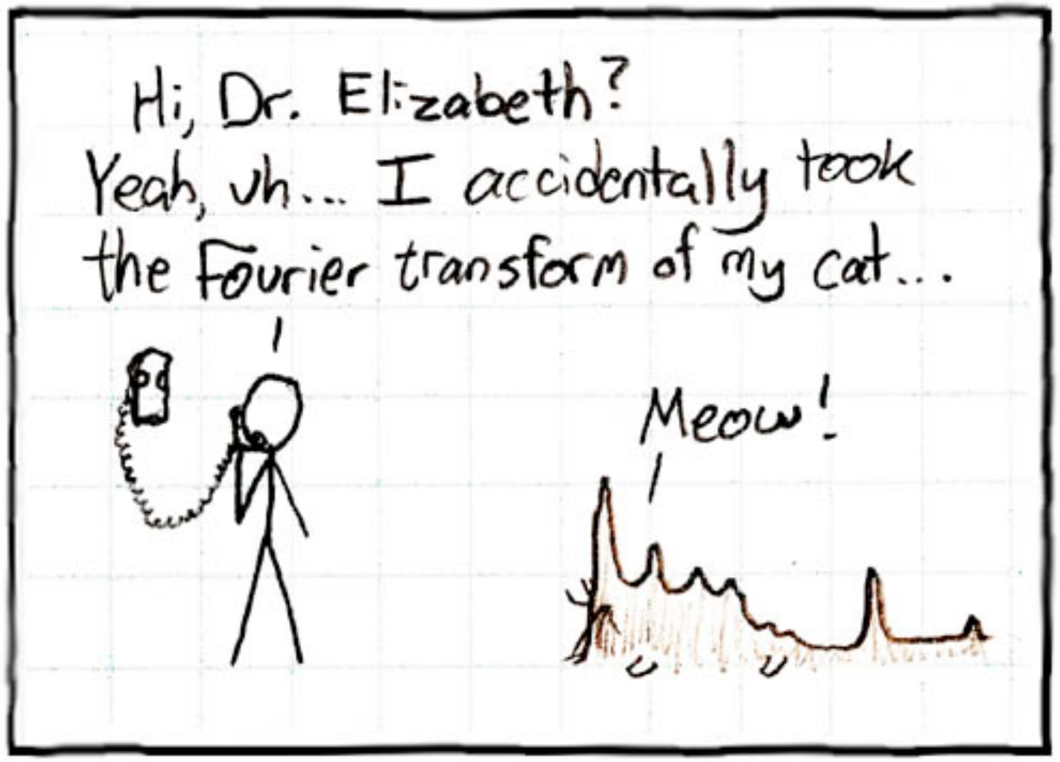
PY410 / 505
Computational Physics 1

Salvatore Rappoccio

Spectral Analysis

- You should be familiar with Fourier transforms :
  - http://en.wikipedia.org/wiki/Fourier_transform



Continuous

$$f(x) = \int_{-\infty}^{\infty} F(k)\, e^{2\pi i k x}\, dk$$

$$F(k) = \int_{-\infty}^{\infty} f(x)\, e^{-2\pi i k x}\, dx.$$

Discrete

$$Y_{k+1} = \sum_{j=0}^{n-1} y_{j+1} e^{-2\pi i j k / N}$$

Spectral Analysis

- Fourier was looking to solve the heat conduction equation :

$$\frac{\partial}{\partial t}T(x,t) = \kappa \frac{\partial^2}{\partial x^2}T(x,t)$$

  – We'll actually do this later in the class

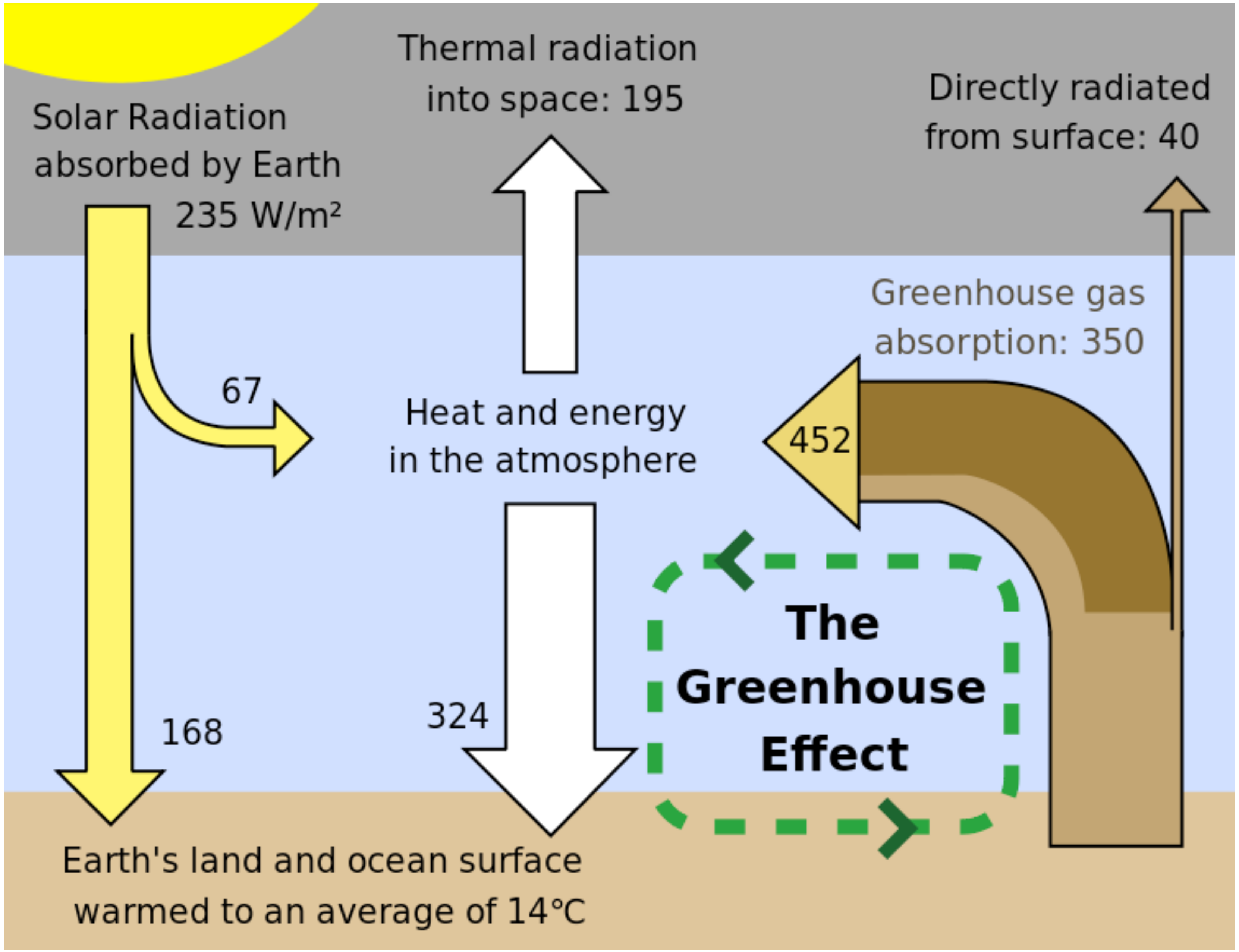
- The solution can be expanded as a sum of trigonometric functions :

$$T(x,t) = c_0 + c_1 x + \sum_{n=0}^{\infty} a_n e^{-\kappa \lambda_n^2 t} \sin\left(\lambda_n x\right) \qquad \text{where } \lambda_n = n\pi/L$$

$$a_n = \frac{2}{L}\int_0^L dx\, [T(x,0) - c_0 - c_1 x] \sin(\lambda_n x)$$

Spectral Analysis

- Fourier was also credited with the discovery of the Greenhouse effect!
  - http://en.wikipedia.org/wiki/Greenhouse_effect
- So, we'll combine these two things!



Discrete Fourier Transform

- Suppose you take a rod and measure the temperature at N equally spaced points :

$$x_j = j\frac{L}{N}$$

- Then the temperatures can be expressed as :

$$T(x_j) - T(0) = T_j = \sum_{k=1}^{N-1} F_k \sin\frac{\pi k j}{N}$$

- With coefficients :

$$F_k = \frac{2}{N}\sum_{j=1}^{N-1} T_j \sin\frac{\pi k j}{N}$$

Discrete Fourier Transform

- Write this in matrix form :

$$\vec{T} = \mathbf{S}\vec{F} \qquad \vec{F} = \frac{2}{N}\mathbf{S}^{-1}\vec{T}$$

- where the components of S are

$$S_{jk} = \sin\frac{\pi jk}{N}$$

- There are N-1 coefficients
- Each is a representation of N-1 terms

- So the total is $(N-1)^2$ operations

"Big-Ohh" Notation

- The "big-ohh" notation stands for "order"

- $O(N^2)$ operations means "the leading coefficient in the number of operations scales like N^2"

- Remember, "operations" here really means "multiplications"... addition is cheap!

- In computing, we want to minimize this as much as possible since the computational time scales the same way

Discrete Fourier Transform

- Loop over the indices of the Fourier series (0...N-1)
  - For each, compute each coefficient :
    - Loop over the indices of the expansion (0...N-1)
    - Compute the angle
    - Add to the transform

```
def sine_transform(data):
    """Return Fourier sine transform of a real data vector"""
    N = len(data)
    transform = [ 0 ] * N
    for k in range(N):
        for j in range(N):
            angle = math.pi * k * j / N
            transform[k] += data[j] * math.sin(angle)
    return transform
```

Fast Fourier Transforms

- Can we do any better?

- Heck yes!

