Lecture 15: Makes all the finite difference in the world

Recap

So, just like integral equations, differential equations are important to solve.
Often, if you can solve one, you can get the other (they are inverse problems).
So we want to numerically solve **ordinary differential equations**.
In fact, ODEs are more common than integral equations.

And, just like we wanted to approximate **integrals** with **sums**,
we want to approximate **derivatives** with **differences**.

By using ***Taylor*'s *theorem***, you can systematically derive **finite difference** approximations to derivatives of

- desired **order** (second, third derivatives...), and to
- desired **accuracy** (second order accurate, fourth order ...).

And we'll have to deal with more problems at boundaries again.

These will be the topics of today's lecture.

To the ends of the Earth. And no further

Last time we discussed the 2nd-order symmetric centred finite difference formula

$$f'(x) \;\approx\; \frac{f(x+h)-f(x-h)}{2h} \;-\; \frac{h^2}{6}f'''(x) \;+\; \mathcal{O}(h^3)$$

This worked well. But there is one small slight of hand that I made when testing it. ***I assumed I could evaluate the function at any point***. What did I do?

```
I made a function I could evaluate at any point.

I made a grid of x with spacing h

I evaluated the function at points x + h, and x - h.

This corresponded to the grid points i + 1, and i - 1.

This would be fine, *except for the end points*.

Nothing failed because I could evaluate the function anywhere.

This is not the case with real data. You don't have points beyond the end
s.
```

What if I don't know the function?

Let's make this more "computer" and see what I mean.

Suppose we have grid points.

$$x_i, \quad \text{for} \quad i = 0, 1, \ldots, n$$

We also have function values

$$y_i \; = f(x_i)$$

Suppose we know the y_i values, but ***we do not know the actual function***, $f(x)$. This kind of situation happens a lot. If we did know the function, then we could just differentiate it and this would all be very silly.

The whole point of finite differences is to ***estimate*** $f'(x_i)$ even if we only know the values y_i. When considered this way, the finite difference formula look a bit different,

$$f'(x_i) \; \approx \; \frac{y_{i+1} - y_{i-1}}{2h} \; + \; \mathcal{O}(h^2)$$

Perhaps we see the problem. The index $0 \leq i \leq n$. But the finite difference formula needs y_{-1} and y_{n+1} for the end-point calculations. In lecture 15, I just made a function that would compute these, even though I wouldn't normally be able to.

Let's try this for a few functions. For example:

$$f(x) = e^x, \quad f'(x) \; = \; e^x$$

This is a good example because we don't need the derivative, it's the same as the function. But we pretend we don't know that.

In [3]:

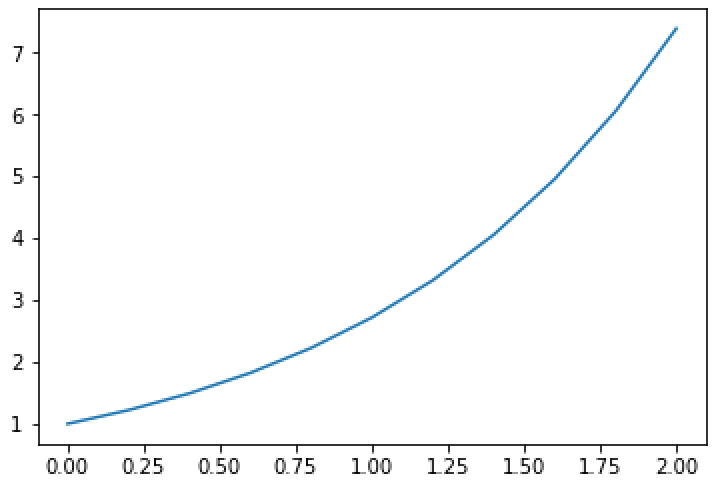
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

In [4]:

```
x  = np.linspace(0,2,11)
y  = np.exp(x)
fig, ax = plt.subplots()
ax.plot(x,y)
```

Out[4]:

```
[<matplotlib.lines.Line2D at 0x7f9a7af20b38>]
```



What we want to compute the finite difference is a ***shift*** operation

$$y_i \rightarrow y_{i \pm 1}$$

or more generally

$$y_i \rightarrow y_{i \pm k}$$

How do we do this?

http://lmgtfy.com/?q=How+do+shift+Python%3F (http://lmgtfy.com/?q=How+do+shift+Python%3F)

In [5]:

```
L = np.array([0,1,2,3,4,5])
print(L)
print(np.roll(L,1))
print(np.roll(L,-1))
```

