Lecture 13: Breakdown of integration, broke down

Let's start by looking at the:

Euler-Maclaurin formula: Recap

$$\int_0^h f(x)\,\mathrm{d}x \;=\; \frac{h}{2}(f(h) \;+\; f(0)\,) - \frac{h^2}{12}\left(f'(h) - f'(0)\right) \;+\; \frac{h^4}{720}\left(f'''(h) - f'''(0)\right) - \frac{h^6}{6!}\int_0^h f^{\,(5)}(x)\,\frac{d}{dx}$$

Remember roughly how we got here:

We wanted the integral,

$$I \;=\; \int_0^h f(x)\,\mathrm{d}x$$

For example, with the **trapezoidal rule**, we approximated

$$f(x) \;\approx\; L(x) \;=\; \frac{x}{h}\,f(h) + \frac{h-x}{h}\,f(0)$$

And therefore

$$\int_0^h f(x)\,\mathrm{d}x \;\approx\; \int_0^h L(x)\,\mathrm{d}x \;=\; h\,\frac{f(0)+f(h)}{2}$$

Considering the error led to defining the function

$$e(x) \;=\; f(x) - L(x),$$

and error integral,

$$\mathcal{E} \;\equiv\; \int_0^h e(x)\,\mathrm{d}x.$$

This is equivalent to

$$\mathcal{E} \;\equiv\; h\int_0^h e(x)\,\frac{d}{dx}B_1(x/h)\,\mathrm{d}x$$

with

$$B_1(x) \;=\; x \;-\; \frac{1}{2}.$$

Integrating by parts leads to

$$\mathcal{E} \;\equiv\; \frac{h}{2}\,[\;e(h)+e(0)\;] - h\int_0^h e'(x)\,B_1(x/h)\,\mathrm{d}x$$

But recall that $e(h) = e(0) = 0$ by design. Therefore

$$\mathcal{E} \;=\; -\,h\int_0^h e'(x)\,B_1(x/h)\,\mathrm{d}x.$$

We integrate by parts again and get:

$$\mathcal{E} \;=\; -\,\frac{h^2}{12}\big(\;e'(h) - e'(0)\;\big) \;+\; \mathcal{O}\big(\;h^4\;\big)$$

Even though $e(h) = e(0) = 0$, we can't make any guarantee about $e'(h)$ and $e'(0)$. So for the trapezoidal rule we stop here and satisfy ourselves with the notion that the integration is accurate to $\mathcal{O}(\;h^2\;)$.

Looks pretty good? How could we get anything else?

But remember functions of the form:

$$f(x) = x^p, \quad p > 0.$$

With

$$\int_0^h f(x)\,\mathrm{d}x = \frac{h^{p+1}}{p+1}$$

We can compute the linear approximation easily

$$L(x) = h^{p-1}x$$

We have

$$L(0) = f(0) = 0, \quad \text{and} \quad L(h) = f(h) = h^p$$

We also have

$$\int_0^h L(x)\,\mathrm{d}x = \frac{h^{p+1}}{2}$$

Therefore

$$\mathcal{E} = -\frac{2p+1}{p+1}h^{p+1} \quad (\textit{Exactly}). \quad \text{This is } \textit{not } \mathcal{O}(h^2) \text{ unless } p \geq 1.$$

In [2]:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

class power():

    def __init__(self,p):
        self.p = p

    def __call__(self,x):
        return x**self.p

    def __str__(self):
        if self.p == 1: return 'x'
        if type(self.p)==int: return 'x^{{{:d}}}'.format(self.p)
        else: return 'x^{{{:.2f}}}'.format(self.p)

    def area(self,a=0,b=1):
        p = self.p
        return (b**(p+1) - a**(p+1))/(p+1)

    def show(self,a=0,b=1,n=2000):
        x = np.linspace(a,b,n)
        y = self(x)
        fig, ax = plt.subplots()
        ax.plot(x,y,label='${}$'.format(self))
        title = '$f(x) = {}$'.format(self)
        ax.set(title=title,ylabel='$f(x)$',xlabel='$x$',aspect=1)
        return fig, ax

    def q(self, x):
        return Q(self, x)
```

In [3]:

```
def grid(n): return np.linspace(0,1,n+1)
```

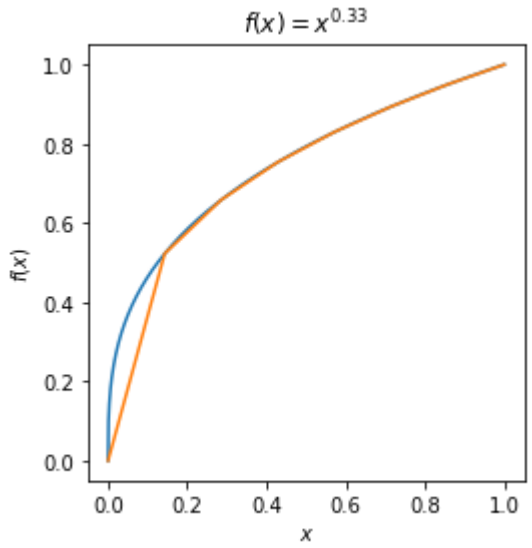
The error near the origin is going to dominate

In [4]:

```
p = 1/3
f = power(p)

fig, ax = f.show()

x = grid(7)
plt.plot(x,f(x))
```

Out[4]:

```
[<matplotlib.lines.Line2D at 0x7fdbd1b3b710>]
```

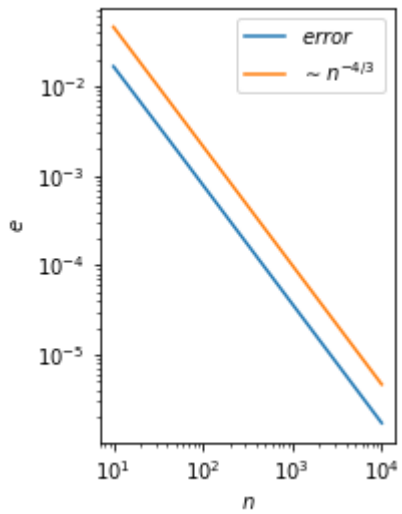


In [5]:

```
nmin, nmax = 10, 10000

def trap(y,x): return (x[1]-x[0])*(0.5*( y[0] + y[-1] )  + np.sum(y[1:-1]) )

n, e = [], []


for k in range(nmin,nmax+1):

    x = grid(k)
    e += [abs(trap(f(x),x) -f.area())/abs(f.area())]
    n += [k]

n, e = np.array(n), np.array(e)