- Divide et impera! (divide and conquer)
  - "Cooley-Tukey" method

- Divide the sampling into some number 2^n
- Then you can do half at a time and add them together at the end

Fast Fourier Transforms

- So, the discrete Fourier transform can be written as

$$Y_{k+1} = \sum_{j=0}^{N-1} y_{j+1} W^{kj}$$

$$W = e^{-2\pi i/N}$$

"Roots of unity"

- But let's work instead in binary :

$$x = \sum_a 2^a x_a \qquad x_a = (0, 1)$$

(example: x = 101 = 5 decimal)

- So the coefficients in binary format are :

$$y_{j+1} = y(\{x_a\})$$

Fast Fourier Transform

- Take a concrete example of $N=2^3 = 8$
- Then we have :

$$j = 4j_2 + 2j_1 + j_0$$

$$k = 4k_2 + 2k_1 + k_0$$

- We now define :

$$y_{j+1} = y(j_2, j_1, j_0) \quad Y_{k+1} = Y(k_2, k_1, k_0)$$

- The DFT now becomes :

$$Y(k_2, k_1, k_0) = \sum_{j_0=0}^{1} \sum_{j_1=0}^{1} \sum_{j_2=0}^{1} y(j_2, j_1, j_0) W^{(4k_2+2k_1+k_0)(4j_2+2j_1+j_0)}$$

Fast Fourier Transform

- Notice now that $W^8 = W^{16} = \ldots = 1$ since $W = e^{-2\pi i/8}$ and $e^{-2\pi i n} = 1$

- So if you separate this out you can notice :

$$W^{(4k_2+2k_1+k_0)4j_2} = W^{4k_0 j_2}$$

$$W^{(4k_2+2k_1+k_0)2j_1} = W^{(2k_1+k_0)2j_1}$$

Danielson-Lanczos lemma

Fast Fourier Transform

- Can now compute this recursively!

$$y_1(k_0, j_1, j_0) = y(0, j_1, j_0) + y(1, j_1, j_0)W^{4k_0}$$

$$y_2(k_0, k_1, j_0) = y_1(k_0, 0, j_0) + y_1(k_0, 1, j_0)W^{(4k_1+2k_0)}$$

$$y_3(k_0, k_1, k_2) = y_2(k_0, k_1, 0) + y_2(k_0, k_1, 1)W^{(4k_2+2k_1+k_0)}$$

- Reverse the order of the bits ("bit unscrambling") and you get Y!

$$Y(k_2, k_1, k_0) = y_3(k_0, k_1, k_2)$$

Fast Fourier Transform

- Can now compute this recursively!

log₂(N) layers

N complex multiplications, N complex additions

N complex multiplications, N complex additions $(4k_1+2k_0)$

y N complex multiplications, N complex additions $(4k_2+2k_1+k_0)$

- Reverse the order of the bits (“bit unscrambling”) and you get Y!

N log2(N) operations! $k_1, k_2)$

Fast Fourier Transform

- Input data in space domain
- break into even and odd bits

$$y_1(k_0, j_1, j_0) = y(0, j_1, j_0) + y(1, j_1, j_0)W^{4k_0}$$

- use the recursion relation to solve each half individually

Cooley/Tukey algorithm

$$y_1(k_0, j_1, j_0) = y(0, j_1, j_0) + y(1, j_1, j_0)W^{4k_0}$$

$$y_2(k_0, k_1, j_0) = y_1(k_0, 0, j_0) + y_1(k_0, 1, j_0)W^{(4k_1+2k_0)}$$

Timing of Fourier Transforms

	CPU Time Required at 10^6 Flops	
N	Discrete Fourier Transform	Fast Fourier Trasform
10^3	1.0 sec	0.01 sec
10^6	10^6 sec = 12 days	20 sec
10^9	10^{12} sec = 32,000 years	3.0×10^5 sec = 8.3 hours

Fast Fourier Transform

- Input data in space domain
- break into even and odd bits
- use the recursion relation to solve each half individually

```
fft ( x ) :
   n = size of data
   recursively call fft(even x's)
   recursively call fft(odd x's)
   combine results
```

Fast Fourier Transform

- Input data in space domain
- break into even and odd bits
- use the recursion relation to solve each half individually

```
fft ( x ) :
   n = size of data
   recursively call fft(even x’s)
   recursively call fft(odd x’s)
   combine results
```

```
from cmath import exp, pi

def fft(x):
    N = len(x)
    if N <= 1: return x
    even = fft(x[0::2])
    odd =  fft(x[1::2])
    return [even[k] + exp(-2j*pi*k/N)*odd[k] for k in xrange(N/2)] + \
           [even[k] - exp(-2j*pi*k/N)*odd[k] for k in xrange(N/2)]

print fft([1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0])
```

http://rosettacode.org/wiki/Fast_Fourier_transform#Python

Note! This very simple form only works if $N = 2^n$, so be careful!

Fast Fourier Transform

- Does it work?

- Let’s test it out : $y_k = \sin{(\frac{2\pi f}{N} k)}$

- What do we expect?

Fast Fourier Transform

- Does it work?
- Let's test it out : $y_k = \sin\left(\frac{2\pi f}{N} k\right)$
- What do we expect?

At frequency of "f", we'll get a spike!

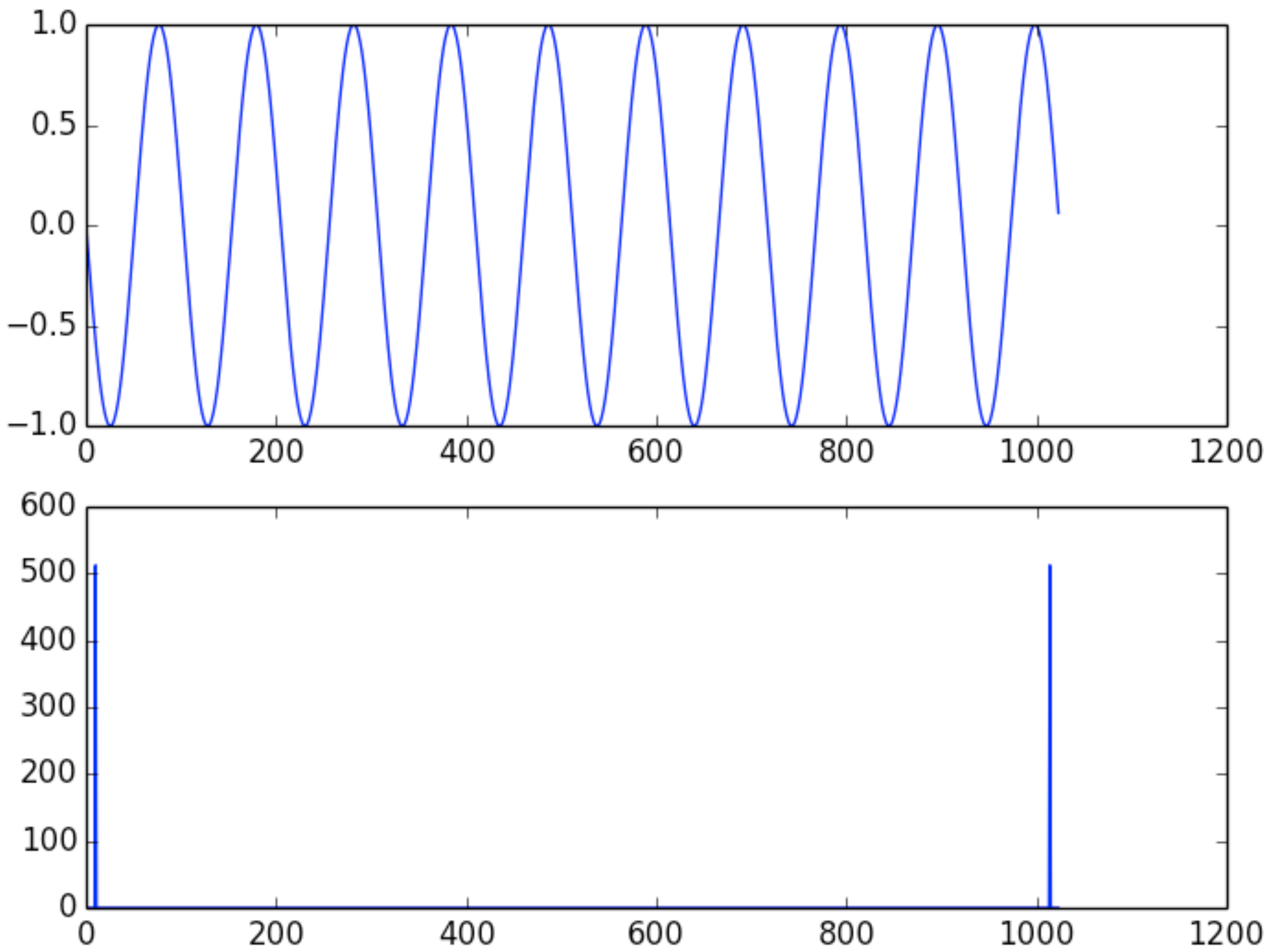
Fast Fourier Transform

```
from cmath import exp, pi
from math import sin, cos
import matplotlib.pyplot as plt

import numpy
from numpy.fft import fft
from numpy import array

def fft(x):
    N = len(x)
    if N <= 1: return x
    even = fft(x[0::2])
    odd =  fft(x[1::2])
    return [even[k] + exp(-2j*pi*k/N)*odd[k] for k in xrange(N/2)] + \
           [even[k] - exp(-2j*pi*k/N)*odd[k] for k in xrange(N/2)]
```

