Lecture 10: Integration + Quadrature = Sums

After exploring a lot of basics concepts surrounding numbers, iteration, and programming, it's time to get into some applied numerical analysis.

The theme for the course will always remain the same:

```
Code: large amounts of abstraction + healthy mixtures of procedural, funct
ional, and object-oriented

Math: iteration, recursion + linear operations on data.
```

As we go forward, it might feel like things are getting complicated at times. But this is not the case.

```
Math is difficult, but it's not complicated.
```

For example, maybe you've seen come "complicated" looking expressions such as

$$\int_D \nabla \cdot u \, \mathrm{d}V = \int_{\partial D} u \cdot \mathrm{d}A, \qquad \int_A (\nabla \times u) \cdot \mathrm{d}A = \int_{\partial A} u \cdot \mathrm{d}\ell$$

Ie, Guass divergence theorem, and/or Stokes theorem?

Maybe it's occurred to you that these look a lot like the **Fundamental Theorem of Calculus:**

$$\int_a^b f'(x)\,\mathrm{d}x = f(b) - f(a), \qquad \frac{d}{dx}\int_a^x f(x)\,\mathrm{d}x = f(x)$$

They are the same thing:

```
Just iterating with the number of dimensions.
```

All of these expressions relate the integral of a function in some dimension to the integral of it's derivative in one higher dimension.

And I've got news for you: It gets simpler.

In fact the integral sign was originally designed to look like a fancy letter

$$S \longleftrightarrow \int$$

Leibniz invented it :

This is true if you are thinking about Riemann integration (sums), or you are fancy and prefer one of the luxury integration formulations; Lebesgue, Stieltjes or even some more exotic brands.

No matter how you shake it. At the bottom level, all integration theorems are re-packaged versions of the following simple fact:

$$s_n - s_0 \;=\; \sum_{k=1}^{n} a_k \qquad \longleftrightarrow \qquad a_n \;=\; s_n - s_{n-1}$$

And while were are at it, let's face the fact that these fancy sums are just high-priced versions of

$$s_1 - s_0 = a_2\,, \quad s_2 - s_0 = a_1 + a_2 \qquad \longleftrightarrow \qquad a_1 \;=\; s_2 - s_1\,, \quad a_2 \;=\; s_1 - s_0$$

Getting from this to the Gauss divergence theorem just requires keeping track of about ten simple facts; difficult, but not complicated.

All the fancy theorems you've ever seen are clever reshufflings of arithmetic.

The fortunate thing, is that if you can reckon a problem out enough to know how to sum it, then you can get a computer to keep track of a huge amount of the information for you.

INTEGRATION BY PARTS

You might also recall things like

$$\int_D \phi\, \nabla \cdot u \,\mathrm{d}V \;=\; \int_{\partial D} \phi\, u \cdot \hat{n} \,\mathrm{d}A - \int_D u \cdot \nabla \phi \,\mathrm{d}V$$

Or

$$\int_a^b f(x)\, g'(x)\,\mathrm{d}x \;=\; f(b)\, g(b) - f(a)\, g(a) - \int_a^b f'(x)\, g(x)\,\mathrm{d}x$$

Guess what?

$$\sum_{k=1}^{n} u_k(v_k - v_{k-1}) \;=\; u_{n+1}v_n - u_1 v_0 - \sum_{k=1}^{n}(u_{k+1} - u_k)v_k$$

Trapezoidal rule

For a function, $f(x)$ defined on $a \le x \le b$, define the **linear interpolation**

$$L(x) \;=\; f(b)\,\frac{x-a}{b-a} \;+\; f(a)\,\frac{b-x}{b-a}$$

This has the property that

$$L(a) \;=\; f(a), \quad L(b) \;=\; f(b), \quad L'(x) \;=\; \frac{f(b)-f(a)}{b-a}, \quad L''(x) \;=\; 0$$

Also

$$\int_a^b L(x)\,\mathrm{d}x \;=\; \frac{b-a}{2}\,(f(b) \;+\; f(a)\,)$$

Let's define a function to work with

$$f(x) \;=\; \frac{x}{\sqrt{2}} + \frac{1}{5}\sin(\,9\,x\,), \quad \text{for} \quad 0 \le x \le 1$$

It's simple to compute the integral of this exactly over any interval.

