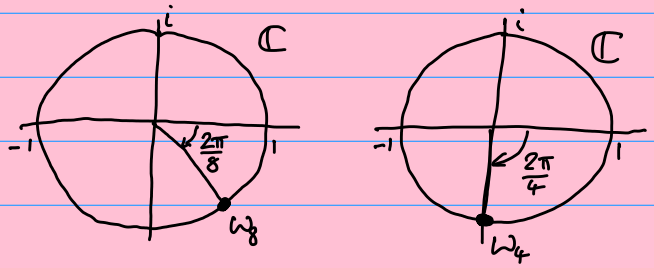
Day 18

1. The Fast Fourier Transform (FFT). Cooley & Tukey 1965
2. Evaluating and estimating derivatives.

We want to form $F_n y$ where $[F_n]_{jk} = \omega_n^{jk}$ where $\omega_n = e^{-i\frac{2\pi}{n}}$.

Examples of ω_n:



Explicitly:...

$$F_n = \begin{bmatrix} \omega_n^0 & \omega_n^0 & \omega_n^0 & \omega_n^0 & \cdots \\ \omega_n^0 & \omega_n^1 & \omega_n^2 & \omega_n^3 & \cdots \\ \omega_n^0 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots \\ \vdots & & & & \end{bmatrix}$$

(j = row index, k = column index)

or in shorthand, just writing the powers of w_n ...

$$\begin{bmatrix} 0 & 0 & 0 & 0 & \cdots \\ 0 & 1 & 2 & 3 & \cdots \\ 0 & 2 & 4 & 6 & \cdots \\ \vdots & & & & \end{bmatrix}$$

For a complete concrete example, here is F_8, writing the powers of w_8:

```
[[ 0  0  0  0  0  0  0  0]
 [ 0  1  2  3  4  5  6  7]
 [ 0  2  4  6  8 10 12 14]
 [ 0  3  6  9 12 15 18 21]
 [ 0  4  8 12 16 20 24 28]
 [ 0  5 10 15 20 25 30 35]
 [ 0  6 12 18 24 30 36 42]
 [ 0  7 14 21 28 35 42 49]]
```

We want to exploit the symmetries of this

To reveal the symmetries, begin by dividing the odd-index (blue) columns by the k=1 column. Dividing means subtracting powers of w:

```
[[ 0  0  0  0  0  0  0  0]
 [ 0  0  2  2  4  4  6  6]
 [ 0  0  4  4  8  8 12 12]
 [ 0  0  6  6 12 12 18 18]
 [ 0  0  8  8 16 16 24 24]
 [ 0  0 10 10 20 20 30 30]
 [ 0  0 12 12 24 24 36 36]
 [ 0  0 14 14 28 28 42 42]]
```

We see duplication of values, which is the key to fast evaluation of $F_n y$.

Now let's mod by n=8, which doesn't change anything because $\omega_8^8 = e^{i2\pi\frac{8}{8}} = e^{i2\pi} = 1$.

```
[[ 0  0  0  0  0  0  0  0]
 [ 0  0  2  2  4  4  6  6]
 [ 0  0  4  4  0  0  4  4]
 [ 0  0  6  6  4  4  2  2]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  2  2  4  4  6  6]
 [ 0  0  4  4  0  0  4  4]
 [ 0  0  6  6  4  4  2  2]]
```

Wow! The upper and lower halves of both the even and odd columns are all the same!

```
[[ 0  0  0  0 ]
 [ 0  2  4  6 ]
 [ 0  4  0  4 ]
 [ 0  6  4  2 ]
 ----------------
 [ 0  0  0  0 ]
 [ 0  2  4  6 ]
 [ 0  4  0  4 ]
 [ 0  6  4  2 ]]
```

All 4 submatrices are in fact

```
0 0 0 0
0 2 4 6
0 4 0 4
0 6 4 2
```

This is F_4 expressed in powers of ω_8

Now observe that the shorthand for F_4, writing powers of ω_4 is

```
0 0 0 0          0 0 0 0
0 1 2 3  mod 4   0 1 2 3
0 2 4 6    =     0 2 0 2
0 3 6 9          0 3 2 1
```

Note this is half of the submatrices above.