Fast Fourier Transform

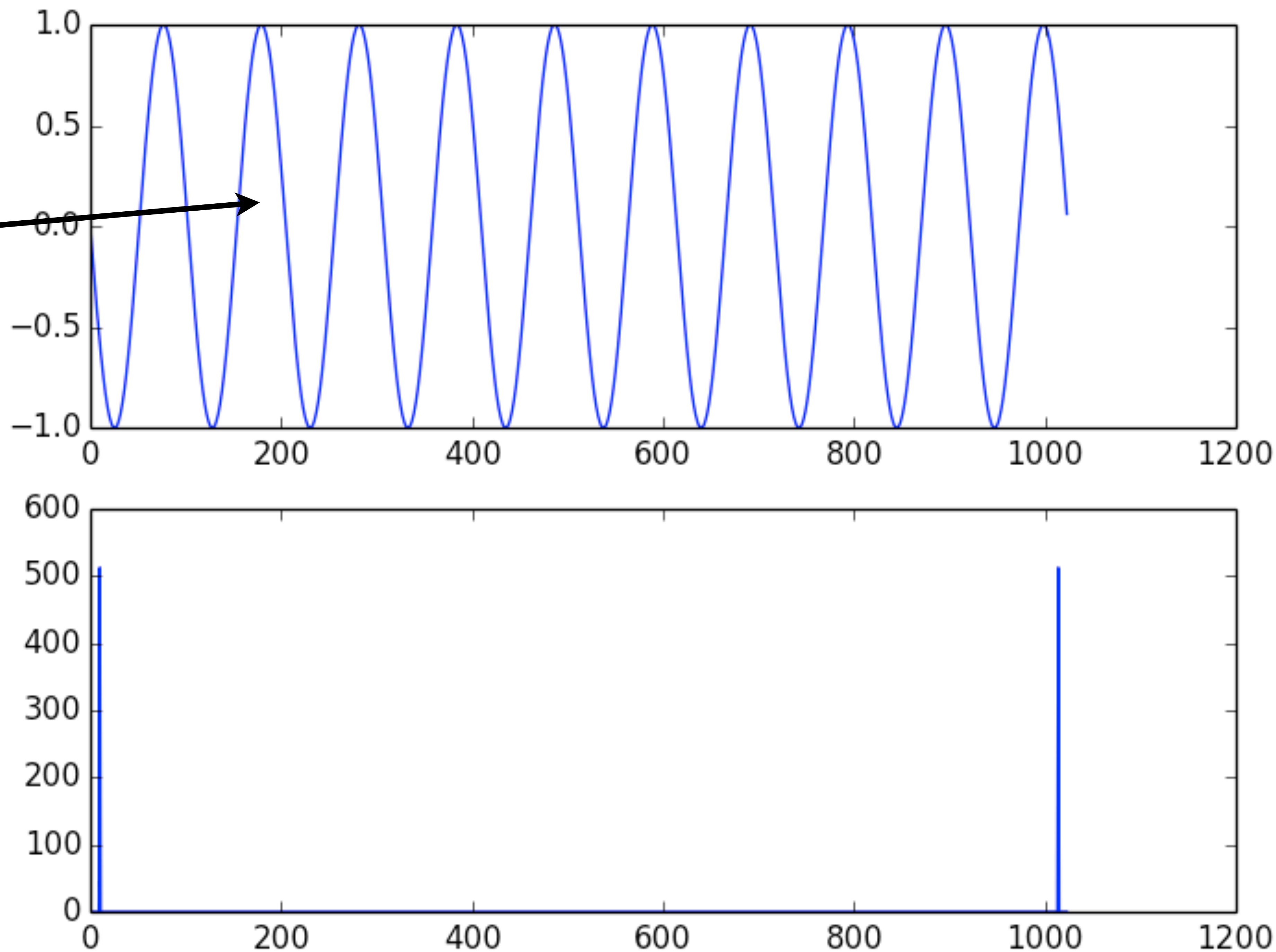
```
import matplotlib.pyplot as plt

from fft import fft
from numpy import array
import math

plotfirst = True

if plotfirst == True :
    # make some fake data :

    N = 1024
    f = 10.0

    x = array([ float(i) for i in xrange(N) ] )
    y = array([ math.sin(-2*math.pi*f* xi / float(N))  for xi in x ])
    #y = array([ xi for xi in x ])
    Y = fft(y)

    Yre = [math.sqrt(Y[i].real**2 + Y[i].imag**2) for i in xrange(N)]

    s1 = plt.subplot(2, 1, 1)
    plt.plot( x, y )

    s2 = plt.subplot(2, 1, 2)
    #s2.set_autoscalex_on(False)
    plt.plot( x, Yre )
    #plt.xlim([0,20])

    plt.show()
```