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

We need to keep track of the direction of the `roll` function.

```
np.roll(L,-1)[i] gives us L[i+1]

np.roll(L,+1)[i] gives us L[i-1]
```

In [6]:

```
print(np.roll(L,-1)[3] == L[4])
print(np.roll(L,+1)[3] == L[2])
```

```
True
True
```

In [7]:

```
def D(y,x):

    Dy = (np.roll(y,-1) - np.roll(y,+1))/(np.roll(x,-1) - np.roll(x,+1))

    return Dy
```

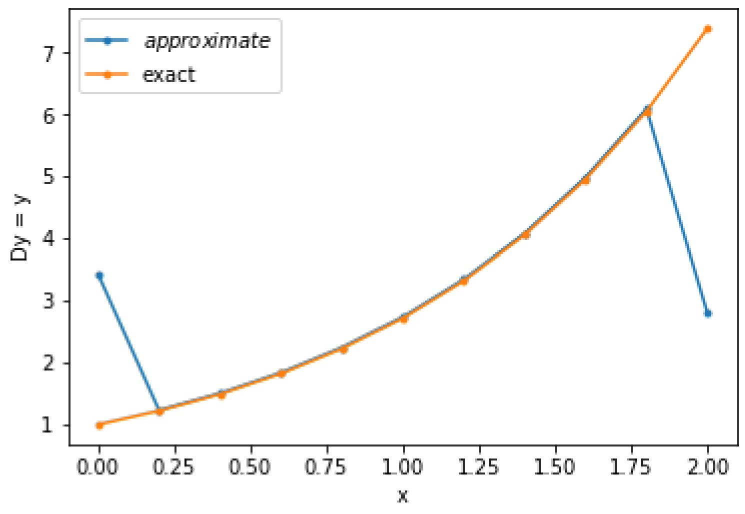
Let's look at the result

In [8]:

```
Dy = D(y,x)

fig, ax = plt.subplots()
ax.plot(x,Dy,marker='.')
ax.plot(x,y,marker='.')
ax.set_xlabel('x')
ax.set_ylabel('Dy = y')
ax.legend(['$approximate$', 'exact'])
```

Out[8]:

```
<matplotlib.legend.Legend at 0x7f9a7ae25a90>
```



Fixing boundary derivatives

Not surprising, the end points are wrong. The middle looks good. Let's see about fixing up the ends.

We also know from the definition of a derivative that

$$f'(x) \approx \frac{f(x+h)-f(x)}{h}, \quad \text{and} \quad f'(x) \approx \frac{f(x)-f(x-h)}{h}$$

These would work at the end points. While the previous formula was called ***centered***, these are called ***non-centered***, or *left*, and/or *right*.

Acting on the array,

$$Dy_0 = \frac{y_1 - y_0}{h}$$

$$Dy_n = \frac{y_n - y_{n-1}}{h}$$

Therefore

In [9]:

```
def D(y,x):
    """Finite difference given function values."""
    Dy = (np.roll(y,-1) - np.roll(y,+1))/(np.roll(x,-1) - np.roll(x,+1))
    # Fix end points
    Dy[0]  = (y[1] -  y[0]) /(x[1]  - x[0])
    Dy[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])
    return Dy
```

Let's try again:

In [10]:

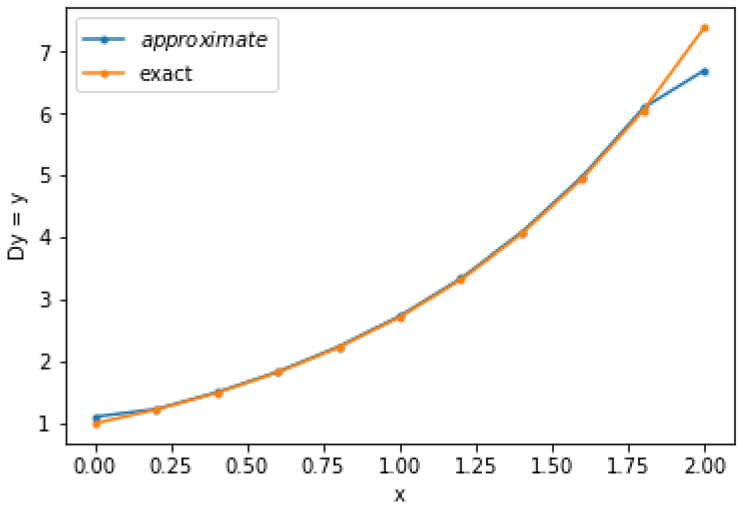
```
x  = np.linspace(0,2,11)
y  = np.exp(x)
Dy = D(y,x)
```

In [11]:

```
fig, ax = plt.subplots()
ax.plot(x,Dy,'.-')
ax.plot(x,y,'.-')
ax.set_xlabel('x')
ax.set_ylabel('Dy = y')
ax.legend(['$approximate$', 'exact'])
```

Out[11]:

```
<matplotlib.legend.Legend at 0x7f9a7ad9ecc0>
```



Interior and boundary errors

Much better. But it isn't doing great at the end. It gets better with more points, but why isn't it as good as the middle?