Now since $\omega_4 = \omega_8^2$ we see that each of the four submatrices above are in fact F_4!

So in conclusion ...

$$F_8 y = \begin{bmatrix} F_4\, y_{even} + \begin{bmatrix} \omega_8^0 \\ \omega_8^1 \\ \omega_8^2 \\ \omega_8^3 \end{bmatrix} F_4\, y_{odd} \\ \hline F_4\, y_{even} + \begin{bmatrix} \omega_8^4 \\ \omega_8^5 \\ \omega_8^6 \\ \omega_8^7 \end{bmatrix} F_4\, y_{odd} \end{bmatrix}$$

to compensate for our subtraction of $\begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \end{bmatrix}$ from blue columns

first 4 rows

second 4 rows

Thus we can express F_8 y in terms of F_4 applied to subvectors of y

In turn, we can express F_4 in terms of F_2, and so on

until we get to F_1 and F_1 is trivial: $F_1 = [\,1\,]$.

Let's code it up!

FFT

```
import numpy as np
```

```
def myfft(y):
    # I'll use recursion - not the most run-time efficient way
    # but makes the coding very simple!

    n = len(y)
    if n==1: return np.array(y,dtype=complex)

    yeven = y[::2]
    yodd  = y[1::2]

    w = np.exp(-2j*np.pi/n)
    W = w**np.arange(n)  # powers of w  0 thru n-1

    Fyeven = myfft(yeven)
    Fyodd  = myfft(yodd)

    return np.hstack([ Fyeven + W[:n//2]*Fyodd, Fyeven + W[n//2:]*Fyodd  ])
```

```
np.arange(5)
```

```
array([0, 1, 2, 3, 4])
```

```
# Let's test it!
n = 2**15
print(n)
y = np.random.rand(n)

from time import time
tic = time()
myFy = myfft(y)
toc = time()
print((toc-tic),'seconds for my version')

from numpy.fft import fft
tic = time()
npFy = fft(y)  # numpy's version
toc = time()
print((toc-tic),'seconds for numpy version')

np.allclose(myFy,npFy)
```

```
32768
0.2265925407409668 seconds for my version
0.00058746337890625 seconds for numpy version

True
```

myfft is correct but 450 times slower than numpy's fft

(because I'm using recursion which has a high overhead)

Recall that the cost of a brute force formation of the matrix F_n and multiplying it on y is $O(n^2)$. (Matrix-vector multiplication needs n elements each requiring 2n operations.)

But exploiting the observations above, can do it with the following cost.

# of operations for size n = $C_n = 2C_{\frac{n}{2}} + n-1 + n + n$

$n-1$: to construct $\begin{bmatrix} w^0 \\ w^1 \\ w^2 \\ \vdots \\ w^{n-1} \end{bmatrix}$; n: mults by w; n: additions

That is,

$$\boxed{C_n = 2C_{\frac{n}{2}} + 3n - 1}$$

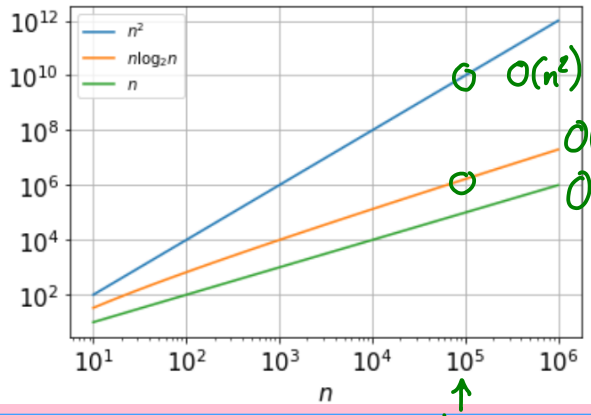
#7

a recurrence relation that we (you, in Homework) can solve.