fig, ax = plt.subplots()
ax.loglog(n,e)
ax.loglog(n,n**(-(1+p)))
ax.legend(['$error$', '$\sim n^{-4/3}$'])
ax.set_xlabel("$n$")
ax.set_ylabel("$e$")
ax.set(aspect=1)
```

Out[5]:

[None]



Why not n^{-2} ?

Because

$$f'(x) = p\,x^{p-1}$$

For (example)

$$p = 1/3$$

gives

$$f'(x) = \frac{1}{3\,x^{2/3}}$$

The Euler-Maclaurin formula requires evaluating

$$f'(0) \to \infty.$$

We made an illegal move!

Math is great and can tell you useful things, as ***long as you don't make an illegal move.***

If you do make an illegal move, then you need to go back until you are in the clear. Then you can go forward with a clearer head and figure out how far you can go without stepping over any lines.

The good news

There is another import aspect of the Euler-Maclaurin formula. What happens if we consider a ***periodic function***?

On $0 < x < 1$, this is function where

$$f(0) = f(1), \quad \text{and} \quad f'(0) = f'(1), \quad \text{and} \quad f''(0) = f''(1), \quad \text{and} \quad f'''(0) = f'''(1), \quad \text{and so on ..}$$

This means that all these pesky boundary terms clean up condiderably.

We can add up a bunch of little segments from $0, h, h, 2h, 2h, 3h, \ldots, (n-1)h, 1$. We get

$$\int_0^1 f(x)\,\mathrm{d}x \;=\; \frac{1}{n}\sum_{i=0}^{n-1} f(i\,h) - \frac{h^6}{6!}\int_0^1 f^{\,(5)}(x)\,\frac{d}{dx}P_6(x/h)\,\mathrm{d}x,$$

(Note: it's periodic, so there's a final interval that wraps around, so you double count all points, not just the interior ones. Also, all the integrations from $(0,h), (h,2h), \ldots, ((n-1)h, 1)$ clearly amount to integrating from 0 to 1.)

We can keep going and all the boundary terms will cancel..

$$\int_0^h f(x)\,\mathrm{d}x \;=\; \frac{1}{n}\sum_{i=0}^{n-1} f(i\,h) - \frac{h^k}{k!}\int_0^1 f^{\,(k-1)}(x)\,\frac{d}{dx}P_k(x/h)\,\mathrm{d}x,$$

The function in the integral is now

$$P_k(x) \;\equiv\; B_k(\,x - \lfloor x \rfloor\,)$$

This is the periodically extended Bernoulli polynomial.

The point of this is that the remainder is smaller that *any finite power of* h as $h \to 0$. This imples the sum converges roughly 'exponentially'.

More bad news

I still haven't said anything about multiple dimensions. If we want to do computations is more than one dimension, we need to build multiple loops and sweeps over multiple dimensions. *This is the topic of the tutorial* - **you need to complete it**.

We've just explored the

Blessing of regularity:

```
Having more finite derivatives helps. Being periodic helps even more.
```

In the tutorial, you'll also see about the

Curse of dimensionality:

```
Stick to one dimension if you can at all possibly can.
```

Stated simply:

$$ERROR = \mathcal{O}\left(N^{-\frac{R}{D}}\right)$$

In this case,

$$ERROR = \text{best possible error associated with computing the integral of a function.}$$

$$N = \text{number of function evaluations.}$$

$$R = \text{number of continuous derivatives of the function; or the } \textit{regularity}$$

$$D = \text{dimension of the space on which the function acts.}$$

In real life "function evaluations" can be very difficult. It could mean obtaining real data in the real world.

```
Think of the election data from recent major elections. Predicting electio
ns is just a big integral.

If could also mean running a numerical simulation on a supercomputer and l
ooking at the outcome.

It could mean computing Feynman diagrams for the Large Hadron Collider.