In [1]:

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

def f(x,integral=False):
    if integral:
        i  = x[-1]*x[-1]/(2*np.sqrt(2)) - np.cos(9*x[-1])/45
        i -= x[0]*x[0]/(2*np.sqrt(2)) - np.cos(9*x[0])/45
        return i
    return np.sqrt(0.5)*x + 0.2*np.sin(9*x)
```

Make a **helper function** to plot more easily

In [2]:

```
def show(x,y):
    fig, ax = plt.subplots()
    ax.plot(x,y)
    ax.set(aspect=1,xlabel='x',ylabel='f(x)',title='f(x) vs piecewise-linear int
erpolation')
    return ax
```

Make a domain $0 \le x \le 1$. And a "grid" that is on the same domain but with a different number of points.

In [3]:

```
x  = np.linspace(0,1,1000)

# A "grid" with fewer points.
def g(n): return np.linspace(x[0],x[-1],n+1)
```

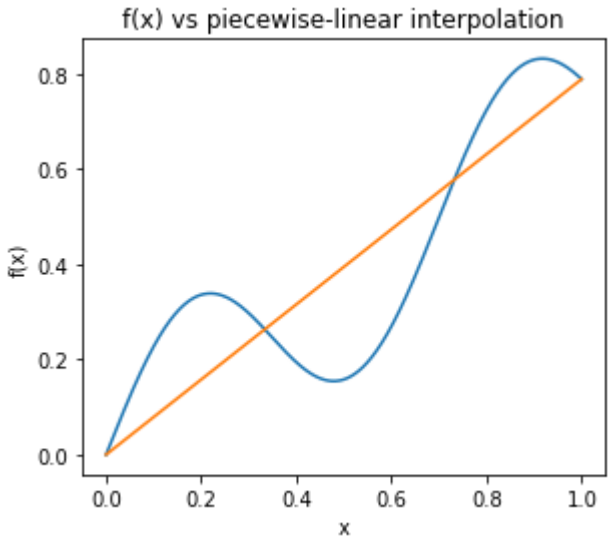
One segment:

In [4]:

```
n=1
ax = show(x,f(x))
ax.plot(g(n),f(g(n)))
```

Out[4]:

```
[<matplotlib.lines.Line2D at 0x7f54b01f86d8>]
```



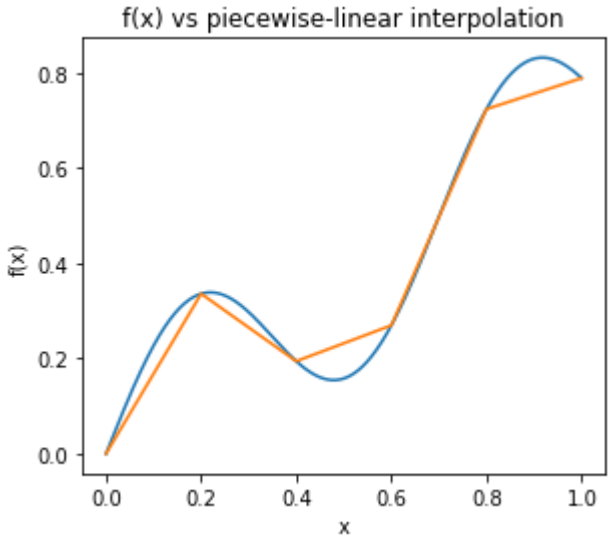
Two segments +

In [5]:

```
n=5
ax = show(x,f(x))
ax.plot(g(n),f(g(n)))
```

Out[5]:

```
[<matplotlib.lines.Line2D at 0x7f548f641c88>]
```



For

$$x_i \;=\; \frac{i}{n}\,(b-a)$$

$$I \;\equiv\; \int_a^b f(x)\,\mathrm{d}x \;\approx\; \sum_{i=1}^{n}\int_{x_{i-1}}^{x_i} L_i(x)\,\mathrm{d}x \;=\; \sum_{i=1}^{n}\frac{x_i - x_{i-1}}{2}\,\Big(\,f(x_i)+f(x_{i-1})\,\Big)$$

Where $x_i - x_{i-1} = (b-a)/n$.

Expanding things out

$$I \;\approx\; \frac{1}{2n}(\,f(x_1)+f(x_0)\,)+\frac{1}{2n}(\,f(x_2)+f(x_1)\,)+\ldots+\frac{1}{2n}\Big(\,f(x_{n-1})+f(x_{n-2})\,\Big)+\frac{1}{2n}\Big(\,f(x_n)+f(x_{n-1})$$

We can see that every point in the sum is counted twice, except the end points. Therefore

$$I \;\approx\; \frac{f(x_0)\;+\;f(x_n)}{2n}\;+\;\frac{1}{n}\sum_{i=1}^{n-1} f(x_i)$$

Let's code it up and see how it behaves:

In [6]:

```
def trap(f,x):

    a, b, mid = x[0], x[-1], x[1:-1]

    h = (b-a)/(len(x)-1)

    return h*(0.5*( f(a) + f(b) )  + np.sum(f(mid)) )
```

The same thing fits on one line. Even thought it's not as clear what's going on.

In [7]:

```
def trap(f,x): return ((x[-1]-x[0])/(len(x)-1))*(0.5*( f(x[0]) + f(x[-1]) )  + n
p.sum(f(x[1:-1])) )
```

Here is a helper function that compares two numbers

In [8]:

```
def error(A,B,report=True,n=''):
    e = abs(A-B)/abs(B)
    if report : print('{:>2s}  {:>6.3f}   {:>5.3f}  {:>9.4e}'.format(str(n),A,B,
e))
    return e
```

In [9]:

```
nmin, nmax = 1, 30

S = f(x,integral=True)

print("'exact' integral value: S = ",S)
print('')

print(' n     T       S     |T-S|/|S| ')
print('--   -----   -----  -----------')
for n in range(nmin,nmax):
    T=trap(f,g(n))
    error(T,S,n=n)
```

```
'exact' integral value: S =  0.39602295196848875

 n     T       S     |T-S|/|S|
--   -----   -----  -----------
 1   0.395   0.396  3.1759e-03
 2   0.276   0.396  3.0204e-01
 3   0.358   0.396  9.5833e-02
 4   0.376   0.396  4.9586e-02
 5   0.384   0.396  3.0650e-02
 6   0.388   0.396  2.0904e-02
 7   0.390   0.396  1.5197e-02
 8   0.391   0.396  1.1556e-02
 9   0.392   0.396  9.0893e-03
10   0.393   0.396  7.3384e-03
11   0.394   0.396  6.0502e-03
12   0.394   0.396  5.0746e-03
13   0.394   0.396  4.3179e-03
14   0.395   0.396  3.7189e-03
15   0.395   0.396  3.2367e-03
16   0.395   0.396  2.8426e-03
17   0.395   0.396  2.5165e-03
18   0.395   0.396  2.2435e-03
19   0.395   0.396  2.0127e-03
20   0.395   0.396  1.8158e-03
21   0.395   0.396  1.6465e-03
22   0.395   0.396  1.4998e-03
23   0.395   0.396  1.3719e-03
24   0.396   0.396  1.2597e-03
25   0.396   0.396  1.1607e-03
26   0.396   0.396  1.0730e-03
27   0.396   0.396  9.9481e-04
28   0.396   0.396  9.2490e-04
29   0.396   0.396  8.6211e-04
```

Let's run it for longer and make a plot of the error

In [10]:

```
S = f(x,integral=True)

nmin, nmax = 10, 10000
e = np.zeros(nmax-nmin+1)
n = np.arange(nmin,nmax+1)

for i in range(len(n)):
    T=trap(f,g(n[i]))
    e[i] = error(T,S,report=False)
```

We how to see the error decreases as some power of n; i.e.,

$$e \sim C_0 n^{-p}$$

The exponent, p, is called the ***order*** of convergence.

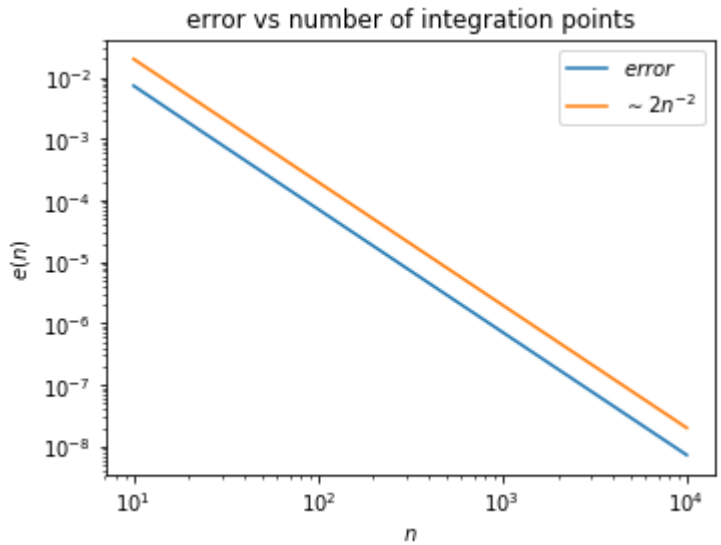
A `log-log` plot is a great way to determin the order of convergence. This is beacuse

$$\log e = -p \log n + \log C_0$$

Power laws become linear on a `log-log` scale. The slope of the line gives the exponent, the offset gives C_0.

In [11]:

```
fig, ax = plt.subplots()
ax.loglog(n,e)
ax.loglog(n,2/n**2)
ax.legend(['$error$', '$\sim  2n^{-2}$'])

ax.set(aspect=1/3,xlabel='$n$',ylabel='$e(n)$',title='error vs number of integra
tion points');
```



We plotted $2n^{-2}$ versus the error. This was becuase we wanted something simple to show the slope, and to keep it out of the way of the actual plot. But we can see that the plots are shifted with respect to each other. How can we find the constant prefactor?

We can make a ***compensated*** plot. This means that we have some error. $e(n)$. And we have a theory that

$$e(n) \approx C_0 n^{-2}$$

.

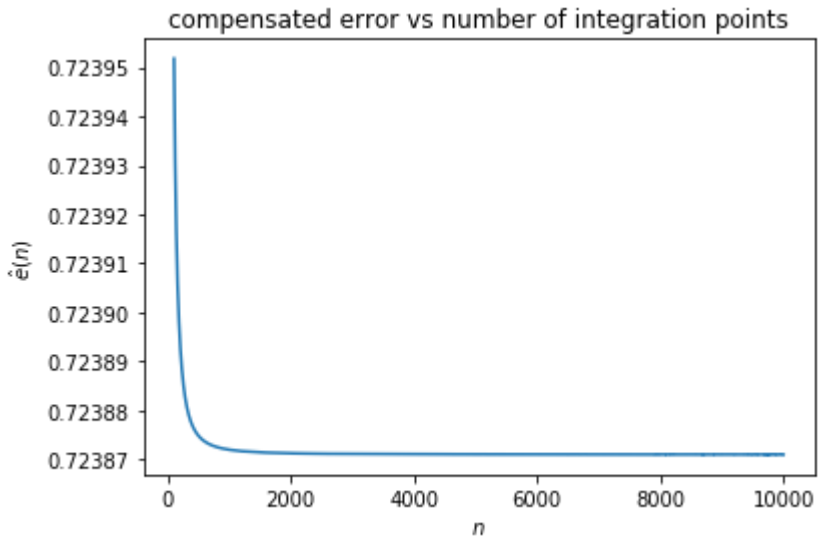
But we don't know C_0. Therefore, we can plot

$$\hat{e}(n) = e(n) \times n^2 \approx C_0.$$

This should be close to constant if our suspicions are correct.

In [12]:

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

i = 100
ax.plot(n[i:],e[i:]*n[i:]**2)

ax.set(xlabel='$n$',ylabel='$\hat{e}(n)$',title='compensated error vs number of
 integration points');
```