The conclusion will be that $C_n = O(n \log_2 n)$ (wow!)

which is "almost linear" in n. So much faster than $O(n^2)$.

```
n = np.logspace(1,6,100)
plt.loglog(n,n**2,label='$n^2$')
plt.loglog(n,n*np.log2(n),label='$n\log_2 n$')
plt.loglog(n,n,label='$n$')
plt.legend(); plt.grid();
plt.xticks(fontsize=15); plt.yticks(fontsize=15);
plt.xlabel('$n$',fontsize=15);
```



$O(n^2)$ brute force pre-1965

$O(n \log n)$ FFT

$O(n)$

piano note

The speed of the FFT is what permits for example real-time auto-tuning of pop singers.

6.1 Computing derivatives

What is $f(x_0)$, where f is some given function and x_0 is an arbitrary point in its domain ?

3 methods available:

- symbolic
- interpolation-based (finite-differences, spectral)
- "automatic differentiation" (AD)

● symbolic

If we have a formula for f, this is possible.

Pros: exact (in exact arithmetic), and very close ($\mathcal{E}_{mach}$) in machine arithmetic.

Cons: inefficient due to sub-expression repetition.

Let's look at an example using sympy ...

Symbolic differentiation

```
import sympy as sp
```

```
x = sp.symbols('x')
y = 3*x + sp.sin(7*x**2/(1+x))
y
```

$$3x + \sin\left(\frac{7x^2}{x+1}\right)$$

```
yp = sp.diff(y,x)
yp
```

$$\left(-\frac{7x^2}{(x+1)^2} + \frac{14x}{x+1}\right)\cos\left(\frac{7x^2}{x+1}\right) + 3$$

```
ypp = sp.diff(y,x,x)
ypp
```

$$\frac{7\left(-\frac{7x^2\left(\frac{x}{x+1}-2\right)^2\sin\left(\frac{7x^2}{x+1}\right)}{x+1} + 2\left(\frac{x^2}{(x+1)^2} - \frac{2x}{x+1} + 1\right)\cos\left(\frac{7x^2}{x+1}\right)\right)}{x+1}$$

```
ypp = sp.diff(y,x,2)
ypp
```

$$\frac{7\left(-\frac{7x^2\left(\frac{x}{x+1}-2\right)^2\sin\left(\frac{7x^2}{x+1}\right)}{x+1} + 2\left(\frac{x^2}{(x+1)^2} - \frac{2x}{x+1} + 1\right)\cos\left(\frac{7x^2}{x+1}\right)\right)}{x+1}$$

```
yp.subs({x:1.1})
```

$$-0.39949092483303$$

```
ypfunc = sp.lambdify(x,yp,'numpy')  # convert expression to a function
```

```
ypfunc(1.1)
```

```
-0.3994909248330303
```

```
import numpy as np
ypfunc( np.array([3.2,1.7,-2.1]) )
```

```
array([1.60998955, 5.13550871, 1.81199062])
```

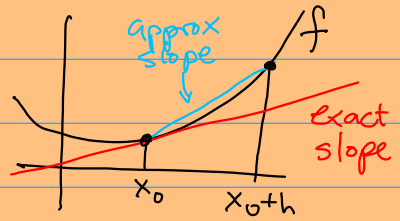
● finite-difference approximation

Pros: applicable even when we do NOT have a formula for f

Cons: truncation error
catastrophic round-off error
} significant even with the best compromise

Example: For f in C^2, approximate $f'(x_0)$.

