Table of Contents

Lecture 11: Adding more integration

Recap

Last time we discussed the trapezoidal rule for numerical integration.

We found that

$$\int_a^b f(x)\,\mathrm{d}x \;=\; \frac{1}{n}\left(\frac{f(x_0)+f(x_n)}{2} \;+\; \sum_{k=1}^{n-1} f(x_k)\right) \;+\; \mathcal{O}(n^{-2}) \quad \text{as} \quad n \to \infty$$

The term $\mathcal{O}(n^{-2})$ means

$$\lim_{n\to\infty} n^2 \left[\int_a^b f(x)\,\mathrm{d}x - \frac{1}{n}\left(\frac{f(x_0)+f(x_n)}{2} \;+\; \sum_{k=1}^{n-1} f(x_k)\right)\right] \;=\; \text{Constant}$$

Today, we're going to look at another way of approximating a function for integration.

Simpson's rule

Rather than fitting $f(x)$ to a straight line $L(x)$, we can fit it to a quadratic function,

$$Q(x) = \alpha x^2 + \beta x + \gamma.$$

The fitting rules are

$$f(a) = Q(a) \quad f(b) = Q(b) \quad f(m) = Q(m)$$

where

$$m = \frac{a+b}{2}$$

We can solve for α, β, γ in $Q(x)$. The algebra is a little messy, but it cleans up a bit

$$Q(x) = \frac{2(x-b)(x-m)}{(b-a)^2} f(a) - 4\,\frac{(x-b)(x-a)}{(b-a)^2} f(m) + \frac{2(x-a)(x-m)}{(b-a)^2} f(b)$$

We can integrate the quadratic to get the approximate integral

$$\int_a^b Q(x)\,\mathrm{d}x = \frac{b-a}{6}(f(a) + 4f(m) + f(b))$$

(Composite) Simpson's rule

We can string together a bunch of Simpsons intervals and get a composite rule:

$$\int_a^b f(x)\,\mathrm{d}x = \frac{h}{3}\left(f(x_0) + f(x_n) + 2\sum_{k=1}^{n/2-1} f(x_{2k}) + 4\sum_{k=1}^{n/2} f(x_{2k-1})\right) + \mathcal{O}(n^{-4}) \quad \text{as} \quad n \to \infty$$

Where

$$h = \frac{b-a}{n}$$

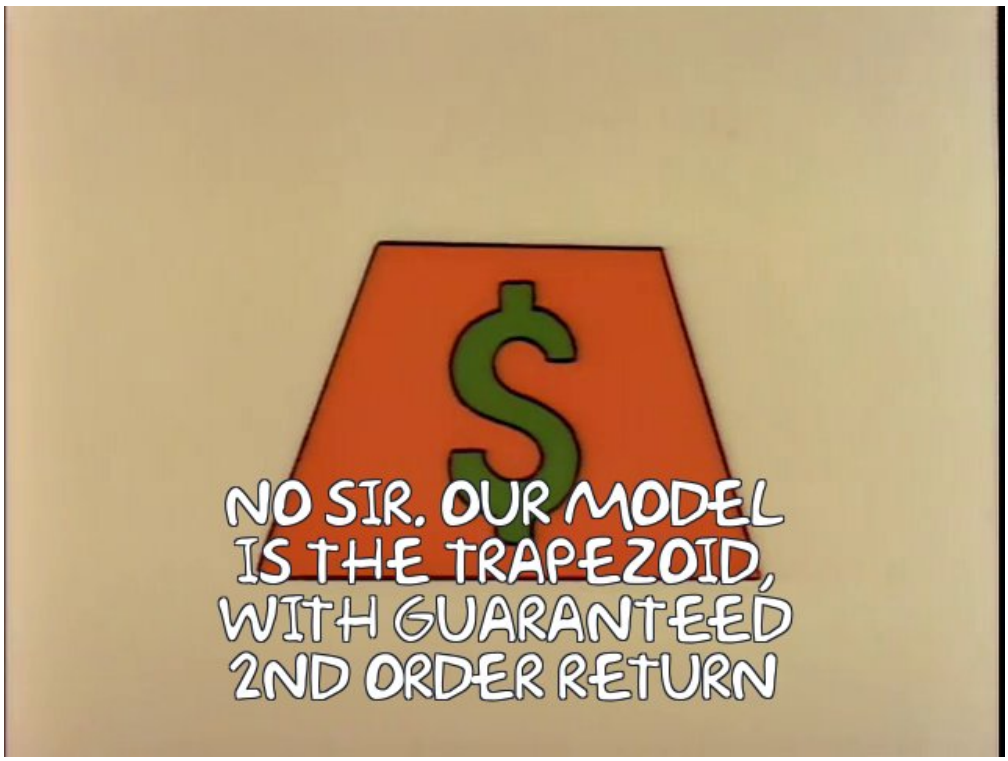
is the spacing between successive points.

Note that single Simpsons rule requires **three** function evaluations and **two** intervals ($[a, m]$, $[m, b]$) -- instead of two function evaluations for one interval $[a, b]$ for the trapezoidal rule.

Therefore, **the number of *points* must be odd; the number of *intervals* must be even.**

Be careful of the indexing choice! The number n here is the number of *intervals* -- one less than the number of points.

Simpsons vs. Trapezoid



(The skit (https://www.youtube.com/watch?v=uuVh36vQkSI))

Let's get comparing!

In [1]:

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

from scipy.integrate import simps as simp
```

In [2]:

```
def trap(y,x):
    """Integrate y values against x values using trapezoidal rule."""
    return (x[1]-x[0])*( 0.5*(y[0] + y[-1]) + np.sum(y[1:-1]) )

def g(n): return np.linspace(0,1,2*(n//2)+1)

def h(n): return 1/(2*(n//2))

def Q(f,x):
    """A quadratic interpolation polynomial."""
    a, m, b = x[0], x[len(x)//2], x[-1]

    qa = 2*(m-x)*(b-x)/(b-a)**2
    qm = 4*(x-a)*(b-x)/(b-a)**2
    qb = 2*(x-a)*(x-m)/(b-a)**2

    return qa*f(a) + qm*f(m) + qb*f(b)
```

These are a few helper functions to make plots and such.

In [3]:

```
def err(A,B=1,report=True):
    """Fractional error of A with respect to B.

    Optionally print errors.
    """
    e = abs(A-B)/abs(B)
    if report : print("T, S, |T-S|/|S| : {:>8.2e}, {:>8.2e}, {:>8.2e} percent".f
ormat(A,B,100*e))
    return e
```

In [4]:

```
def loglog_plot(n,e,c=1,c2=1,p=-2,p2=None,title=''):
    """Log-log plot with optional guide lines."""
    fig, ax = plt.subplots()
    ax.loglog(n,e)
    ax.loglog(n,c*n**float(p))
    if p2 != None:
        ax.loglog(n,c2*n**float(p2))
    ax.legend(['$error$', '$\sim n^{'+str(p)+'}$','$\sim n^{'+str(p2)+'}$'])
    ax.set_xlabel("$n$")
    ax.set_ylabel("$e$")
    ax.set_title(title)
    return fig, ax
```

In [5]:

```
def errors(f,I,pT=-2,cT=1,pS=-4,cS=10):
    """Calculate and plot errors of trapezoidal and Simpson's rule."""
    nmin, nmax = 10, 10000
    n, eT, eS = [], [], []

    for k in range(nmin,nmax):

        x, dx = g(k), h(k)
        y = f(x)