Fast Fourier Transform



Fast Fourier Transform

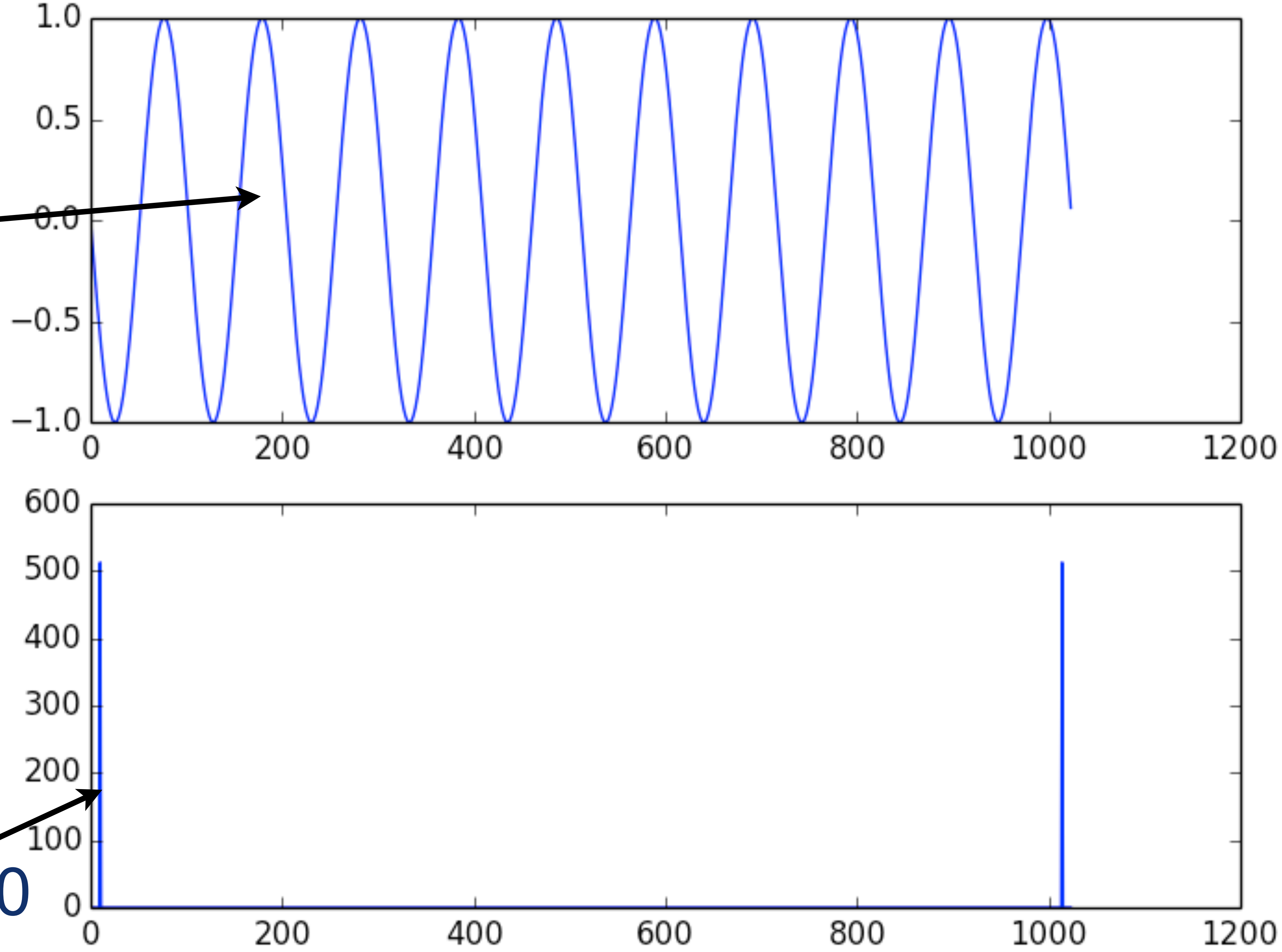
Yay! Sinusoid, freq. 10/N



Fast Fourier Transform

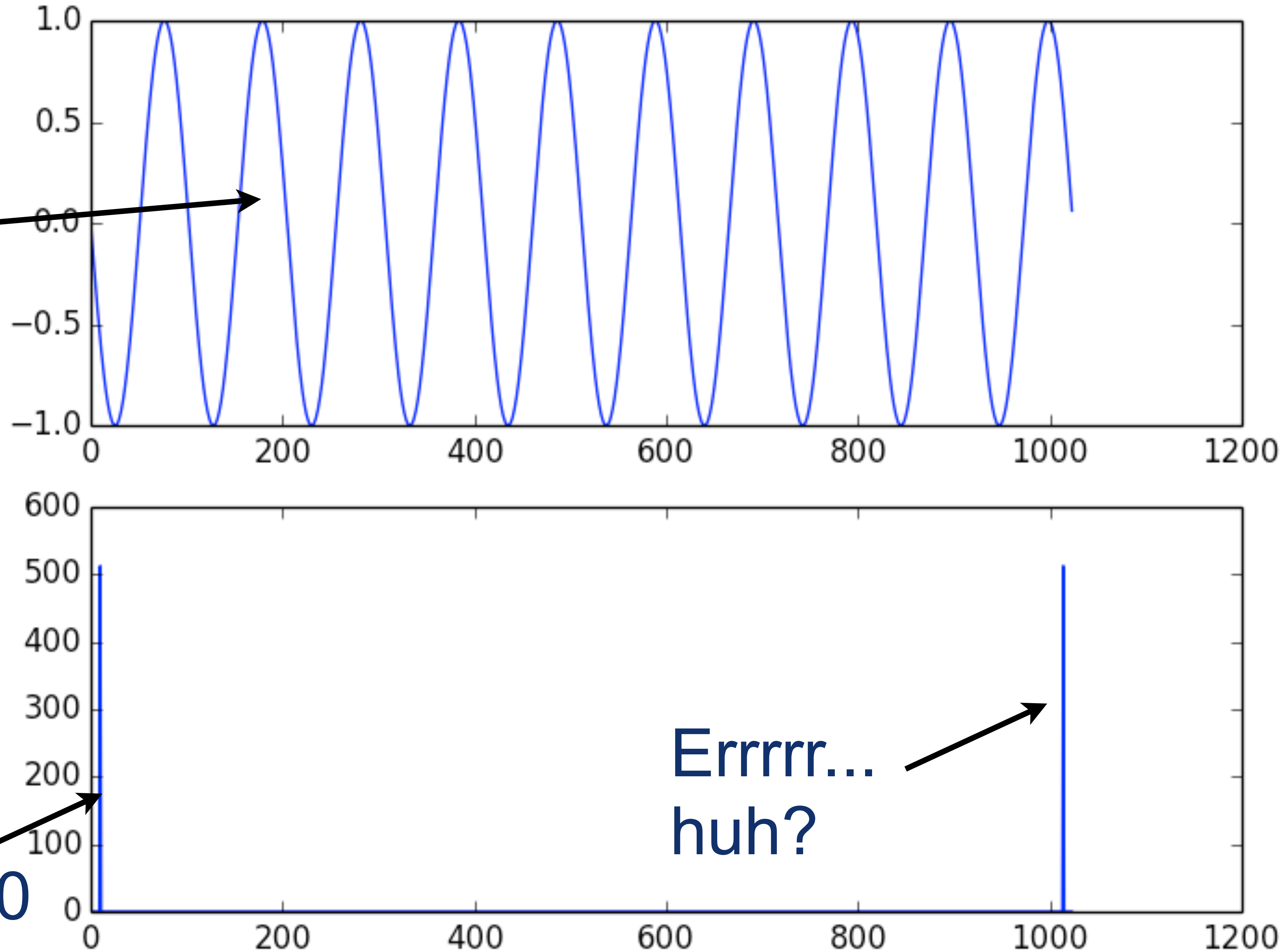
Yay!
Sinusoid,
freq. 10/N

Yay!
Spike at 10



Fast Fourier Transform

Yay!
Sinusoid,
freq. 10/N

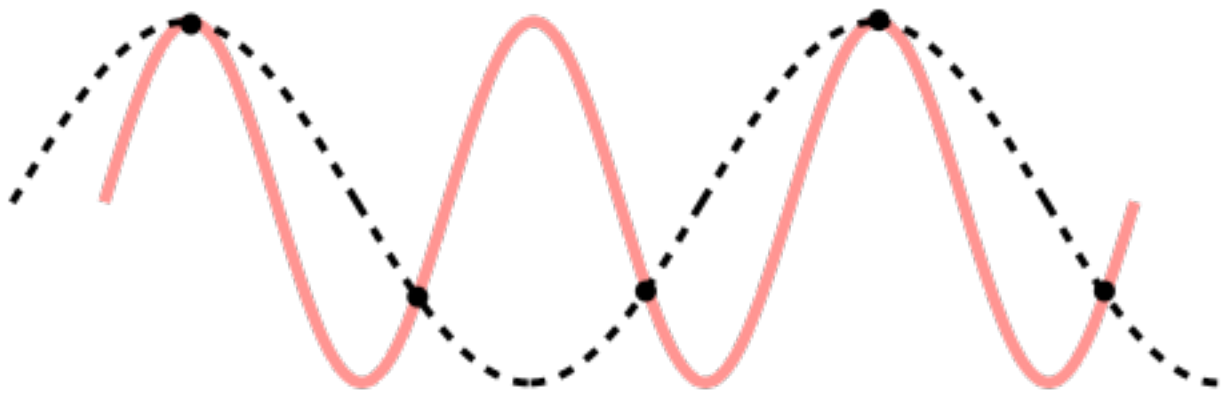


Yay!
Spike at 10

Errrrr...
huh?

Fast Fourier Transform

- Aliasing!



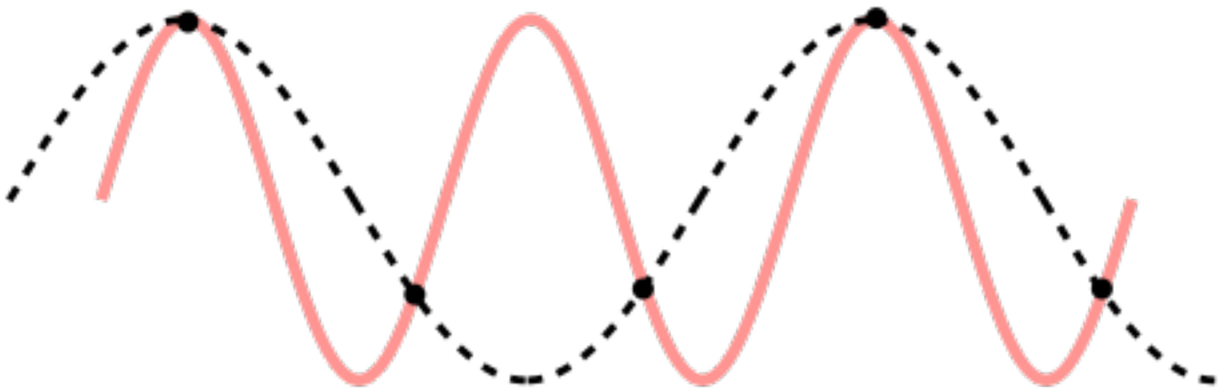
- This relates to the “Nyquist frequency” (half of the sampling rate)

- The upper half of the spectrum is a mirror image of the lower half, separated by the Nyquist frequency

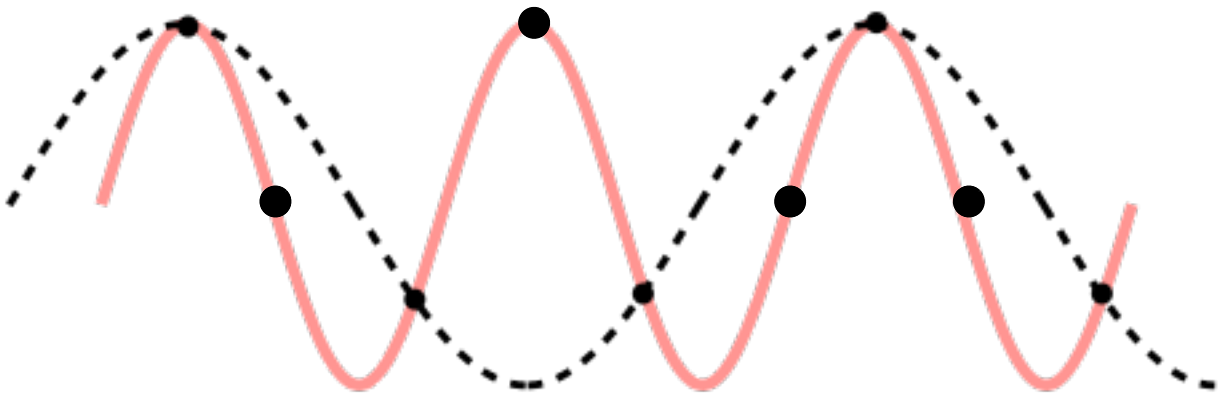
Fast Fourier Transform

- Choosing the right interval depends on the structure that you expect your signal to have

- If your signal is the red, these four are bad!



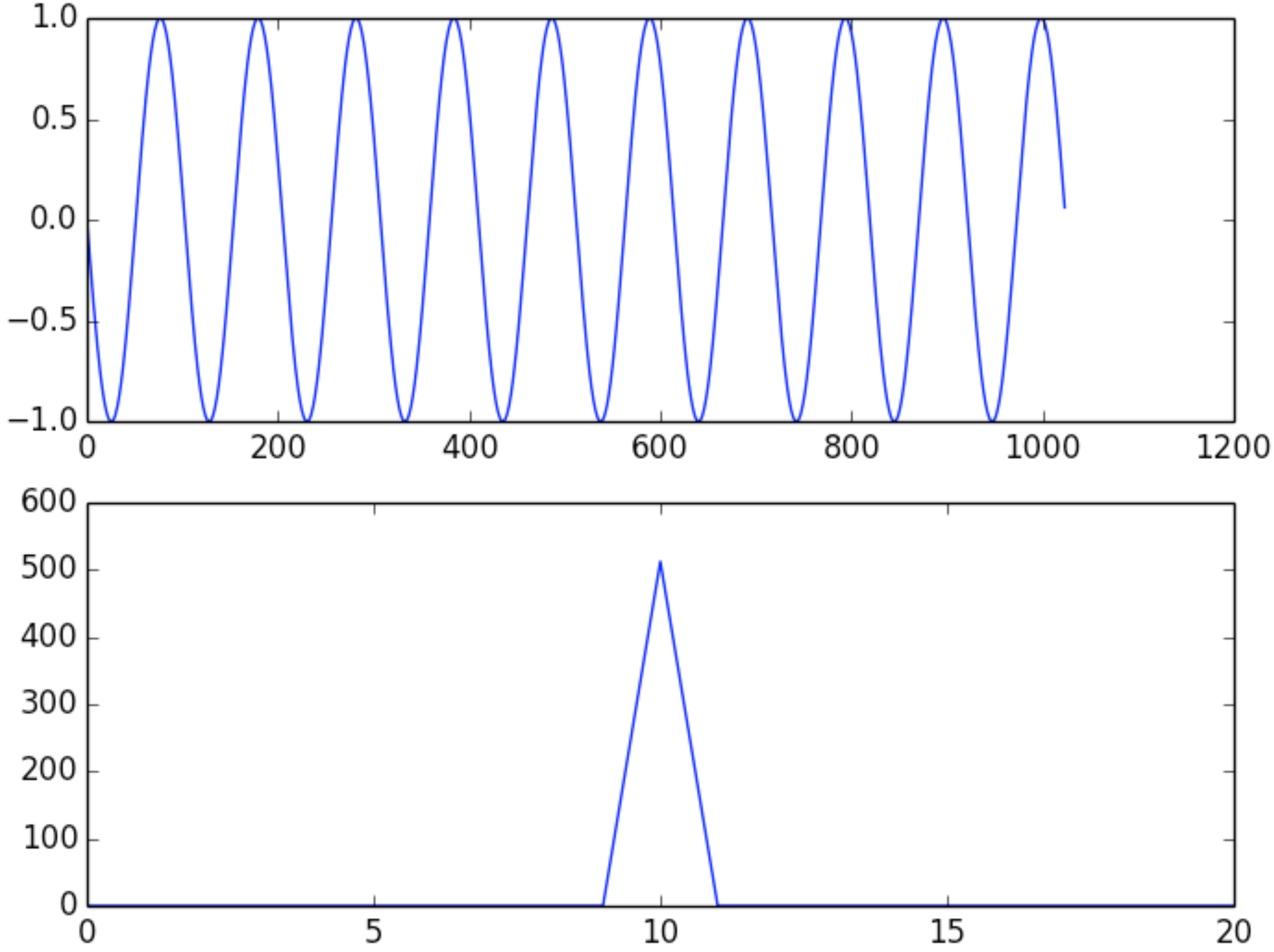
- If you increase the sampling, you can distinguish and remove the aliasing :