It could mean running a medical study with real hospital patients,

It could be a multi-decade biology field campaign looking at the health of
 the Great Barrier Reef.
```

High dimensional integral are everywhere in all these activities.

What can we do?

Monte Carlo integration

Monte Carlo methods (MCMs) are among the most important algorithms out there. They are our first *statistical* method: we get a remarkably powerful method that is simple to understand as long as we are willing to accept a random error. Why would we do this? Monte Carlo methods shine when the dimensionality of the problem is very large.

The idea of Monte Carlo is incredibly simple: we randomly sample the high-dimensional space of the problem and use that sampling to construct an estimate of the integral we care about. In order to do so, we'll start by exploring random number generators.

Here's the beauty of MCMs:

If x is a random variable with probability density $\rho(x)$, then we can compute the expectation value of any function $f(x)$ via

$$\langle f \rangle \;=\; \int f(x)\rho(x)\,\mathrm{d}x$$

The variable x could be in any number of dimensions. The integral on the right is the thing we want. It would be great if we knew the thing on the left; because that is equal to the value of the integral that we can't do.

Alternatively, we can ***sample*** from the distribution.

$$x_1,\; x_1,\; \ldots,\; x_N$$

The samples should be distributed according to $\rho(x)$.

But the law of Large Numbers implies that

$$\lim_{N\to\infty}\; \frac{1}{N}\sum_{k=1}^{N} f(x_k) \;=\; \langle f \rangle$$

Or simply,

$$\int f(x)\,\rho(x)\,\mathrm{d}x \;\approx\; \frac{1}{N}\sum_{k=1}^{N} f(\,x_k\,)$$

For uniform random samples, $\rho(x) = 1$. By definition, this computation requires N function evaluations. This looks a lot like a quadrature scheme; it is in some ways. But the randomness helps. It cancels errors.

The big deal is that for MCMs:

$$ERROR \;=\; \mathcal{O}\Big(\,N^{-\frac{1}{2}}\,\Big)$$

This may not look great. Recall we were getting exponents of -4 with Simpsons. And even exponential decrease sometimes. An exponent of $-1/2$ is rather slow convergences. But:

The rate of convergence is: *independent of the dimension and regularity of the function.*

What's more, the function evaluations could be very computationally difficult. For example, something involving a large matrix. But the individual evaluations can all happen simultaneously. This means more computers will get the job done faster.

The need to take high-dimensional integrals, is partly what drives the demand for computing power in several industries; the financial industry is a particular example.

We'll likely return to statistical methods and MCMs later in the semester. There are several ways to improve on the simple techniques described here.

Questions!

- Why does the dimension D affect the error?
- How can you use the above to determine the error of Simpson's method?

$R = 1$

$D = 4$

$ERROR = cN^{-R/D}$

Let's make some random samples in 2D:

In [112]:

```
def make_data(dim, nsamples):
    r = np.random.rand(dim,nsamples)
    return r

r = make_data(2,200)
```

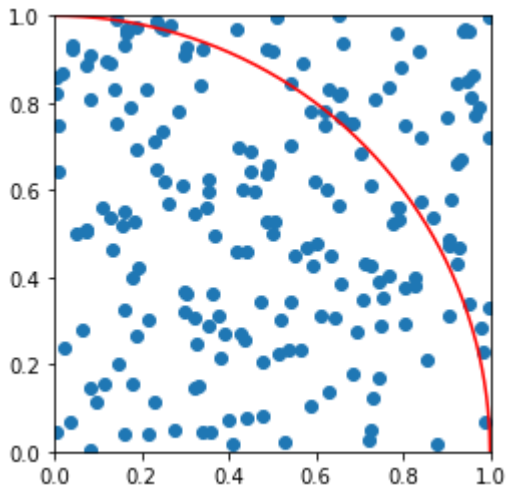
Let's take a look at the samples:

In [113]:

```
fig, ax = plt.subplots()

im = ax.scatter(r[0], r[1], cmap=plt.cm.jet)
t = np.linspace(0,1,1000)
ax.plot(t,np.sqrt(1-t**2),color='red')
ax.set_aspect('equal')
ax.set_xlim([0,1])
ax.set_ylim([0,1])
```

Out[113]:

```
(0, 1)
```



The area under the red curve is

$$\frac{\pi}{4} = \int_0^1 \sqrt{1-t^2}\,\mathrm{d}t$$

A quadrature rule would give disappointing results becuase of the square root singularity at $t = 1$.

Let's make a function the tells us if a point is in the circle:

In [228]:

```
def S(r):
    return 1.0*(np.sum(r**2,axis=0) < 1)
```

Here's what it looks like:

In [229]:

```
S(r)
```

Out[229]:

```
array([0., 1., 1., ..., 1., 0., 1.])
```

Let's make a fuction that does the integral:

In [230]:

```
def integrate(S,r):
    return (2**len(r))*np.sum(S(r))/r.shape[-1]
```

In [312]:

```
def print_error(dim, points):
    r = make_data(dim,points)
    I = integrate(S,r)
    print('Dimension {:d}'.format(dim))
    print('Points {:d}'.format(points))
    print('Volume {:.4f}'.format(V[dim]))
    print('Guess = {:.4f}'.format(I))
    print('Error estimate 1/sqrt(N) = {:.3e}'.format(1/np.sqrt(points)))
    print('Fractional Error = {:.3e}'.format(np.abs(I - V[dim])/V[dim]))
```

In [313]:

```
print_error(2,200)
```

```
Dimension 2
Points 200
Volume 3.1416
Guess = 3.0200
Error estimate 1/sqrt(N) = 7.071e-02
Fractional Error = 3.870e-02
```

Not great. But we should only expect it to be accurate to about $1/\sqrt{200} \approx 7\ \%$

More points:

In [314]:

```
print_error(2,10**6)
```

```
Dimension 2
Points 1000000
Volume 3.1416
Guess = 3.1391
Error estimate 1/sqrt(N) = 1.000e-03
Fractional Error = 7.934e-04
```

A lot of points, but much better. What about in 3D?

In [315]:

```
V = {2:np.pi,3:4*np.pi/3,4:(np.pi**2)/2,5:8*(np.pi**2)/15}
```

In [316]:

```
print_error(3,10**6)
```

```
Dimension 3
Points 1000000
Volume 4.1888
Guess = 4.1842
Error estimate 1/sqrt(N) = 1.000e-03
Fractional Error = 1.090e-03
```

About the same error as in 2D!

What about 4D?

In [317]:

```
print_error(4,10**6)
```

```
Dimension 4
Points 1000000
Volume 4.9348
Guess = 4.9335
Error estimate 1/sqrt(N) = 1.000e-03
Fractional Error = 2.663e-04
```

5D?

In [318]:

```
print_error(5,10**6)
```

```
Dimension 5
Points 1000000
Volume 5.2638
Guess = 5.2589
Error estimate 1/sqrt(N) = 1.000e-03
Fractional Error = 9.265e-04
```

We don't have the answer in 6D +. But we can compute it.

In [323]:

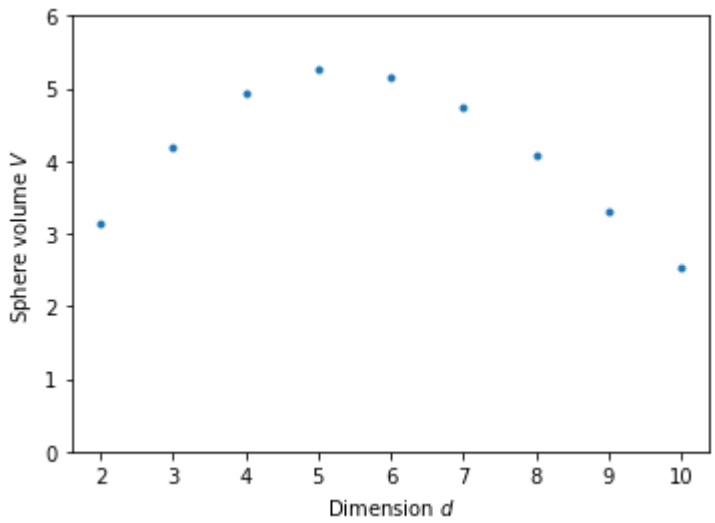
```
npoints = 10**6
large_dim = [integrate(S,make_data(d,npoints)) for d in range(2,11)]
```

In [324]:

```
fig, ax = plt.subplots()
ax.plot(np.arange(2,11),np.array(large_dim),'.')
ax.set(aspect=1,xlabel='Dimension $d$',ylabel='Sphere volume $V$',ylim=[0,6])
```

Out[324]:

```
[(0, 6), Text(0,0.5,'Sphere volume $V$'), Text(0.5,0,'Dimension $d
$'), None]
```



It seems like the numbers are decreasing past 5D. Weird. It turns out this is true.

High dimensional spheres (https://en.wikipedia.org/wiki/Volume_of_an_n-ball) are *spiky*: that is, they fill almost none of the box they're in as you increase the dimension