We can see that the beahviou is very well described as

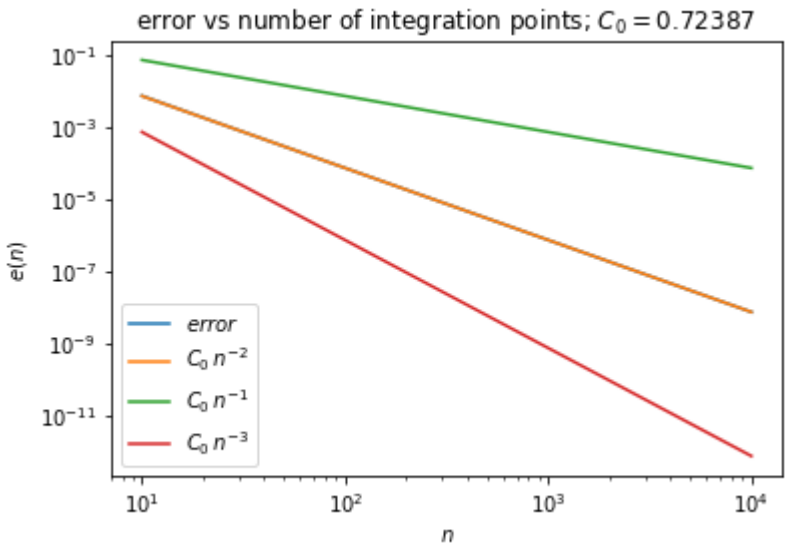
$$e(n) \approx 0.72387\, n^{-2}$$

The trapezoidal rule typically shows ***quadratic convergence*** .

To make sure, it's good to see if the result is *inconsistent* with other power laws:

In [13]:

```
fig, ax = plt.subplots()
ax.loglog(n,e)
ax.loglog(n,0.72387/n**2)
ax.loglog(n,0.72387/n**1)
ax.loglog(n,0.72387/n**3)
ax.legend(['$error$', '$C_{0}\, n^{-2}$','$C_{0}\, n^{-1}$','$C_{0}\, n^{-3}$'])
ax.set(xlabel='$n$',ylabel='$e(n)$',title='error vs number of integration point
s; $C_{0}=0.72387$');
```

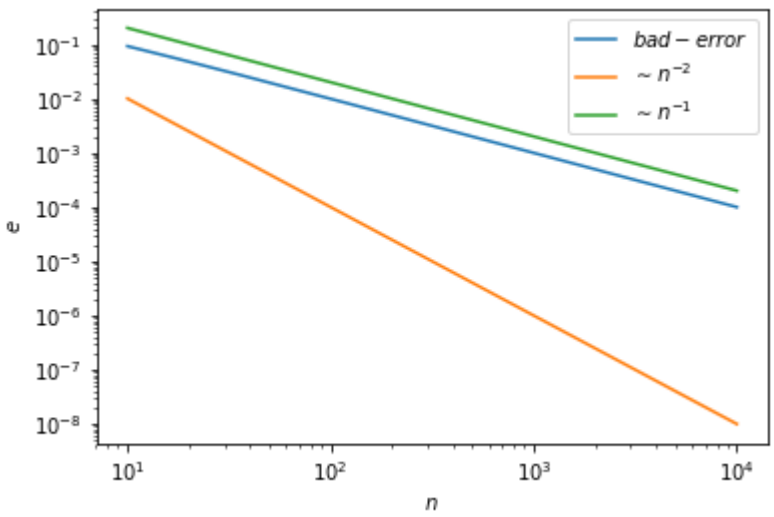


We can't see the blue line becuase it's completly under the orange line.