        T = trap(y,x)
        S = simp(y,x)

        n += [k]
        eT += [err(T,I,report=False)]
        eS += [err(S,I,report=False)]

    n, eT, eS =  np.array(n), np.array(eT), np.array(eS)

    f1,a1 = loglog_plot(n,eT,c=cT,p=pT,title='Trap')
#     a1.set(aspect=.5)
    f2,a2 = loglog_plot(n,eS,c=cS,p=pS,title='Simp')
#     a2.set(aspect=.5)
    return (f1,a1), (f2,a2)
```

Like last time, start with

$$f(x) = \frac{x}{\sqrt{2}} + \frac{1}{5}\sin(9x).$$

With

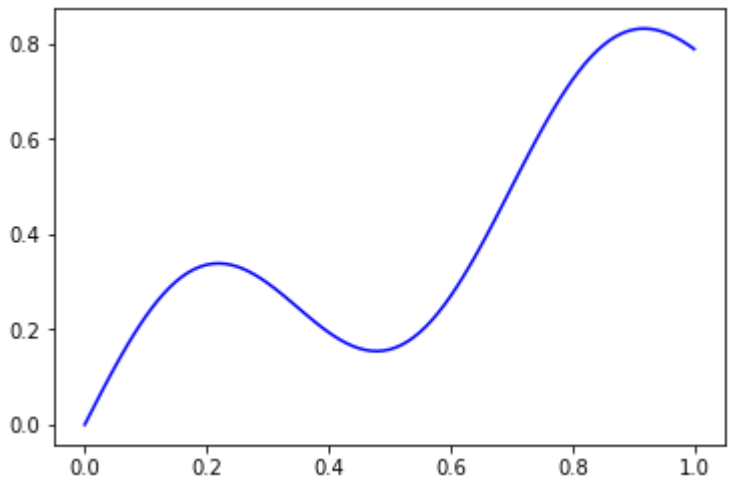
$$\int_0^1 f(x)\,dx = \frac{1}{180}\left(4 + 45\sqrt{2} - 4\cos(9)\right) \approx 0.396023$$

In [6]:

```
def f(x): return np.sqrt(0.5)*x + 0.2*np.sin(9*x)

I = (4 + 45*np.sqrt(2) - 4*np.cos(9))/180

x = np.linspace(0,1,101)

fig, ax = plt.subplots()
ax.plot(x,f(x),color='blue')
```

Out[6]:

```
[<matplotlib.lines.Line2D at 0x7fb0f926a978>]
```



In [7]:

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

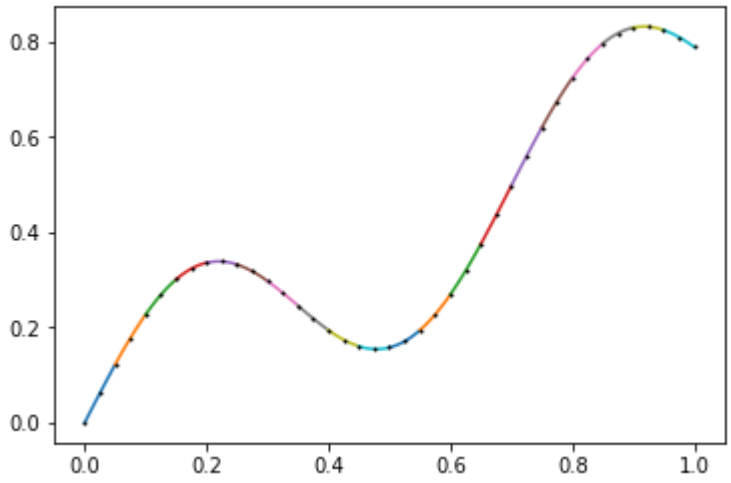
Let's plot Simpson's approximation, and highlight the different intervals:

In [8]:

```
fig, ax = plt.subplots()
n = 20
for k in range(n):
    interval = [k/n, (k+1)/n]
    x = np.linspace(*interval,101)
    ax.plot(x,q(x))
# plot the interval points
points = np.linspace(0,1,2*n+1)
ax.plot(points,f(points),'k.',markersize=2)
```

Out[8]:

```
[<matplotlib.lines.Line2D at 0x7fb0f9164518>]
```



Let's look at the errors of the trapezoidal and Simpson's rule.

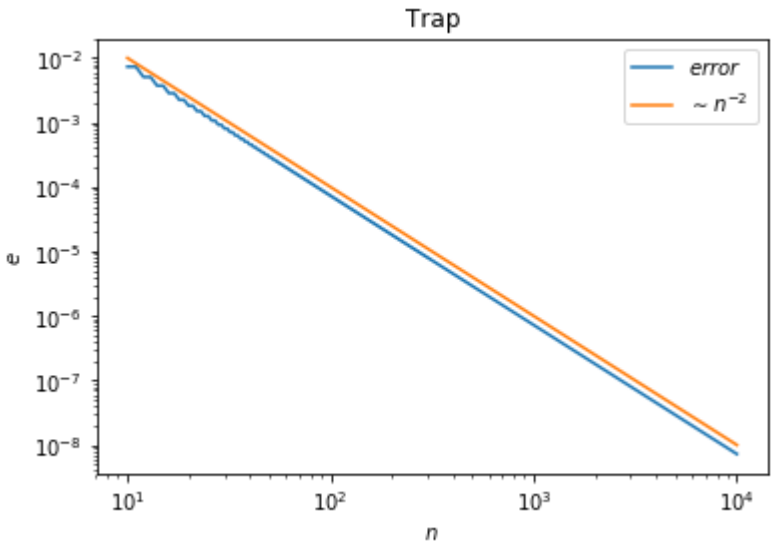
This will loop over different values of n and make a log-log plot of the errors for both methods. It also returns the plot figures and axes in case we want to do something with them.

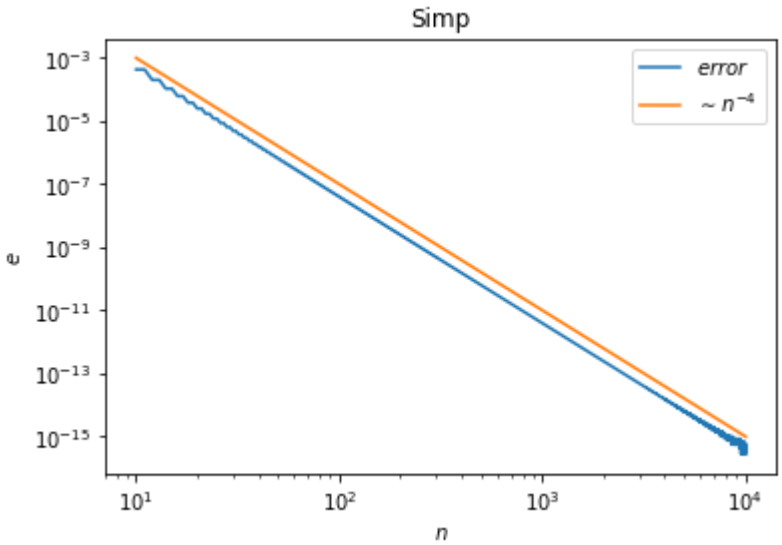
In [9]:

```
errors(f,I)
```

Out[9]:

```
((<Figure size 432x288 with 1 Axes>,
  <matplotlib.axes._subplots.AxesSubplot at 0x7fb0f9268da0>),
 (<Figure size 432x288 with 1 Axes>,
  <matplotlib.axes._subplots.AxesSubplot at 0x7fb0f90c7fd0>))
```





So the trapezoidal rule is second order, and Simpson's rule is 4th order!

Is this always the case?

We saw something funny when we tried periodic functions last week. Let's use

$$f(x) = \frac{1}{2} + \frac{3}{10 + 8\cos(2\pi x)}$$

since

$$\int_0^1 f(x)\,dx = 1$$

In [10]:

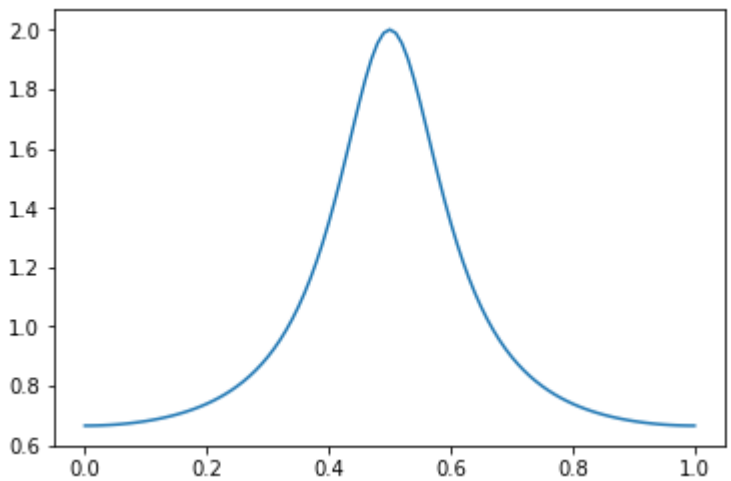
```
def f(x): return 0.5 + 3/(8*np.cos(2*np.pi*x) + 10)

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

In [11]:

```
x = np.linspace(0,1,101)
fig, ax = plt.subplots()
ax.plot(x,f(x))
```

Out[11]:

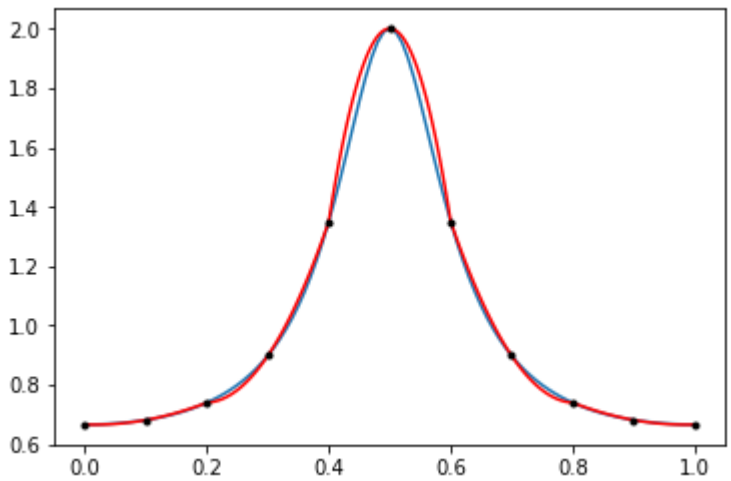
```
[<matplotlib.lines.Line2D at 0x7fb0f8f49f28>]
```



In [12]:

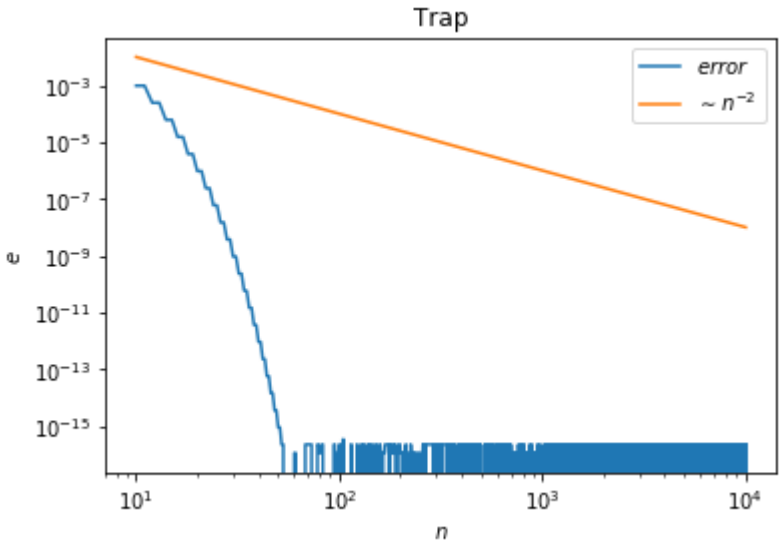
```
n = 5
for k in range(n):
    x = np.linspace(k/n,(k+1)/n,101)
    ax.plot(x,q(x),color='red')
points = np.linspace(0,1,2*n+1)
ax.plot(points,f(points),'k.')
fig
```

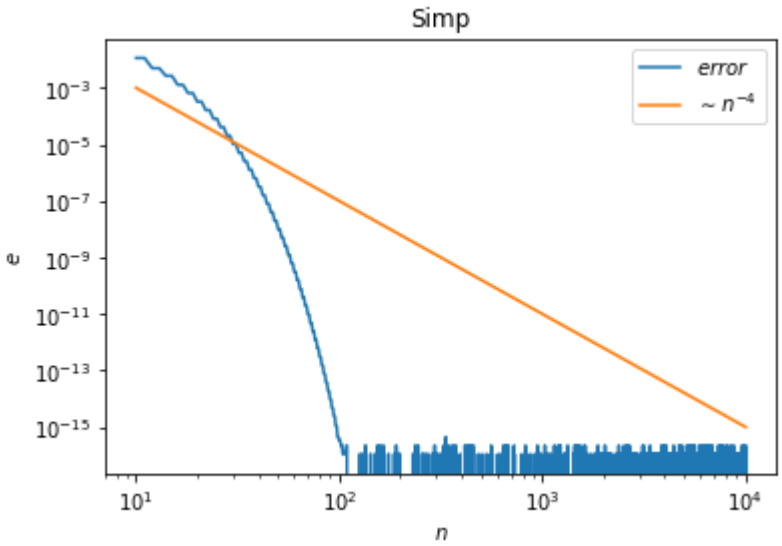
Out[12]:



In [13]:

```
(fT, aT), (fS, aS) = errors(f,1)
```





~5 min break!

List slices

In the composite Simpson's rule, we want to treat the endpoints, even indices, and odd indices differently. How can we extract these things efficiently? By **slicing**!

When you see something like `[a:b:c]` at the end of a list (or array, or any iterable really) it's called a `slice`, (see here (https://stackoverflow.com/questions/509211/understanding-pythons-slice-notation)) and takes the form:

```
[start:stop:step]
```

It takes all the elements from index `start`, up to and not including index `stop`, in steps of `step`. Some important points:

- A negative `stop` counts backwards from the end position.
- A negative `step` takes steps backward.
- Omitting `start` will begin from 0.
- Omitting `end` will continue until the list ends.
- Omitting `step` will take a step of 1.

Test it out on the list `[0,1,2,3,4,5]` if you're not sure.

In [14]:

```
x = list(range(6))
x
```

Out[14]:

```
[0, 1, 2, 3, 4, 5]
```

In [15]:

```
x[0:1]
```

Out[15]:

```
[0]
```

Questions

Here are some things to think about. You should understand how to figure these out. These are going to be central to the next lecture.