Let's go back to Taylor's Theorem to find out about the error.

$$f(x+h) \;=\; f(x) \;+\; hf'(x) \;+\; \frac{h^2}{2}f''(x) \;+\; \frac{h^3}{3!}f'''(x) \;+\; \dots$$

You can rearrange this to get

$$f'(x) \;=\; \frac{f(x+h)-f(x)}{h} \;-\; \frac{h}{2}f''(x) \;+\; \dots$$

The error is only $\mathcal{O}(h)$, not $\mathcal{O}(h^2)$. The low-order formula on the boundary spoils order of the scheme.

Error norms

Let's look at the error using different ℓ_p-norms.

In [12]:

```
def norm(x,p):
    a = np.abs(x)
    if p == 'inf': return np.max(a)
    return np.sum(a**p)**(1/p)
```

In [13]:

```
nmin, nmax = 10, 10000
n = np.array([2**i for i in range(int(np.log2(nmin)),int(np.log2(nmax))+1)])
```

In [14]:

```
error_one = np.zeros(len(n))
error_two = np.zeros(len(n))
error_inf = np.zeros(len(n))
```

In [15]:

```
for k in range(len(n)):

    x  = np.linspace(0,2,n[k])
    y  = np.exp(x)
    Dy = D(y,x)

    error_one[k] = norm(Dy-y,1)/norm(y,1) # sum of absolute values
    error_two[k] = norm(Dy-y,2)/norm(y,2) # sqrt of sum of squares
    error_inf[k] = norm(Dy-y,'inf')/norm(y,'inf') # max value
```

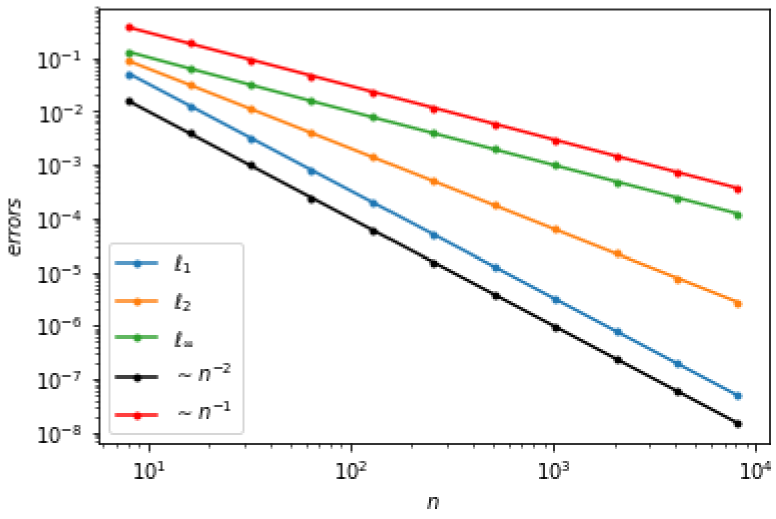
Now, as before, let's look at how different error norms scale with the number of points n, using a log-log plot

In [16]:

```
fig, ax = plt.subplots()
ax.loglog(n,error_one,'.-')
ax.loglog(n,error_two,'.-')
ax.loglog(n,error_inf,'.-')
ax.loglog(n,1*n**(-2.),'.-',color='black')
ax.loglog(n,3*n**(-1.),'.-',color='red')
ax.legend(['$\ell_{1}$', '$\ell_{2}$','$\ell_{\infty}$','$\sim n^{-2}$','$\sim n
^{-1}$'])
ax.set_xlabel("$n$")
ax.set_ylabel("$errors$")
```

Out[16]:

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



This is the same error viewed in different ways!

Different norms pick out different things.

$$\ell_\infty \text{ is dominated by the largest outlier.}$$

$$\ell_1 \text{ reduces the contribution of outliers.}$$

We may want either of these things depending on the situation.

```
Do we want to know about the likelihood of a structure collapse?

Do we want to get an idea about the 'typical' house price in Sydney?
```

Question break

I sure talk about Taylor's theorem a lot.