Fast Fourier Transform

- Typically if you expect your signal to have the highest frequency f, you should have a sampling of at least 2f or 4f (higher is better, of course)

Fast Fourier Transform



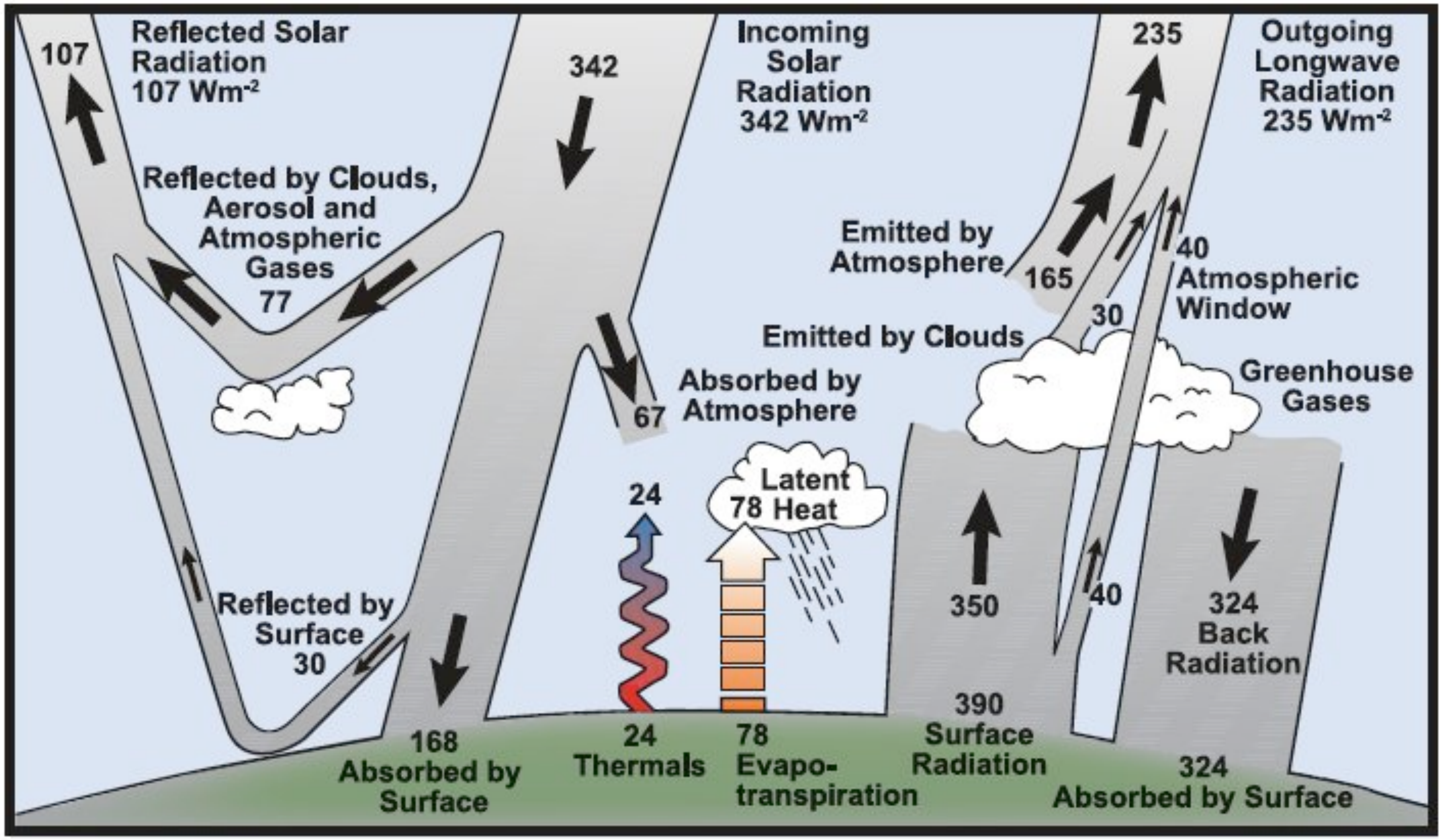
Global Warming

[Humans] have now all but destroyed this once salubrious planet as a life-support system in fewer than 200 years, mainly by making thermodynamic whoopee with fossil fuels.