- Why does the trapezoidal rule give second order convergence normally?
- Why did it get *exponentially* accurate for periodic functions?
- How can you prove these similar behaviours for Simpson's rule?
- Can you extend the `Q` function to make a function that returns a composite Simpson's approximation for a given `f`

Quiz

Now for *two* quiz questions!

Approximating polynomials

An obvious, and important, `class` of functions to consider is the class of powers of x:

$$f(x) = x^p, \quad p > -1.$$

where

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

Taking the hint from above, and recalling our object oriented programming from previous lectures, using a `class` is a good way to define functions with a parameter. The `class` consists of all functions of a given form. An `object` is a particular one of those functions.

In [16]:

```
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)
```

There are some convenient methods to make nice plots.

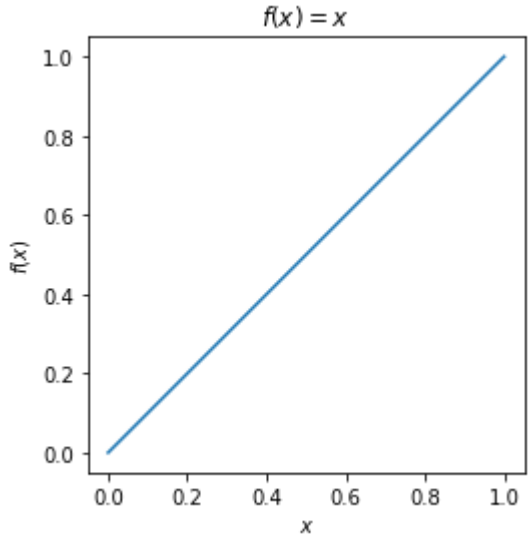
Let's use them!

Example 1: Line

Well if we can't do this properly, we're in trouble

In [17]:

```
f = power(1)
x = np.linspace(0,1,100)

fig, ax = f.show()
```

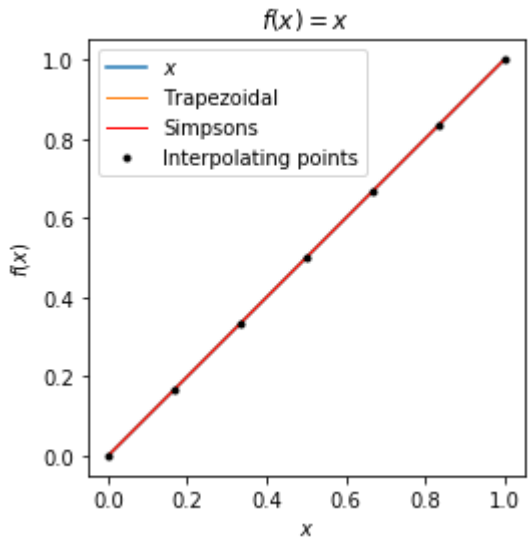


Because we're returning the initial plot from `self.show`, we can add things to it later:

In [18]:

```
ax.plot(points,f(points),'C1',label='Trapezoidal',linewidth=1)
n = 3
for k in range(n):
    x = np.linspace(k/n,(k+1)/n,101)
    ax.plot(x,f.q(x),color='red',linewidth=1)
points = np.linspace(0,1,2*n+1)
ax.plot(points,f(points),'k.',label='Interpolating points')

ax.lines[-2].set_label('Simpsons') # Why am I doing this?
ax.legend()
fig
```

Out[18]:

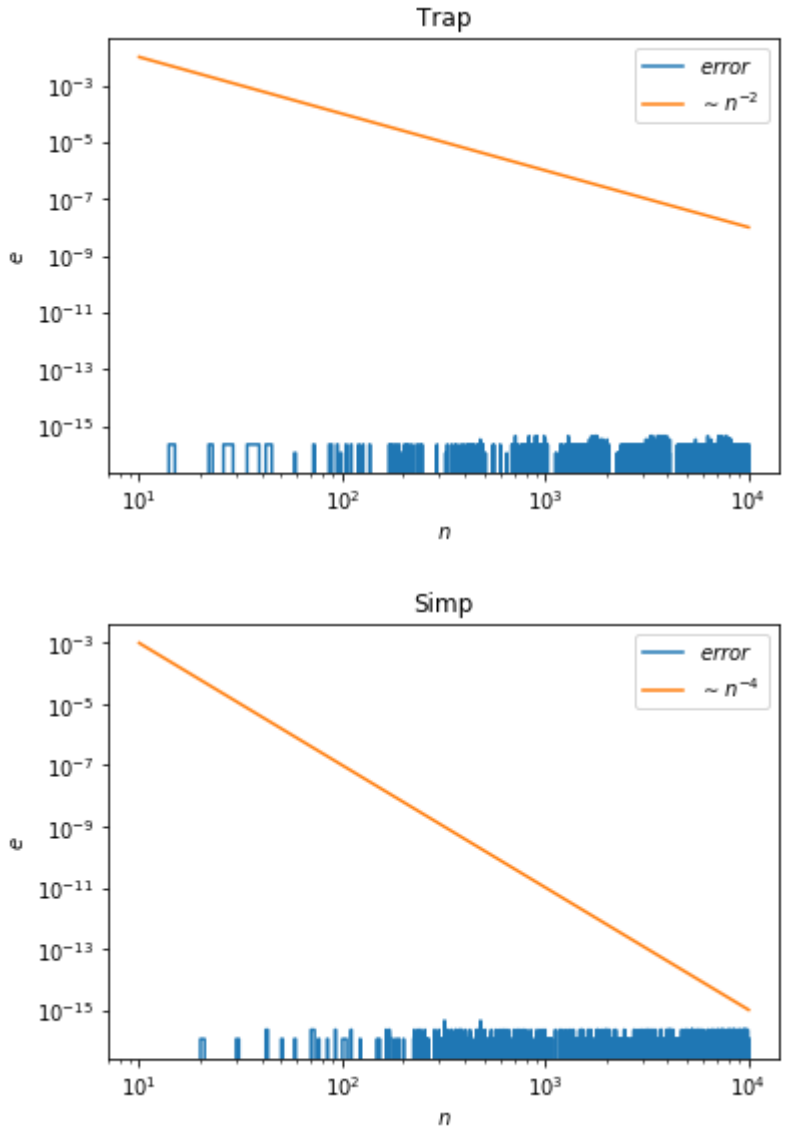


In [19]:

```
errors(f,f.area())
```

Out[19]:

```
((<Figure size 432x288 with 1 Axes>,
  <matplotlib.axes._subplots.AxesSubplot at 0x7fb0f89037b8>),
 (<Figure size 432x288 with 1 Axes>,
  <matplotlib.axes._subplots.AxesSubplot at 0x7fb0f89a1198>))
```



Ok, well that was a bit silly - of course a line can fit a line, and a parabola can too (it's degenerate now). But we know that our functions are behaving correctly.

What about a parabola?

Example 2: A parabola

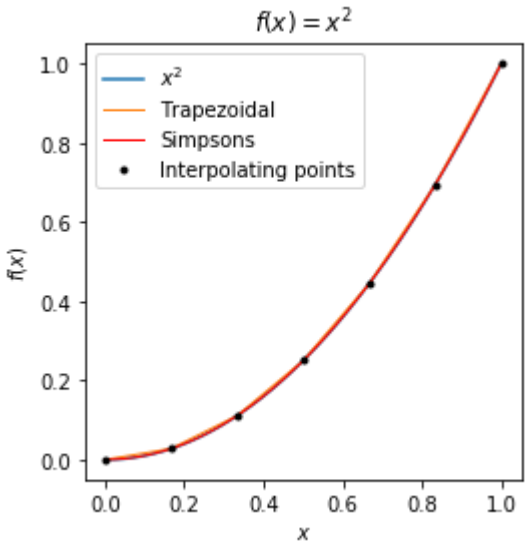
This defines $f(x) = x^2$.

