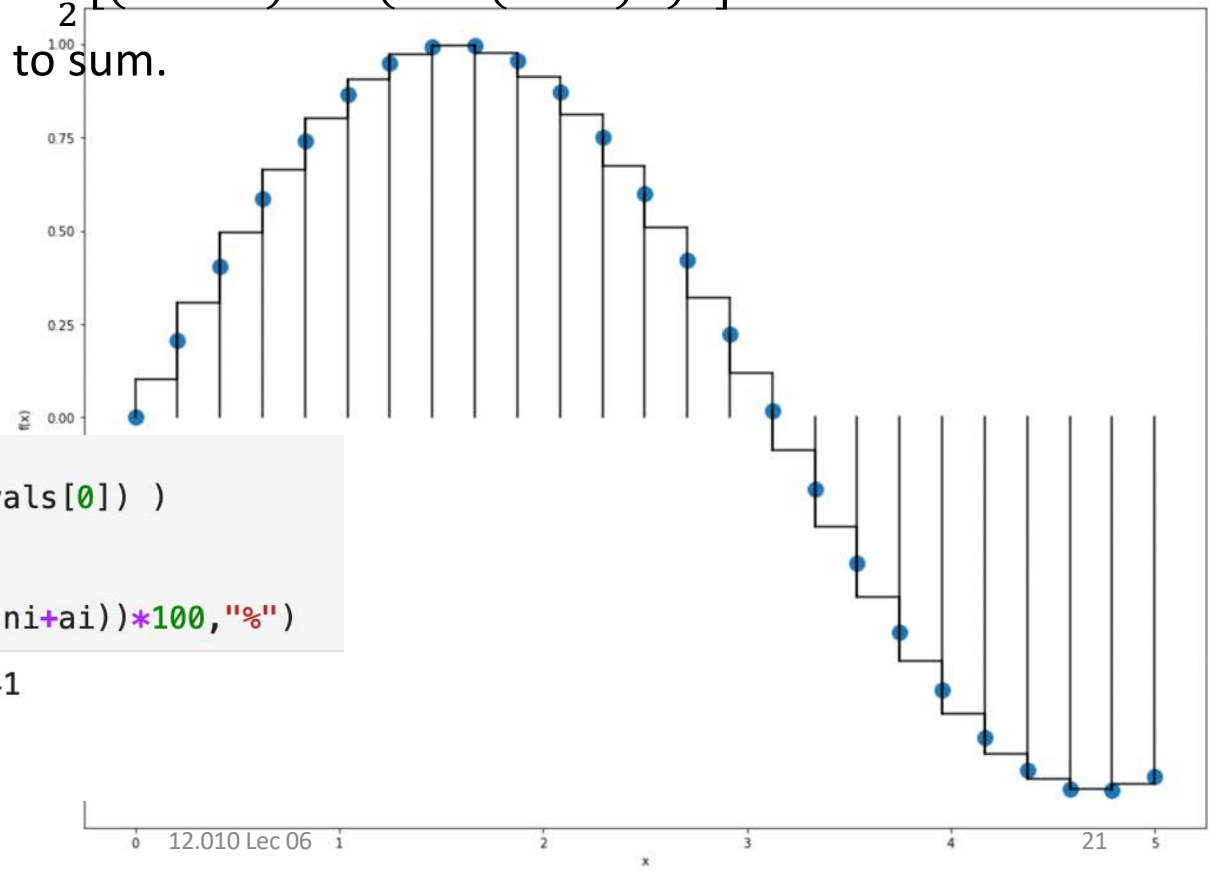


Numerical methods: Integration

$\sum_{i=0}^{i=N-1} \frac{1}{2}[(a+i\Delta)^n + (a+(i+1)\Delta)^n]\,\Delta$

- f(x) = sin(x)

steps to sum.




```
ni=myint(fx,xvals)
ai=( a_int_fx(xvals[-1]) - a_int_fx(xvals[0]) )
print("Numerical integral =", ni  )
print("Error = ", ni - ai )
print("Percent error =",(ni-ai)/(0.5*(ni+ai))*100,"%")
```

```
Numerical integral = 0.7137450174635941
Error =  -0.00259279707317972
Percent error = -0.3626079574080298 %
```

Numpy/Scipy modules

- The Python Scipy package (https://docs.scipy.org/doc/scipy/index.html) contains advanced functions for numerical integration (and differentiation) working on Numpy arrys.

- These operate similarly to the basic examples shown
  1. define some function f(y,t) that evaluates a function at points y, t (y can be vector of values)
  2. invoke a “control” function to evaluate the derivative or integral over some domain
- The Scipy functions have many sophisticated features built-in to adjust step sizes and select where to evaluate functions and how to combine evaluations. These can reduce error relative to simple approaches.

Numpy/Scipy modules - odeint

- To fit sin(x) function, define container function
- Treat x as the t variable in odeint.
- Invoking odeint will integrate our function and return estimated values of integral at xvals locations.

```
def fx_sin(x):
    import math
    # print("fx evlauted at x=",x)
    return math.sin(x)
```

```
    def fx(x):
        return fx_sin(x)
```

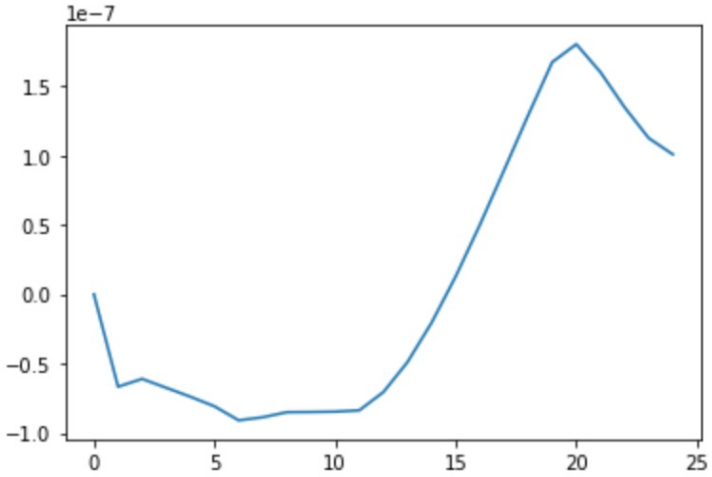
```
from scipy.integrate import odeint
def fgenx(y,t):
    return fx(t)
ys = odeint(fgenx, 0, xvals )
```

Numpy/Scipy modules - odeint

- odeint is usually quite accurate without any optional parameters.

```
from scipy.integrate import odeint
def fgenx(y,t):
    return fx(t)
ys = odeint(fgenx, 0, xvals )
ni=ys[-1]
print("Numerical integral =", ni  )
print("Error = ", ni - ai )
print("Percent error =",(ni-ai)/(0.5*(ni+ai))*100,"%")
errvec=np.abs(ys.flatten()-[ a_int_fx(xv) for xv in xvals]) + a_int_fx(xvals[0])
plt.plot(errvec);
```

```
Numerical integral = [0.71633792]
Error =  [1.00731731e-07]
Percent error = [1.40620419e-05] %
```



Numerical Integration and Numpy/Scipy modules

- try with notebook *Lec06-integration.ipynb*

MIT OpenCourseWare

https://ocw.mit.edu

12.010 Computational Methods of Scientific Programming, Fall 2024