-- Kurt Vonnegut

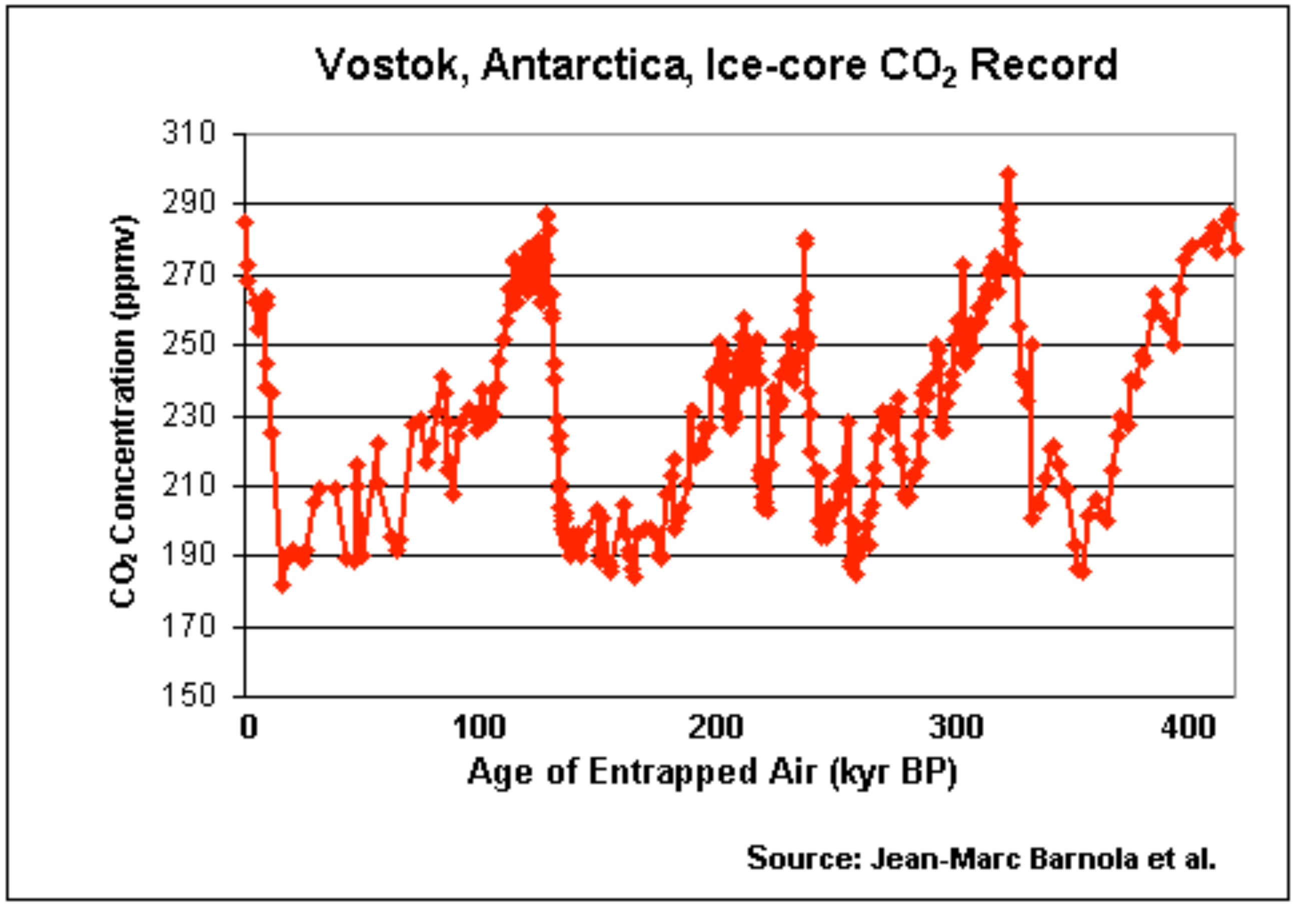
Global Warming

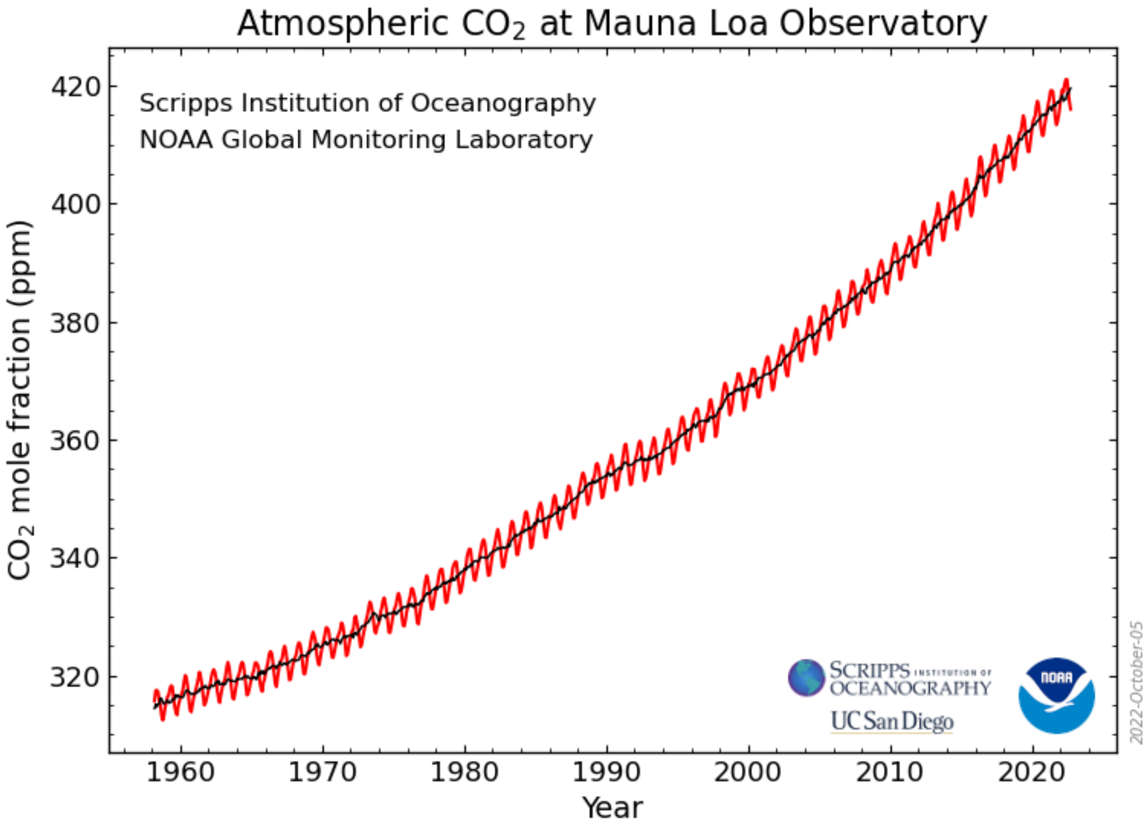
- Solar energy incident on Earth's is partially reflected back into space as lower wavelength infrared radiation
- CO_2 in the atmosphere tends to trap this radiation and is an important factor in the phenomenon of global warming. Global warming has important consequences for the biosphere and human society.
- Interested parties should read the reports of the Intergovernmental Panel on Climate Change http://www.ipcc.ch/.



Global Warming

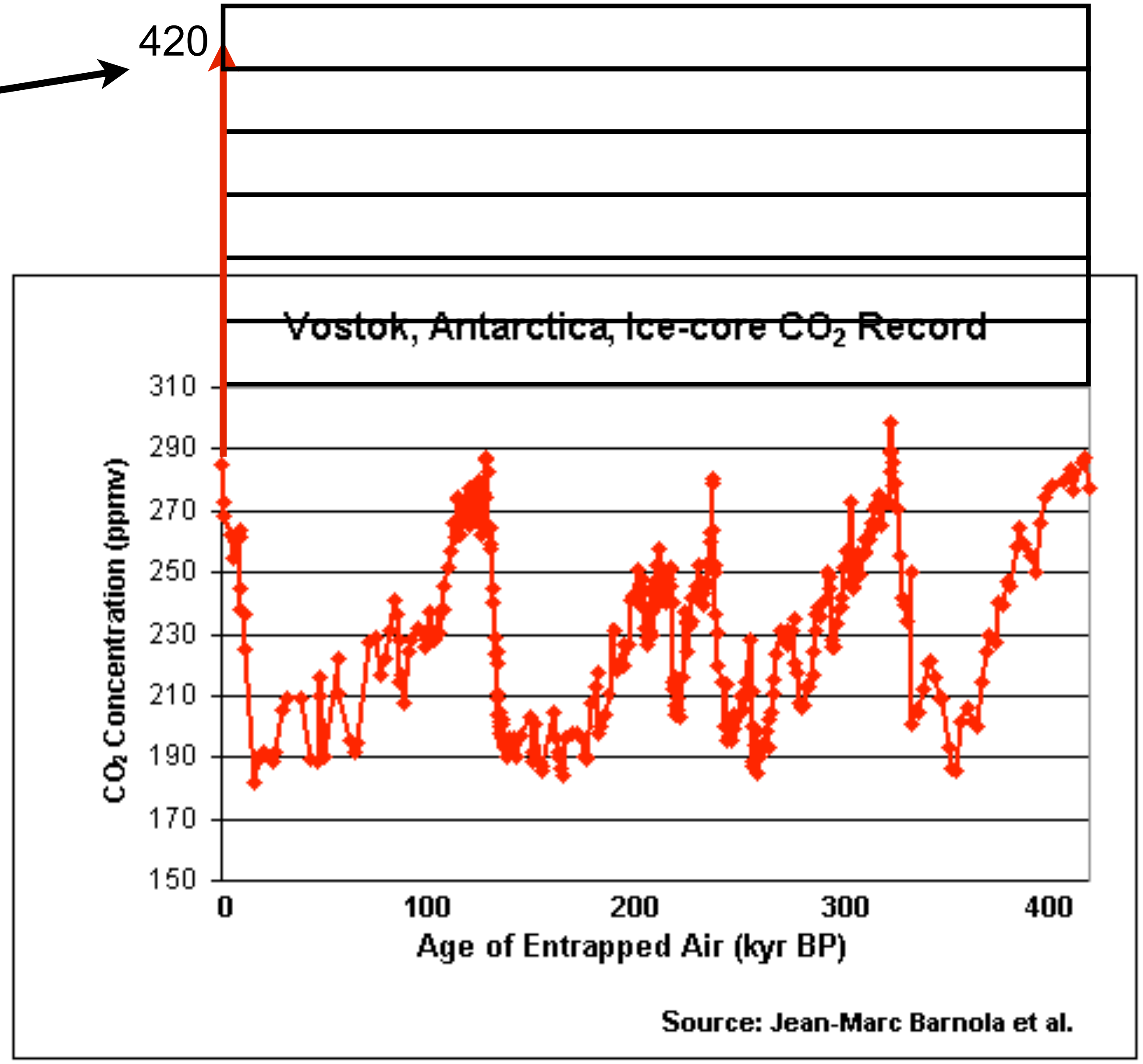
- Situated at 11,135 ft on the north flank of the Mauna Loa volcano on the Big Island of Hawaii, the National Oceanic and Atmospheric Administration's Mauna Loa Observatory http://www.mlo.noaa.gov/ has been monitoring the level of carbon dioxide in Earth's atmosphere for over 50 years. The levels of this greenhouse gas have been rising steadily during this observation period.
- Globally we're at the highest point in hundreds of thousands of years
  - Can read ice core data from Vostok, Antarctica
  - http://cdiac.ornl.gov/trends/co2/vostok.html





Global Warming

Today



Analyze the data!

- We already have our fft code, we just have to read in the spectrum and perform the transformation

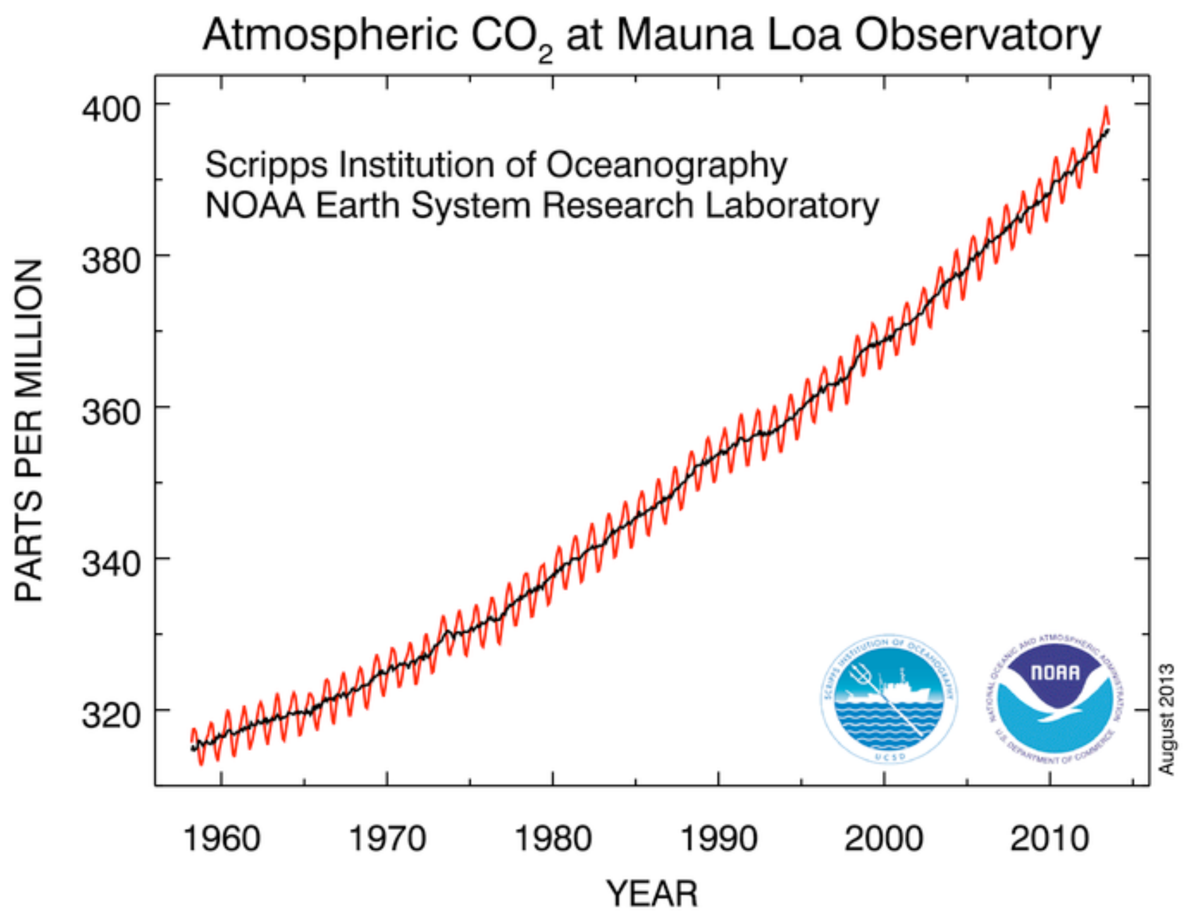
```
else :
    # data downloaded from ftp://ftp.cmdl.noaa.gov/ccg/co2/trends/co2_mm_mlo.txt
    print ' CO2 Data from Mauna Loa'
    data_file_name = 'co2_mm_mlo.txt'
    file = open(data_file_name, 'r')
    lines = file.readlines()
    file.close()
    print ' read', len(lines), 'lines from', data_file_name

    yinput = []
    xinput = []

    for line in lines :
        if line[0] != '#' :
            try:
                words = line.split()
                xval = float(words[2])
                yval = float( words[4] )
                yinput.append( yval )
                xinput.append( xval )
            except ValueError :
                print 'bad data:',line

    y = array( yinput[0:256] )
    x = array([ float(i) for i in xrange(len(y)) ] )
    Y = fft(y)

    Yre = [math.sqrt(Y[i].real**2+Y[i].imag**2) for i in xrange(len(Y))]


    plt.subplot(2, 1, 1)
    plt.plot( x, y )

    plt.subplot(2, 1, 2)
    plt.plot( x, Yre )
    plt.yscale('log')


    plt.show()
```