Use a difference quotient, and apply Taylor's theorem to bound the error:



$$f(x_0+h) = f(x_0) + f'(x_0)h + f''(\xi)\frac{h^2}{2!} \quad \text{with } \xi \text{ between } x_0 \text{ \& } x_0+h.$$

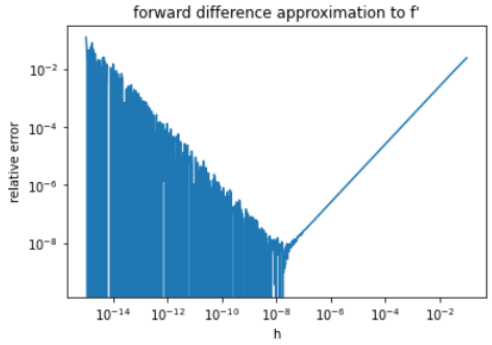
Solving for $f'(x_0)$:

$$f'(x_0) = \underbrace{\frac{f(x_0+h)-f(x_0)}{h}}_{\text{forward difference approximation}} \underbrace{- \frac{h}{2}f''(\xi)}_{\text{truncation error} = O(h)}$$

Let's try it out, and see how the error depends on h ...

Finite difference approximation

```
def f(x):        return  np.cos(x)
def fp_exact(x): return -np.sin(x)

# compute error in forward FD approx to f' for a range of h values ...

h = 10**np.linspace(-15,-1,1000)  # various values of h
x0 = 1.1

fdapprox = ( f(x0+h) - f(x0) )/h  # for all the h-values
exact_deriv = fp_exact(x0)

total_fd_error = fdapprox - exact_deriv
total_relative_error = total_fd_error / exact_deriv

import matplotlib.pyplot as plt
plt.loglog(h,total_relative_error)
plt.xlabel('h')
plt.ylabel('relative error')
plt.title("forward difference approximation to f'");
```



Let's now compute a bound on the relative error that includes the round-off error we suffer when h is small ...

$$\frac{\text{machine}\left(\frac{f(x+h)-f(x)}{h}\right) - f'(x)}{f'(x)}$$

$$= \frac{\left(f\left((x+h)(1+\delta_1)\right)(1+\delta_2) - f(x)(1+\delta_3)\right)(1+\delta_4)(1+\delta_5) - 1}{h f'(x)}$$

$x + h + \delta_1 + \delta_1 h$

$$= \frac{\left[\left(f(x) + (h+\delta_1+\delta_1 h)f'(x) + \frac{(h+\delta_1+\delta_1 h)^2}{2} f''(x) + \ldots\right)(1+\delta_2) - f(x) - \delta_3 f(x)\right](1+\delta_4)(1+\delta_5) - 1}{h f'(x)}$$

with a little help from sympy ...

$$= \frac{1}{h}\left(\delta_1 + (\delta_2 - \delta_3)\frac{f}{f'}\right) + \left(\delta_1 + \delta_2 + \delta_4 + \delta_5 + \delta_1 \frac{f''}{f'}\right)$$

$$+ \frac{h f''}{2 f'}\left(1 + 2\delta_1 + \delta_2 + \delta_4 + \delta_5\right) + \text{h.o.t.}$$

delta2 and delta3 are the relative errors in evaluating f, which can be many multiples of macheps if there are many steps in evaluating f — say μ times $\varepsilon/2$.

With $|\delta_1|, |\delta_4|, |\delta_5| \le \frac{\varepsilon_{mach}}{2}$ and $|\delta_2|, |\delta_3| \le \mu \frac{\varepsilon_{mach}}{2}$