In [20]:

```
f = power(2)

fig, ax = f.show()

ax.plot(points,f(points),'C1',label='Trapezoidal',linewidth=1)
n = 3
for k in range(n):
    x = np.linspace(k/n,(k+1)/n,101)
    ax.plot(x,f.q(x),color='red',linewidth=1)
points = np.linspace(0,1,2*n+1)
ax.plot(points,f(points),'k.',label='Interpolating points')

ax.lines[-2].set_label('Simpsons') # Why am I doing this?
ax.legend()
```

Out[20]:

```
<matplotlib.legend.Legend at 0x7fb0f8748438>
```

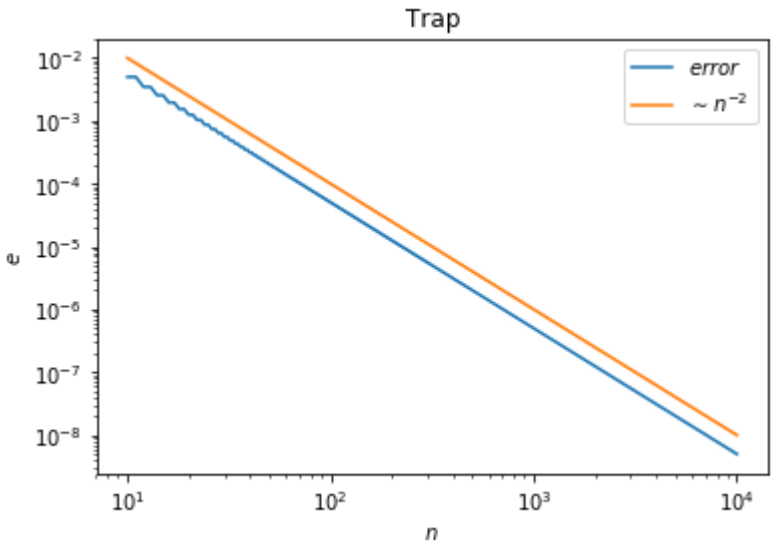


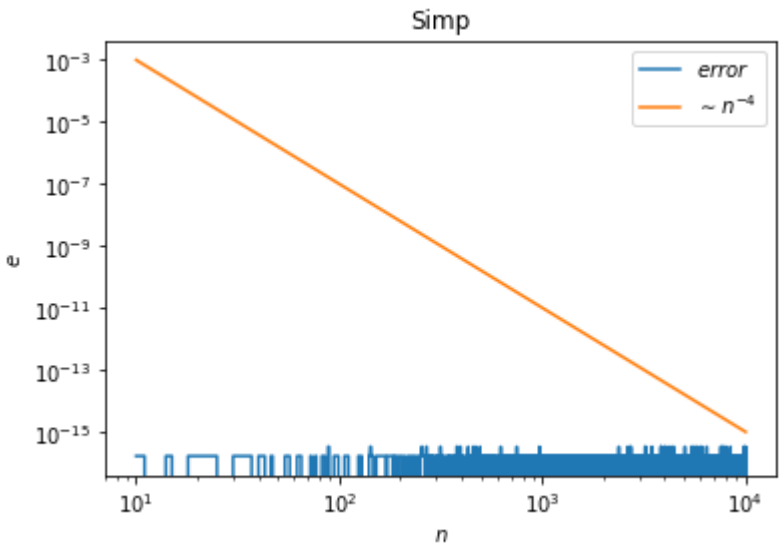
In [21]:

```
errors(f,f.area())
```

Out[21]:

```
((<Figure size 432x288 with 1 Axes>,
  <matplotlib.axes._subplots.AxesSubplot at 0x7fb0f8721940>),
 (<Figure size 432x288 with 1 Axes>,
  <matplotlib.axes._subplots.AxesSubplot at 0x7fb0f869a0f0>))
```





Well, that was boring. No surprise that a quadratic function can exactly approximate a parabola.

The trapezoidal rule is back to our guaranteed second order convergence.

Example 3: A cubic

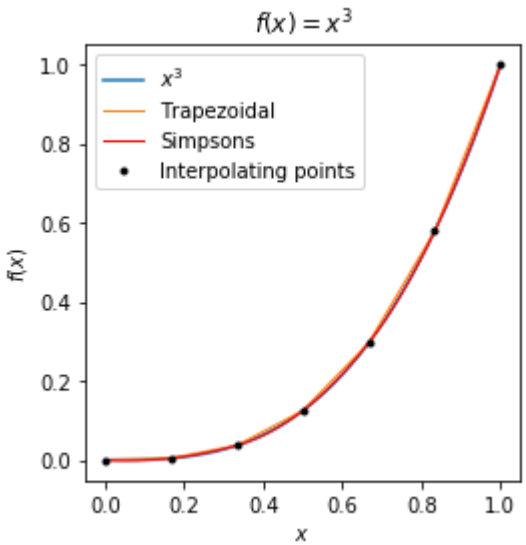
Well, I guess we know what's going to happen for x^3.

In [22]:

```
f = power(3)

fig, ax = f.show()

ax.plot(points,f(points),'C1',label='Trapezoidal',linewidth=1)
n = 3
for k in range(n):
    x = np.linspace(k/n,(k+1)/n,101)
    ax.plot(x,f.q(x),color='red',linewidth=1)
points = np.linspace(0,1,2*n+1)
ax.plot(points,f(points),'k.',label='Interpolating points')

ax.lines[-2].set_label('Simpsons') # Why am I doing this?
ax.legend()
```