What about the end points?: We can see that those fussy factors of $1/2$ for the end values are important. The sum still converges, but at a much--reduced rate

In [14]:

```
def bad_trap(f,x):

    a, b = x[0], x[-1]

    h = (b-a)/(len(x)-1)

    return h*np.sum(f(x))

nmin, nmax = 10, 10000
bad_e = np.zeros(nmax-nmin+1)
n = np.arange(nmin,nmax+1)

for i in range(len(n)):
    T=bad_trap(f,g(n[i]))
    bad_e[i] = error(T,S,report=False)

fig, ax = plt.subplots()
ax.loglog(n,bad_e)
ax.loglog(n,1/n**2)
ax.loglog(n,2/n**1)
ax.legend(['$bad-error$', '$\sim n^{-2}$','$\sim n^{-1}$'])
ax.set_xlabel("$n$")
ax.set_ylabel("$e$");
```



Make sure to keep track of the end values!

Let's try it with another function:

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

In this case,

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

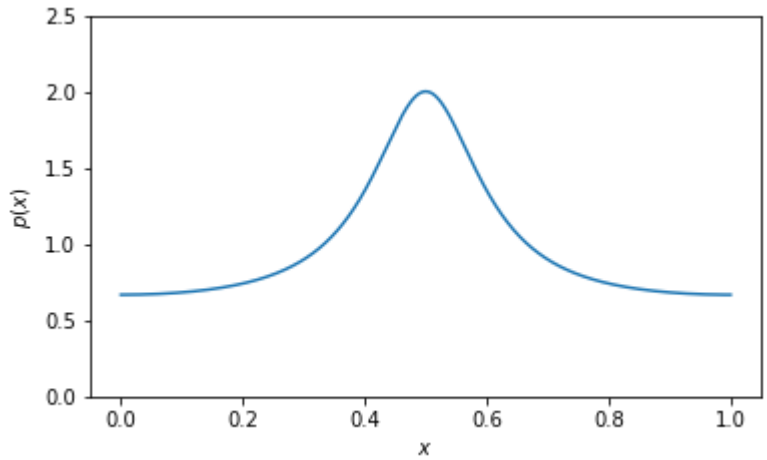
Here's what it looks like:

In [15]:

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

In [16]:

```
n = 0
ax = show(x,p(x))
ax.plot(g(n),p(g(n)))
ax.set(aspect=1/4,xlabel='$x$',ylabel='$p(x)$',title='',ylim=(0,2.5));
```



Let's make a table of the errors:

In [17]:

```
nmin, nmax = 1, 30

S = 1

print("'exact' integral value: S = ",S)
print('')

print(' n     T       S     |T-S|/|S| ')
print('--   -----   -----  -----------')
for n in range(nmin,nmax):
    T=trap(p,g(n))
    error(T,S,n=n)
```

```
'exact' integral value: S =  1

 n     T       S     |T-S|/|S|
--   -----   -----  -----------
 1   0.667   1.000  3.3333e-01
 2   1.333   1.000  3.3333e-01
 3   0.889   1.000  1.1111e-01
 4   1.067   1.000  6.6667e-02
 5   0.970   1.000  3.0303e-02
 6   1.016   1.000  1.5873e-02
 7   0.992   1.000  7.7519e-03
 8   1.004   1.000  3.9216e-03
 9   0.998   1.000  1.9493e-03
10   1.001   1.000  9.7752e-04
11   1.000   1.000  4.8804e-04
12   1.000   1.000  2.4420e-04
13   1.000   1.000  1.2206e-04
14   1.000   1.000  6.1039e-05
15   1.000   1.000  3.0517e-05
16   1.000   1.000  1.5259e-05
17   1.000   1.000  7.6293e-06
18   1.000   1.000  3.8147e-06
19   1.000   1.000  1.9073e-06
20   1.000   1.000  9.5368e-07
21   1.000   1.000  4.7684e-07
22   1.000   1.000  2.3842e-07
23   1.000   1.000  1.1921e-07
24   1.000   1.000  5.9605e-08
25   1.000   1.000  2.9802e-08
26   1.000   1.000  1.4901e-08
27   1.000   1.000  7.4506e-09
28   1.000   1.000  3.7253e-09
29   1.000   1.000  1.8626e-09
```

It looks like we are getting another *factor of two* of accuracy at each step!