Adding the round-off and truncation errors:

$$|\text{rel error in } f'| \le \frac{1}{h}\left(1 + 2\mu\left|\frac{f}{f'}\right|\right)\frac{\varepsilon_{mach}}{2} + \left(3 + \mu + \left|\frac{f''}{f'}\right|\right)\frac{\varepsilon_{mach}}{2}$$

$$+ \frac{h}{2}\left|\frac{f''}{f'}\right|$$ ← truncation error previously anticipated

$$\frac{d}{dh}\,\text{rel error bound} = -\frac{1}{h^2}\left(1 + 2\mu\left|\frac{f}{f'}\right|\right)\frac{\varepsilon_{mach}}{2} + \frac{1}{2}\left|\frac{f''}{f'}\right| \overset{\text{set}}{=} 0$$

$$h = \sqrt{\left|\frac{f'}{f''}\right|\left(1 + 2\mu\left|\frac{f}{f'}\right|\right)\varepsilon_{mach}} = \sqrt{\quad}\sqrt{\varepsilon_{mach}}$$

$\frac{A}{h^2} = B \quad h = \sqrt{\frac{A}{B}}$

Summarizing, the error bound on the forward difference $\frac{f(x+h)-f(x)}{h} \approx f'(x)$

is minimized at

$$h_{opt} = \sqrt{\quad}\sqrt{\varepsilon_{mach}}$$

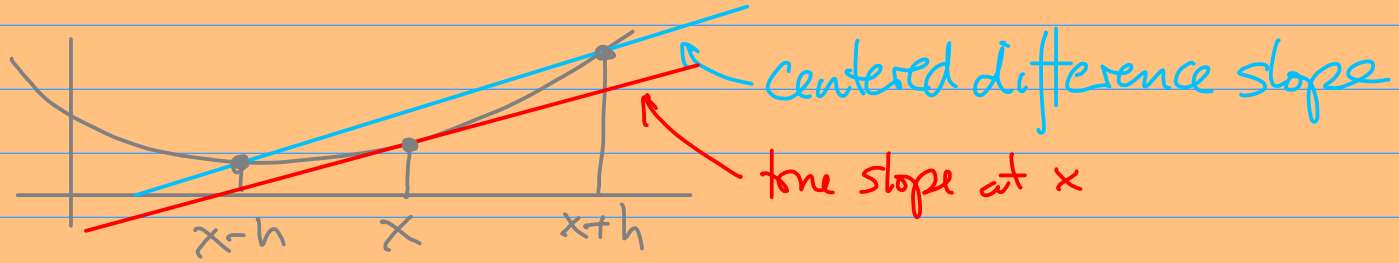
some number depending on f, maybe 5

and from above

$$\text{rel. error}\Big|_{h=h_{opt}} = \text{Some number like 5}\ \sqrt{\varepsilon_{mach}} \leftarrow \underline{\text{huge}}\ \text{compared to } \varepsilon_{mach}$$

because $\sqrt{\varepsilon_{mach}} \gg \varepsilon_{mach}$!

Can we do better? Yes - use a centered difference



If $f \in C^3$

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(\xi_+)$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(\xi_-)$$

Thus

$$\frac{f(x+h)-f(x-h)}{2h} = \frac{2hf'(x) + \frac{h^3}{3}\left(\frac{f'''(\xi_+)+f'''(\xi_-)}{2}\right)}{2h}$$

$$= f'(x) + \frac{h^2}{6}f'''(\xi) = f'(x) + O(h^2)$$

Compare with the O(h) error for the forward difference.

Let's do an empirical comparison of the errors in the forward and centered differences ...

[space for empirical comparison of forward and centered differences]

● "automatic" differentiation (AD)

(differentiation-rule-based)

Would be great if we had another method that has
(i) no truncation error
(ii) no catastrophic round-off error

There is such a thing: we use a "differentiation arithmetic".

Suppose you have 2 functions f & g, and define $h = f\,g$ (their product).

Question: What do we need to know in order to compute say $h'(5.7)$?

Answer:

So develop an arithmetic of objects like $< f(5.7), f'(5.7) >$, $< g(5.7), g'(5.7) >$,

for example $< 3.4, -1.3 >$, $<7, -2>$,

by defining the product of two objects like this:

Compare complex arithmetic:

Other arithmetic operations, and elementary transcendental functions, similarly ...