Out[22]:

```
<matplotlib.legend.Legend at 0x7fb0f83caeb8>
```

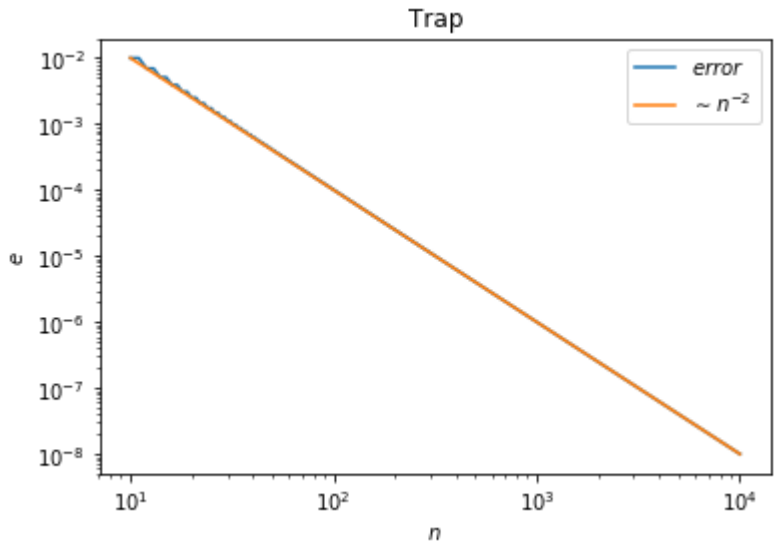


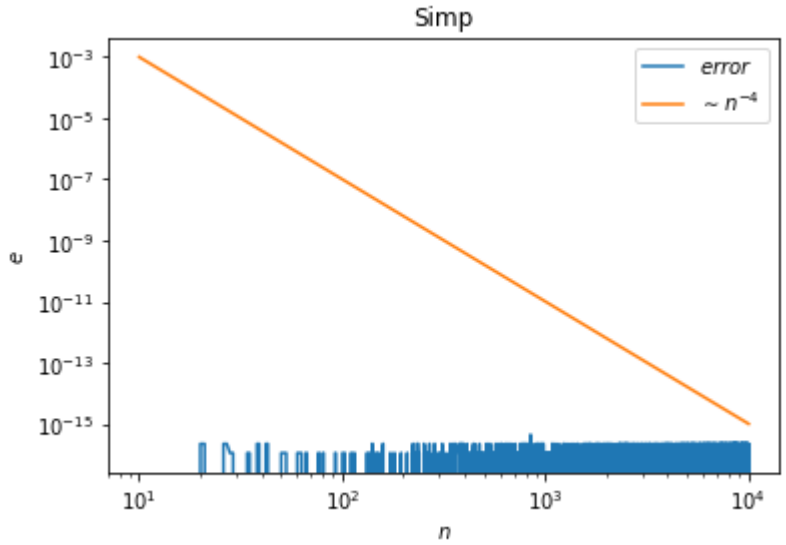
In [23]:

```
errors(f,f.area())
```

Out[23]:

```
((<Figure size 432x288 with 1 Axes>,
  <matplotlib.axes._subplots.AxesSubplot at 0x7fb0f838d550>),
 (<Figure size 432x288 with 1 Axes>,
  <matplotlib.axes._subplots.AxesSubplot at 0x7fb0f8361b70>))
```





Eh?

Even though a parabola doesn't exactly approximate a cubic - Simpson's method gives the exact answer for the integral!

Why?!

Example 4: A quartic

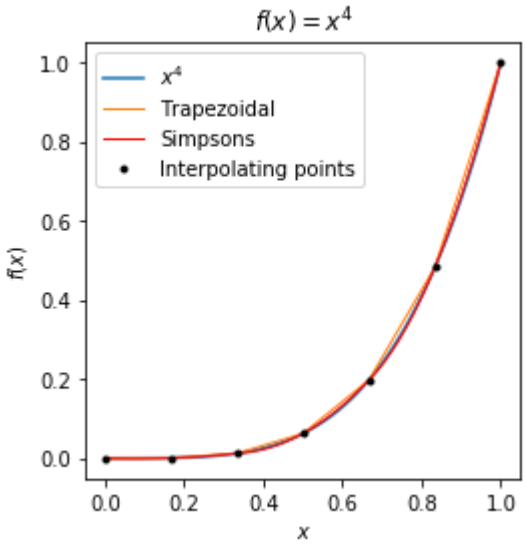
Well, what about x^4 ?

In [24]:

```
f = power(4)
fig, ax = f.show()

ax.plot(points,f(points),'C1',label='Trapezoidal',linewidth=1)
n = 3
for k in range(n):
    x = np.linspace(k/n,(k+1)/n,101)
    ax.plot(x,f.q(x),color='red',linewidth=1)
points = np.linspace(0,1,2*n+1)
ax.plot(points,f(points),'k.',label='Interpolating points')

ax.lines[-2].set_label('Simpsons') # Why am I doing this?
ax.legend()
```

Out[24]:

```
<matplotlib.legend.Legend at 0x7fb0f802fb00>
```

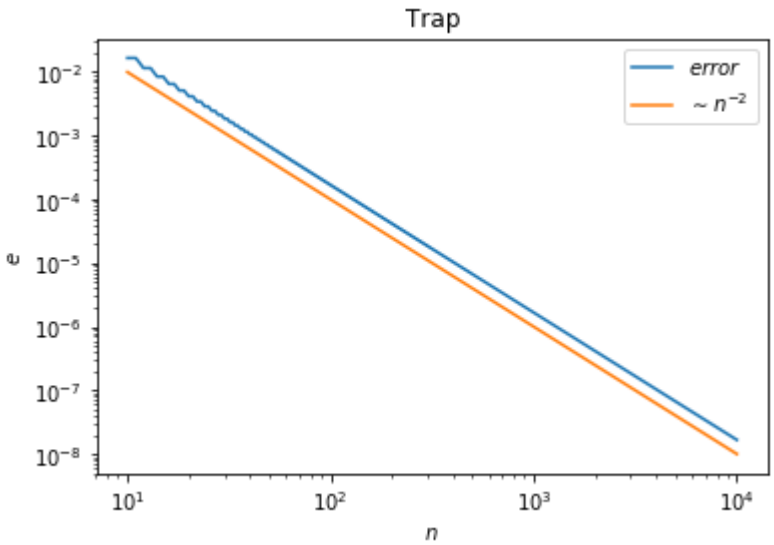


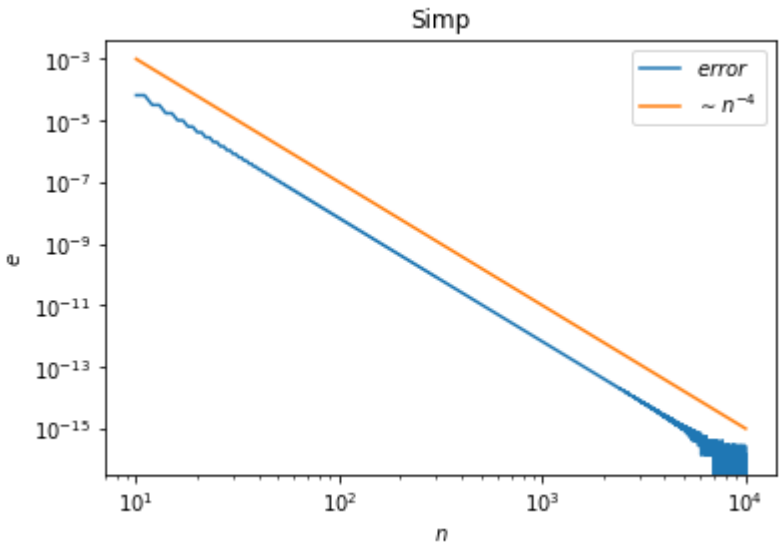
In [25]:

```
errors(f,f.area())
```

Out[25]:

```
((<Figure size 432x288 with 1 Axes>,
  <matplotlib.axes._subplots.AxesSubplot at 0x7fb0f805b080>),
 (<Figure size 432x288 with 1 Axes>,
  <matplotlib.axes._subplots.AxesSubplot at 0x7fb0f7fcda58>))
```





Ok, that's a bit more expected. Our expected second order trapezoidal rule, and fourth order Simpson's rule.

But is the salesman genuine? Are we *guaranteed* at least second and fourth order convergence?