What do we expect?

- Remember, we're performing the transform as :

$$y_k = \sin\left(\frac{2\pi f}{N} k\right)$$

- There are two features in our data :
  - Overall rise
  - Seasonal trends (12 months)



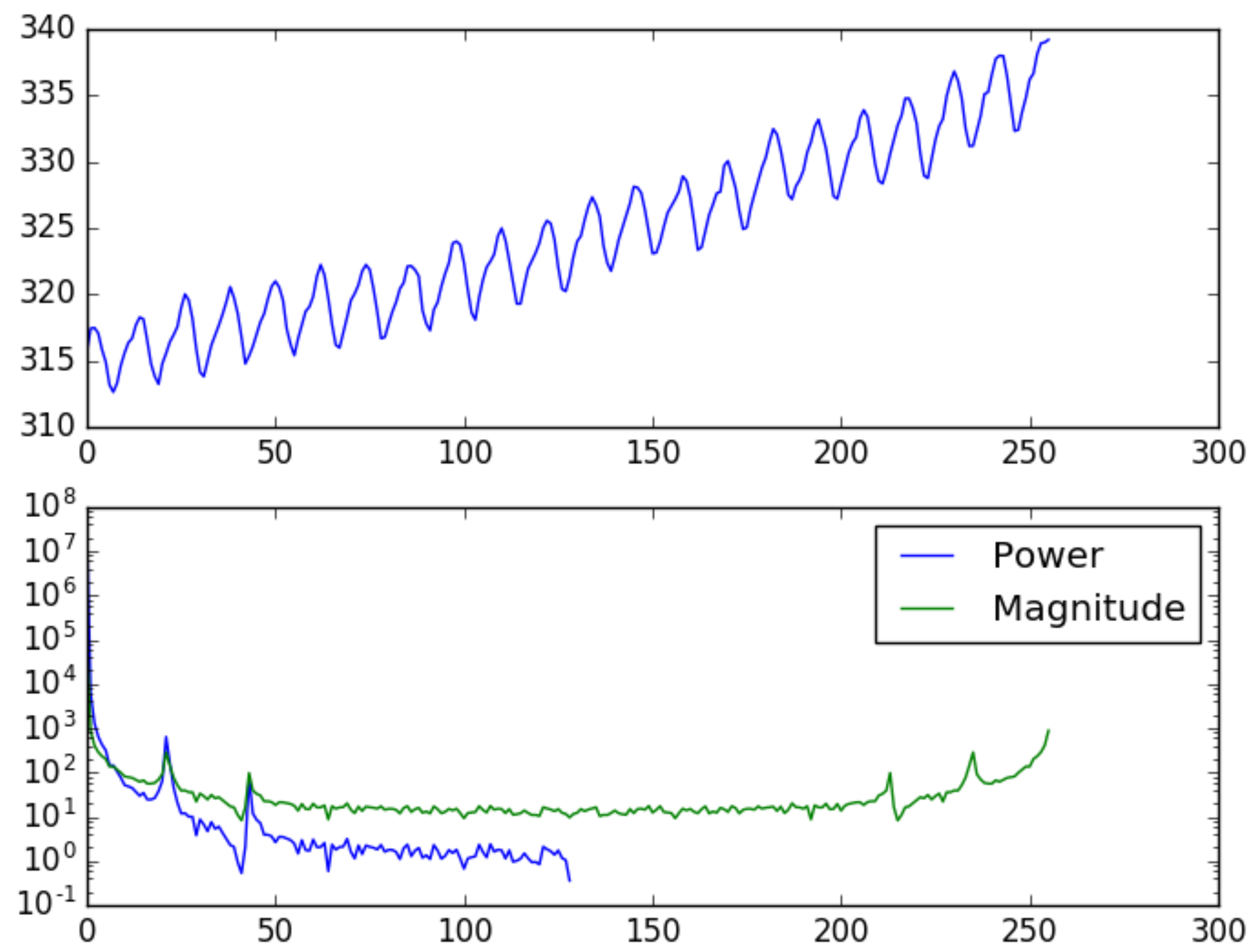
- How will they manifest?
  - Take 5 minutes and think about it!

Power Spectrum

- We want to define the "power spectrum" (or "periodogram")

$$P(\omega_k) = \frac{1}{N}\begin{cases} |g_0|^2 & k=0 \\ \left[|g_k|^2 + |g_{N-k}|^2\right] & k=1,2,\ldots,\frac{N}{2}-1 \\ |g_{N/2}|^2 & k=\frac{N}{2} \end{cases}$$

- This is a better way to represent the "readable" signal, because otherwise it's a complex function that we have to take the magnitude of
- This also has Nyquist frequency issues!



More on FFT's

- What about N != 2^n?
- Signal processing
- Sampling rate

More on FFT's

- First, let's take a look at a generalization of our previous program :
- If the number is even :
  - use the Cooley/Tukey algorithm we discussed last time
- If the number is odd :
  - use the discrete Fourier transform, not the FFT

- The same code works for both! Since it's recursive, it will do the bits that are 2^n quickly and the bits that are not 2^n very, very slowly

Recall : FFT's from Danielson-Lanczos

- Input data in space domain
- break into even and odd bits
- use the recursion relation to solve each half individually

```
fft ( x ) :
    n = size of data
    recursively call fft(even x's)
    recursively call fft(odd x's)
    combine results
```

```
from cmath import exp, pi

def fft(x):
    N = len(x)
    if N <= 1: return x
    even = fft(x[0::2])
    odd =  fft(x[1::2])
    return [even[k] + exp(-2j*pi*k/N)*odd[k] for k in xrange(N/2)] + \
           [even[k] - exp(-2j*pi*k/N)*odd[k] for k in xrange(N/2)]

print fft([1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0])
```

http://rosettacode.org/wiki/Fast_Fourier_transform#Python

Note! This very simple form only works if $N = 2^n$, so be careful!

More on FFT's

- New code, which can handle N odd :

```
def discrete_transform(data):
    """Return Discrete Fourier Transform (DFT) of a complex data vector"""
    N = len(data)
    transform = [ 0 ] * N
    for k in range(N):
        for j in range(N):
            angle = 2 * pi * k * j / N
            transform[k] += data[j] * exp(1j * angle)
    return transform

def fft(x):
    N = len(x)
    if N <= 1: return x
    elif N % 2 == 1:         # N is odd, lemma does not apply
        print 'N is ' + str(N) + ', fall back to discrete transform'
        return discrete_transform(x)
    even = fft(x[0::2])
    odd =  fft(x[1::2])
    return [even[k] + exp(-2j*pi*k/N)*odd[k]  for k in xrange(N/2)] + \
           [even[k] - exp(-2j*pi*k/N)*odd[k]  for k in xrange(N/2)]
```

More on FFT's

- Implement timing from the python code :

Vary N

Time this step!

```
# make some fake data :

N = 1024
f = 10.0

print 'Processing N = ' + str(N)

x = array([ float(i) for i in xrange(N) ] )
y = array([ math.sin(-2*math.pi*f* xi / float(N))  for xi in x ])
#y = array([ xi for xi in x ])
start_time = time.time()
Y = my_fft(y)
print time.time() - start_time, " seconds"

Yre = [math.sqrt(Y[i].real**2 + Y[i].imag**2) for i in xrange(N)]
```

N	time (seconds)
1024	0.037
1023	9.75
1022	4.62
1021	9.51
1020	2.29

More on FFT's

- So what did we see?

- $N = 2^n$: lightening fast

- N odd : snail's pace

- N even : fast, but not remotely as fast as $N=2^n$

- OK, so let's go through that bit reversal thing once again!

FFT's : Go back to $N=2^n$

- Take a concrete example of $N=2^3 = 8$
- Then we have :

$$j = 4j_2 + 2j_1 + j_0$$

$$k = 4k_2 + 2k_1 + k_0$$

- We now define :

$$y_{j+1} = y(j_2, j_1, j_0) \quad Y_{k+1} = Y(k_2, k_1, k_0)$$

- The DFT now becomes :

$$Y(k_2, k_1, k_0) = \sum_{j_0=0}^{1} \sum_{j_1=0}^{1} \sum_{j_2=0}^{1} y(j_2, j_1, j_0) W^{(4k_2+2k_1+k_0)(4j_2+2j_1+j_0)}$$

See Garcia Section 5.2!