Can you use it to calculate the error of a finite difference formula?

Using Taylor's theorem

Back to computing derivatives.

```
Q: How can we fix the end points and still get a good error?

A: We can sample more points in the interior.
```

We can use Taylor's Theorem to help us.

Now we want to try

$$Dy_0 \approx \frac{a\,y_0 + b\,y_1 + c\,y_2}{h}$$

For some unknown constant, a, b, c. We've already tried $a = -1$, $b = 1$, and $c = 0$.

In math, this means

$$f'(x) = \frac{af(x) + bf(x+h) + cf(x+2h)}{h} + \mathcal{O}(h^2)$$

Let's solve for a, b, c using Taylor's Theorem.

First,

$$f(x) = f(x)$$

That's it.

Then

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{3!}f'''(x) + \dots$$

And then,

$$f(x+2h) = f(x) + 2hf'(x) + \frac{4h^2}{2}f''(x) + \frac{8h^3}{3!}f'''(x) + \dots$$

Multiply each of these equations by a, b, c respectively, and add them, and divide by h.

$$\begin{aligned} f'(x) = \frac{af(x) + bf(x+h) + cf(x+2h)}{h} &= \frac{a+b+c}{h} f(x) \\ &+ (b+2c) f'(x) \\ &+ (b+4c)\,\frac{h}{2} f''(x) \\ &+ (b+8c)\frac{h^2}{3!} f'''(x) + \ldots \end{aligned}$$

This expression needs to equal $f'(x)$. We can pick off, equations for a, b, c.

So that the $\mathcal{O}(h^{-1})$ term vanishes:

$$a + b + c = 0$$

So that the $\mathcal{O}(1)$ term matches:

$$b + 2c = 1$$

So that the $\mathcal{O}(h)$ term vanishes:

$$b + 4c = 0$$

This can be written as a matrix equation

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 2 \\ 0 & 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

There are many ways to solve this. No matter how you do it,

$$a = -\frac{3}{2}, \quad b = 2, \quad c = -\frac{1}{2}$$

Knowing this, we can estimate the error term,

$$(b+8c)\,\frac{h^2}{3!} f'''(x) = -\frac{h^2}{3} f'''(x)$$

A second order forward difference!

All together:

$$f'(x) \;=\; \frac{-3f(x) \;+\; 4f(x+h) \;-\; f(x+2h)}{2h} \;+\; \frac{h^2}{3}f'''(x) \;+\; \dots$$

Or, going the other way,

$$f'(x) \;=\; \frac{3f(x) \;-\; 4f(x-h) \;+\; f(x-2h)}{2h} \;+\; \frac{h^2}{3}f'''(x) \;+\; \dots$$

Hopefully you can see a few patterns if you look for a little while. This method generalises easily to different numbers of derivatives, and different orders of error. We've also only done this for ***uniform*** spacing of points. The formulae get more complicated, but the principle is the same in every case.

In [17]:

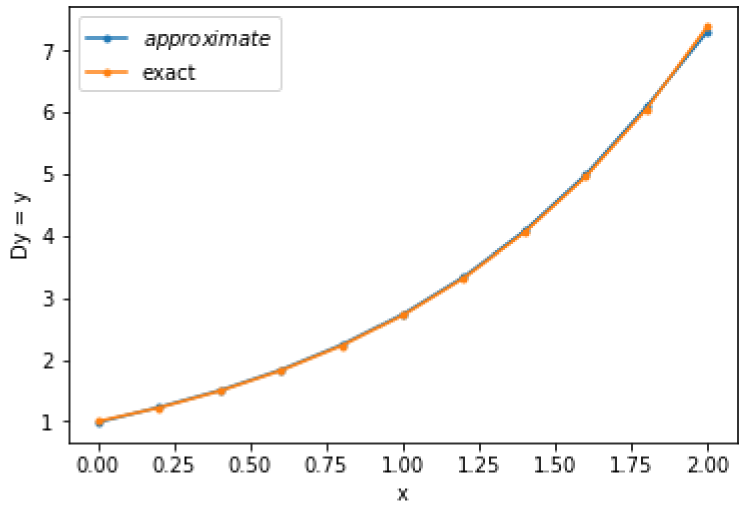
```
def D(y,x,periodic=False):
    Dy = (np.roll(y,-1) - np.roll(y,+1))/(np.roll(x,-1) - np.roll(x,+1))
    if not periodic:
        # Really fix end points
        Dy[0]  = ( -y[2] + 4*y[1]  -  3*y[0]) /(x[2]   - x[0])
        Dy[-1] = ( y[-3] - 4*y[-2] +  3*y[-1])/(x[-1]  - x[-3])

    return Dy
```

In [19]:

```
x  = np.linspace(0,2,11)

y  = np.exp(x)

Dy = D(y,x)

fig, ax = plt.subplots()
ax.plot(x,Dy,marker='.')
ax.plot(x,y,marker='.')
ax.set_xlabel('x')
ax.set_ylabel('Dy = y')
ax.legend(['$approximate$', 'exact'])
```

Out[19]:

```
<matplotlib.legend.Legend at 0x7f9a7aa3b860>
```



MUCH BETTER!!