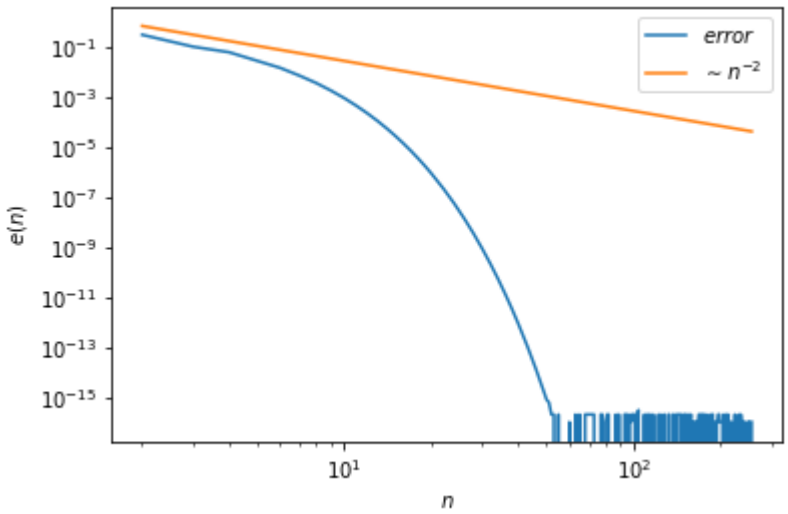
Let's look closer and see what's going on?

In [18]:

```
nmin, nmax = 2, 256

e = np.zeros(nmax-nmin+1)
n = np.arange(nmin,nmax+1)

for i in range(len(n)):
    T=trap(p,g(n[i]))
    e[i] = error(T,S,report=False)
```

In [19]:

```
fig, ax = plt.subplots()
ax.loglog(n,e)
ax.loglog(n,3/n**2)
ax.legend(['$error$', '$\sim n^{-2}$'])
ax.set_xlabel("$n$")
ax.set_ylabel("$e(n)$");
```



WOA! Something changed. It seems that we are getting an exponentially converging trapezoidal rule.

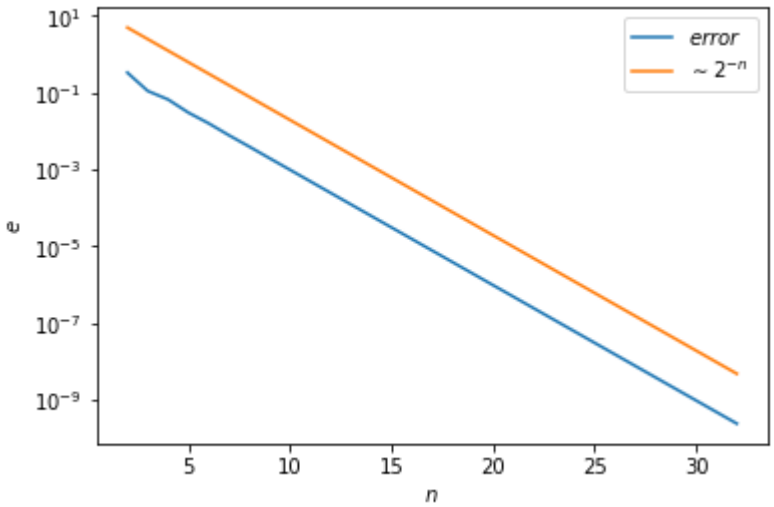
To find out, let's plot the error on a `semi-log` scale.