Example 1/2: A square root

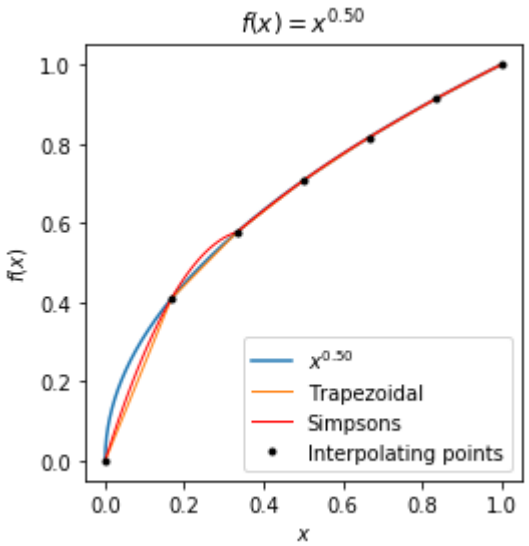
Let's look at $f(x) = \sqrt{x}$. It's a simple enough function to want to approximate.

In [26]:

```
f = power(1/2)
fig, ax = f.show()

ax.plot(points,f(points),'C1',label='Trapezoidal',linewidth=1)
n = 3
for k in range(n):
    x = np.linspace(k/n,(k+1)/n,101)
    ax.plot(x,f.q(x),color='red',linewidth=1)
points = np.linspace(0,1,2*n+1)
ax.plot(points,f(points),'k.',label='Interpolating points')

ax.lines[-2].set_label('Simpsons') # Why am I doing this?
ax.legend()
```

Out[26]:

```
<matplotlib.legend.Legend at 0x7fb0f7cef4e0>
```



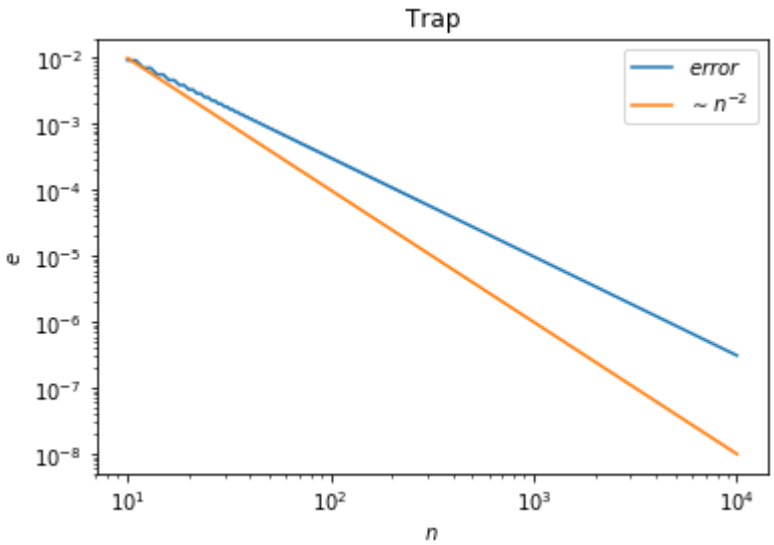
Hmm, doesn't look great around $x = 0$...

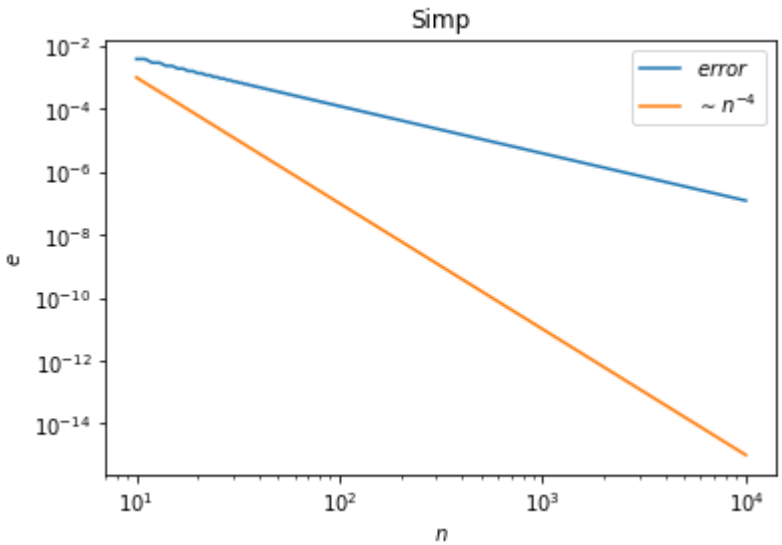
In [27]:

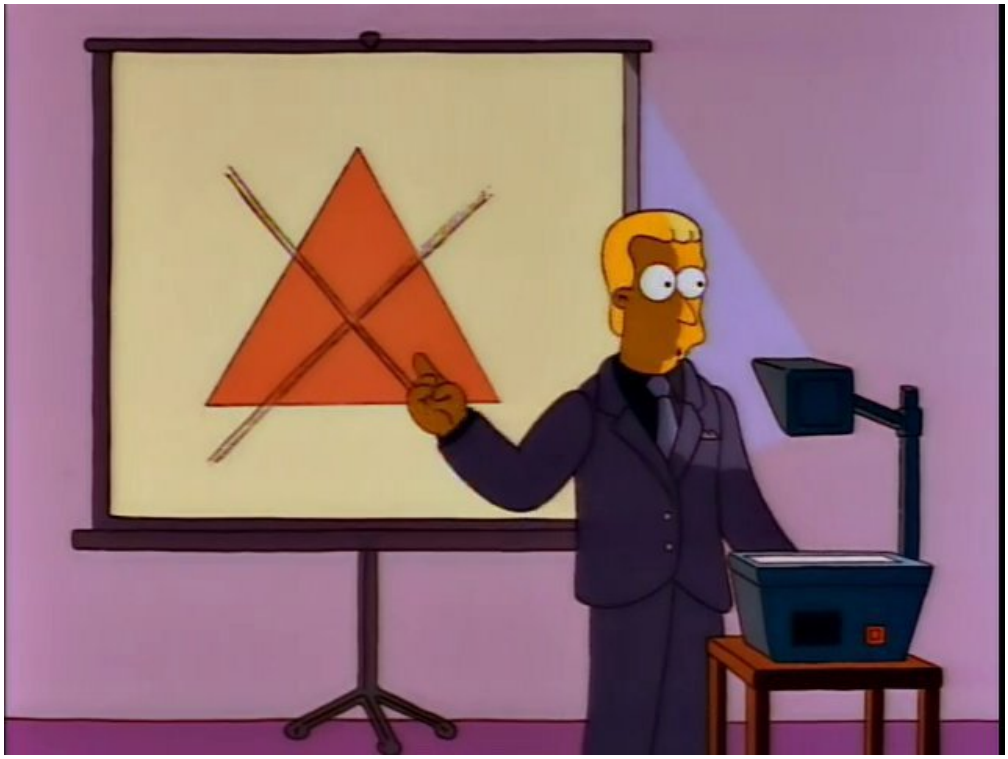
```
errors(f,f.area())
```

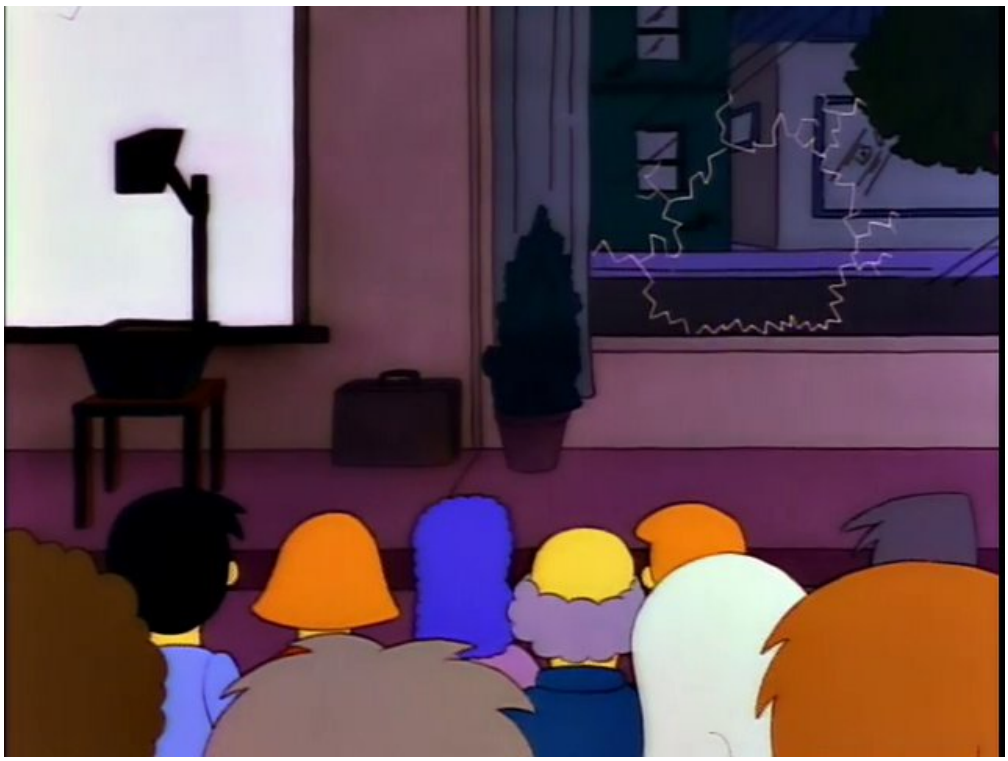
Out[27]:

```
((<Figure size 432x288 with 1 Axes>,
  <matplotlib.axes._subplots.AxesSubplot at 0x7fb0f7cc4278>),
 (<Figure size 432x288 with 1 Axes>,
  <matplotlib.axes._subplots.AxesSubplot at 0x7fb0f7c806d8>))
```









He lied!

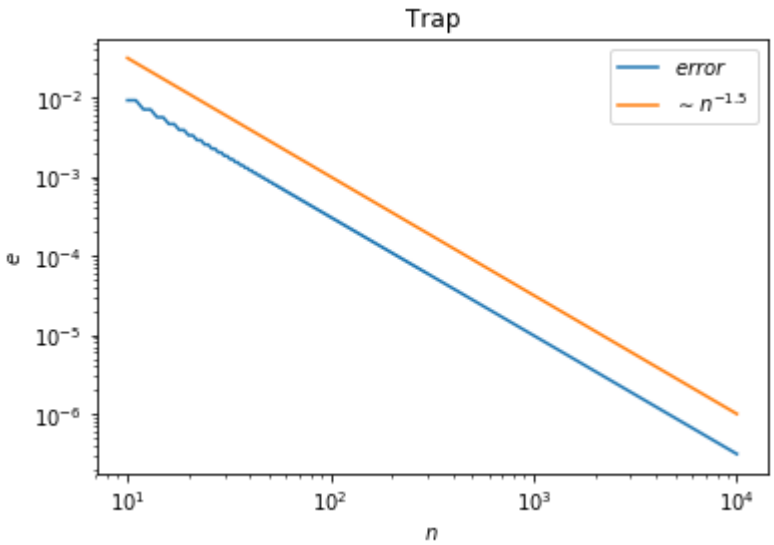
Well, what convergence *are* we getting?

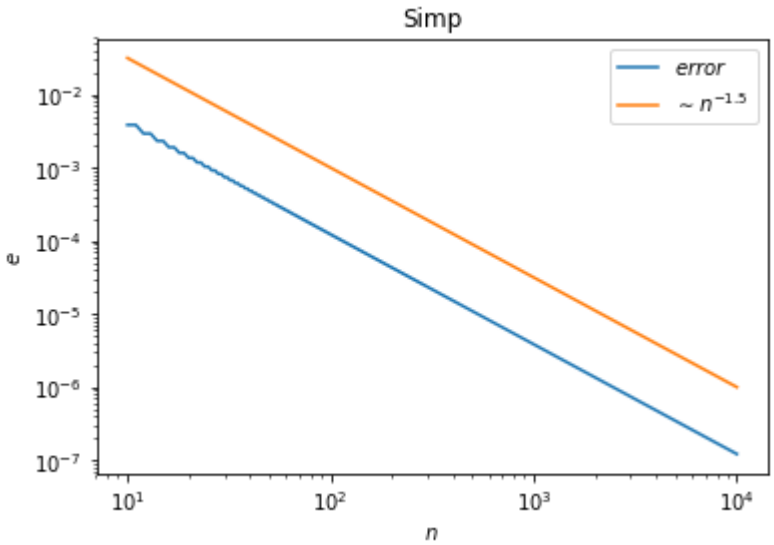
In [28]:

```
errors(f,f.area(),pT=-1.5,cT=1,pS=-1.5,cS=1)
```

Out[28]:

```
((<Figure size 432x288 with 1 Axes>,
  <matplotlib.axes._subplots.AxesSubplot at 0x7fb0f7ddf320>),
 (<Figure size 432x288 with 1 Axes>,
  <matplotlib.axes._subplots.AxesSubplot at 0x7fb0f82227b8>))
```





Why are we only getting $n^{-1.5}$ convergence!

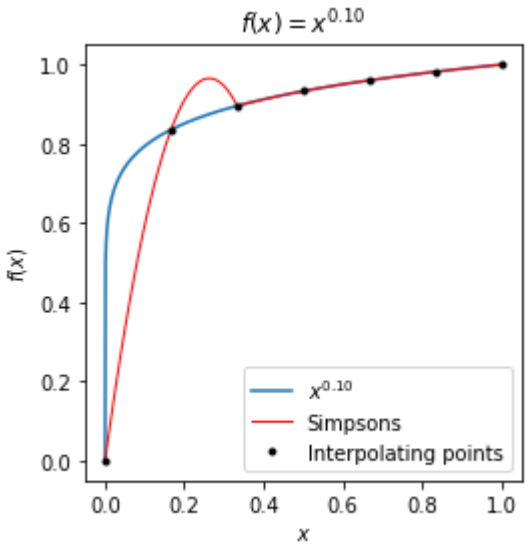
Example 1/10: A tenth root?

In [29]:

```
f = power(1/10)
x = np.linspace(0,1,100)

fig, ax = f.show()

n = 3
for k in range(n):
    x = np.linspace(k/n,(k+1)/n,101)
    ax.plot(x,f.q(x),color='red',linewidth=1)
points = np.linspace(0,1,2*n+1)
ax.plot(points,f(points),'k.',label='Interpolating points')

ax.lines[-2].set_label('Simpsons') # Why am I doing this?
ax.legend()
```

Out[29]:

```
<matplotlib.legend.Legend at 0x7fb0f75dba58>
```



In [30]:

```
errors(f,f.area(),pT=-(1+p),cT=1,pS=-(1+p),cS=1)
```

```
---------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-30-4e594ca8cdfd> in <module>()
----> 1 errors(f,f.area(),pT=-(1+p),cT=1,pS=-(1+p),cS=1)

NameError: name 'p' is not defined
```

Well there's definitely a pattern going on here.

Now, this is a mathematical computing course - so we need to be understand how to examine these things theoretically too.

Homework: can you figure it out for tomorrow?