In [20]:

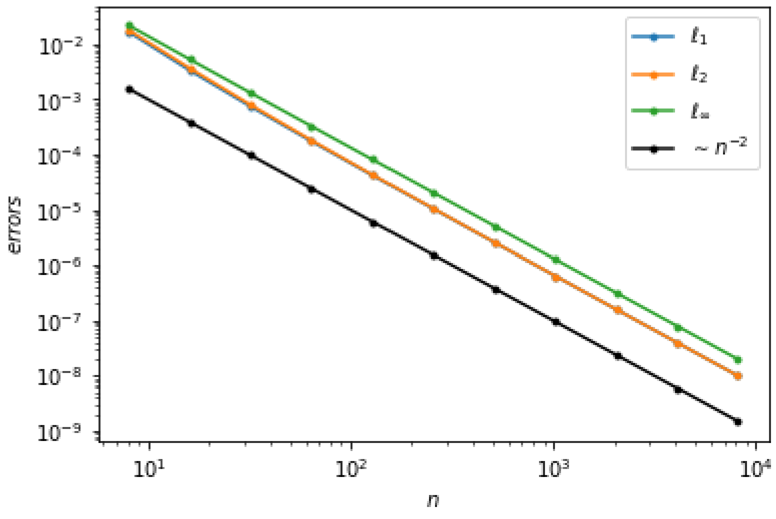
```
nmin, nmax = 10, 10000
n = np.array([2**i for i in range(int(np.log2(nmin)),int(np.log2(nmax))+1)])

error_one = np.zeros(len(n))
error_two = np.zeros(len(n))
error_inf = np.zeros(len(n))

for k in range(len(n)):

    x  = np.linspace(0,2,n[k])
    y  = np.exp(x)
    Dy = D(y,x)

    error_one[k] = norm(Dy-y,1)/norm(y,1)
    error_two[k] = norm(Dy-y,2)/norm(y,2)
    error_inf[k] = norm(Dy-y,'inf')/norm(y,'inf')
```

In [21]:

```
fig, ax = plt.subplots()
ax.loglog(n,error_one,'.-')
ax.loglog(n,error_two,'.-')
ax.loglog(n,error_inf,'.-')
ax.loglog(n,0.1*n**(-2.),'.-',color='black')
ax.legend(['$\ell_{1}$', '$\ell_{2}$','$\ell_{\infty}$','$\sim n^{-2}$'])
ax.set_xlabel("$n$")
ax.set_ylabel("$errors$")
```

Out[21]:

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



Now all the error measurements agree. When you are doing well, everyone agrees you are doing well. When you are doing poorly, reasonable people might disagree about how poor.

Extrapolation

At first we tried to compute

$$f'(x) = \frac{f(x-h) + f(x+h)}{2h} \; + \; \mathcal{O}(h^2)$$

but we ran into trouble because we didn't know $f(x+h)$ for values on the end point.

Then we used

$$f'(x) \; = \; \frac{3f(x) \; - \; 4f(x-h) \; + \; f(x-2h)}{2h} \; + \; \mathcal{O}(h^2)$$

What if we combine them? We can estimate $f(x+h)$, even though we don't know it. Subtracting the two expressions and multplying by h gives

$$f(x+h) \; = \; 3f(x) \; - \; 3f(x-h) \; + \; f(x-2h) \; + \; h^3 f'''(x) \; + \; \ldots$$

This is one of many ***extrapolation*** formulae we could derive using Taylor's Theorem. This can come in handy sometimes. But remember extrapolation is generally a dangerous procedure unless you understand what you are doing, and you don't try to go too far into the "future".

The direct way you would derive this is by assuming

$$f(x) \; \approx \; af(x-h) + bf(x-2h) + cf(x-3h)$$

It's easier if you use a shifted version of the first extropolation formula. If you use Taylor's Theorem you'll find

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 4 & 9 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

See if you can derive this yourself.

Now that we can derive a lot of finite-difference formulae, we can solve ODEs in a number of different ways.