In [20]:

```
fig, ax = plt.subplots()
i=31
ax.semilogy(n[0:i],e[0:i])
ax.semilogy(n[0:i],20/2**n[0:i])
ax.legend(['$error$', '$\sim 2^{-n}$'])
ax.set_xlabel("$n$")
ax.set_ylabel("$e$")
```

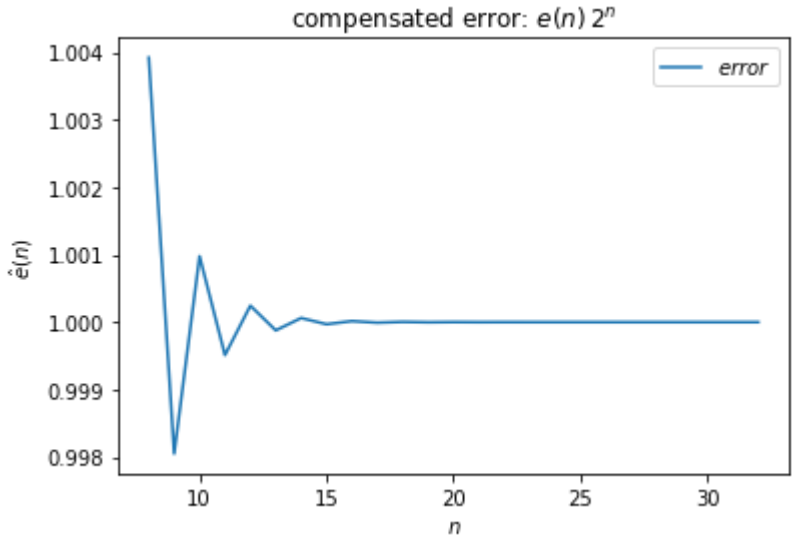
Out[20]:

```
Text(0,0.5,'$e$')
```



In [21]:

```
fig, ax = plt.subplots()
i,j=6,31
ax.plot(n[i:j],e[i:j]*2**n[i:j])
ax.legend(['$error$'])
ax.set(xlabel='$n$',ylabel='$\hat{e}(n)$',title='compensated error: $e(n) \, 2^{n}$');
```



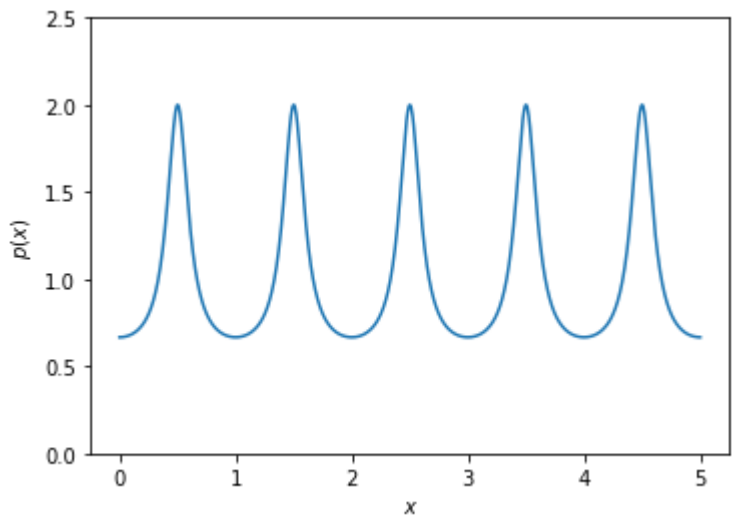
It is going down exponentially fast

$$e(n) \approx 2^{-n}.$$

The full reasons for this will become clear later. For now we can ask what is the difference between the two examples. One of these functions is half of a ***periodic*** function. The `cosine` is a bit of a giveaway. Why do you think I called it $p(x)$? The p stands for periodic.

In [22]:

```
ax = show(5*x,p(5*x))
ax.set(aspect=1.5,xlabel='$x$',ylabel='$p(x)$',title='',ylim=(0,2.5));
```



We can see that the function is periodic if we extend it 5-fold.

We'll se more than a few magical properties of periodic functions as the course progresses.