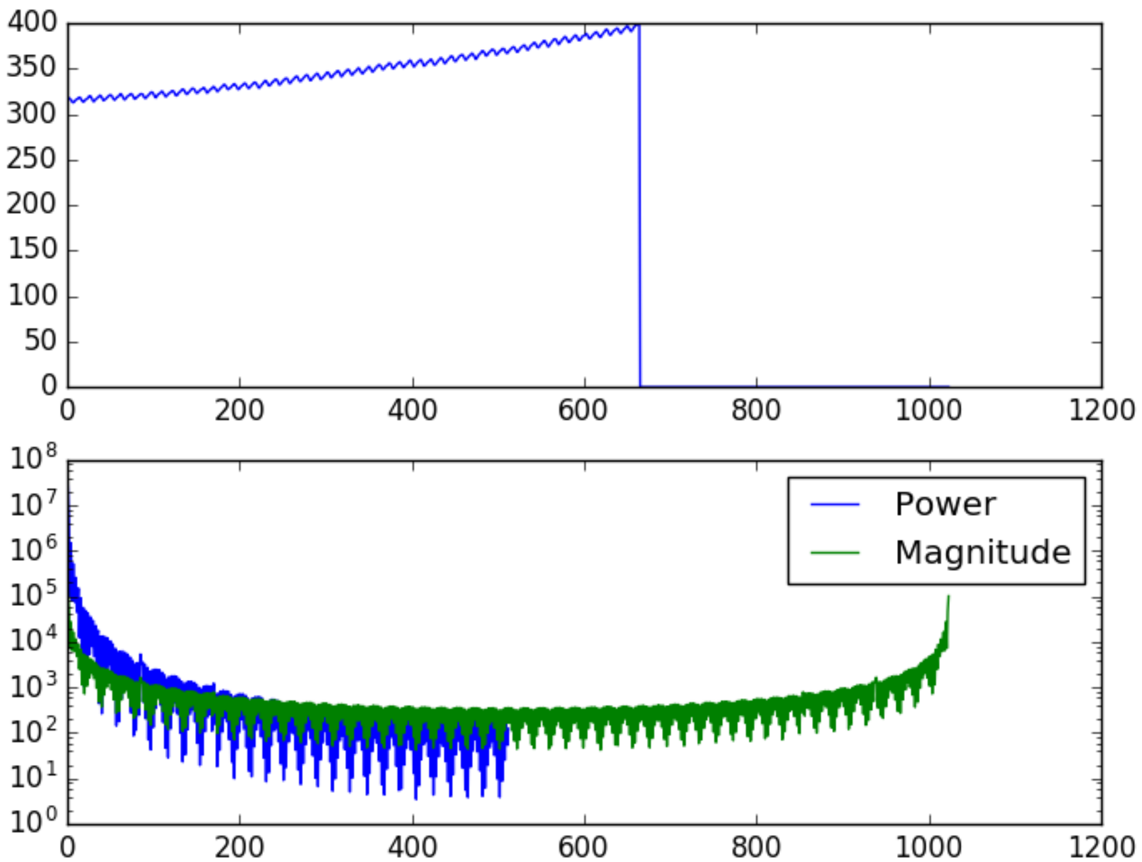
FFT : Tricks and Tips

- So, we've seen that this "bit reversal" magic really does pay off a lot
- What happens if $N \mathrel{!}= 2^n$?
- Well, as we saw, we can solve the problem, but it's complicated
- Trick : if $N \mathrel{!}= 2^n$, then pad with zeros
  - However, then you've actually got to massage the output a bit so you get what you want

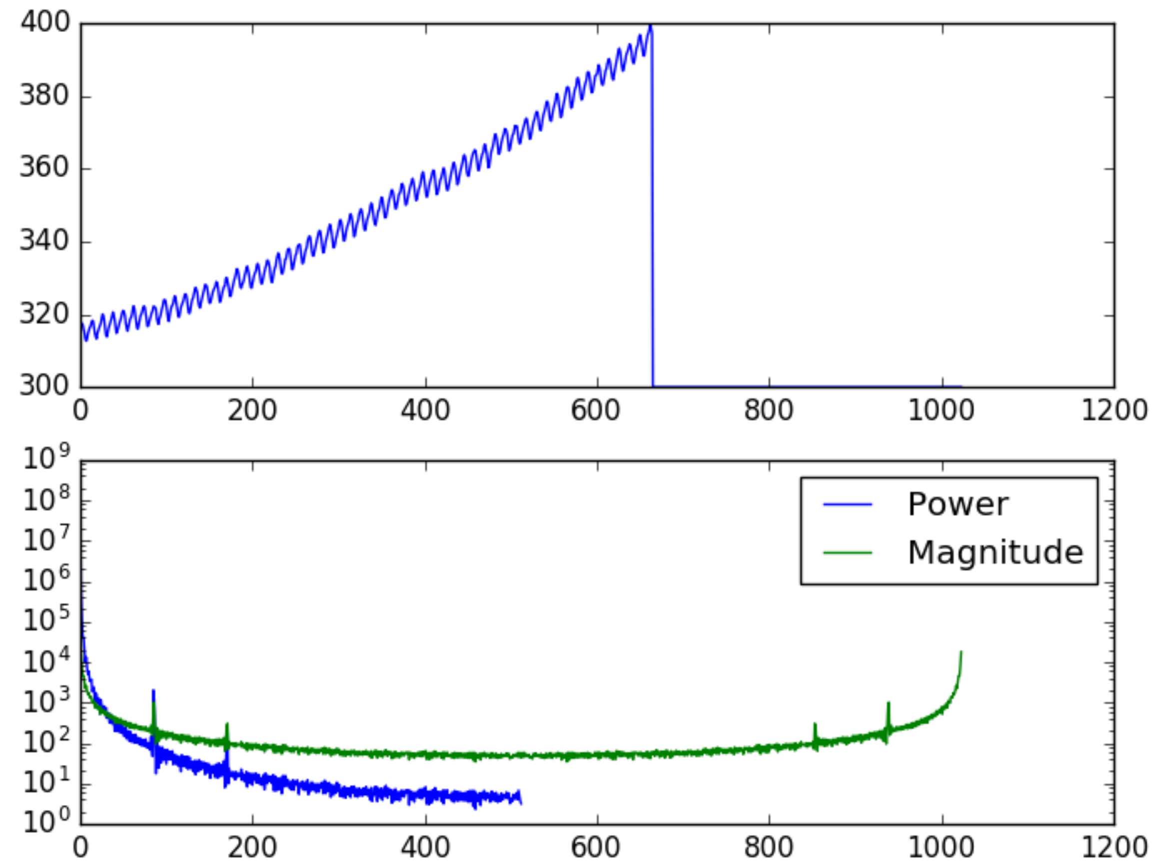
FFT : Padding

- So when we “pad”, what do we really mean here?
- We’re adding a “DC” offset, but can basically pick what we want :

Pad with 0.0 ppm



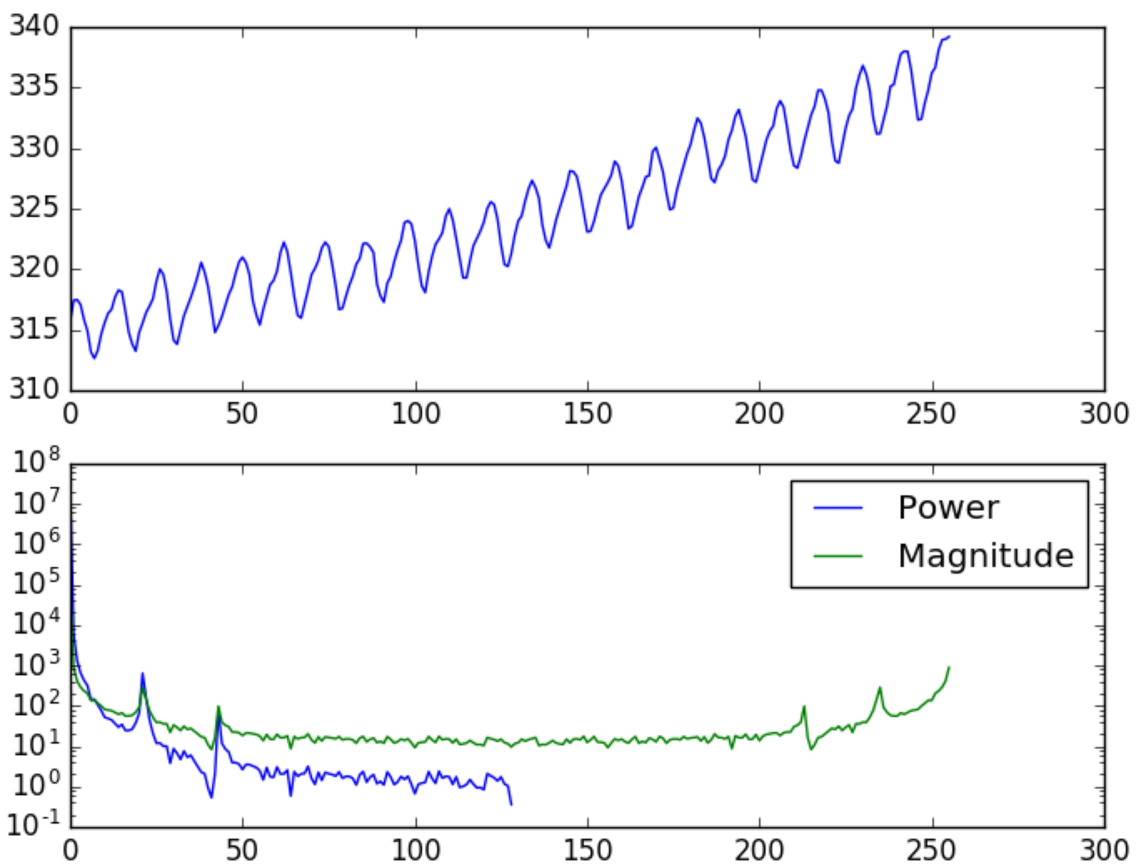
Pad with 300.0 ppm



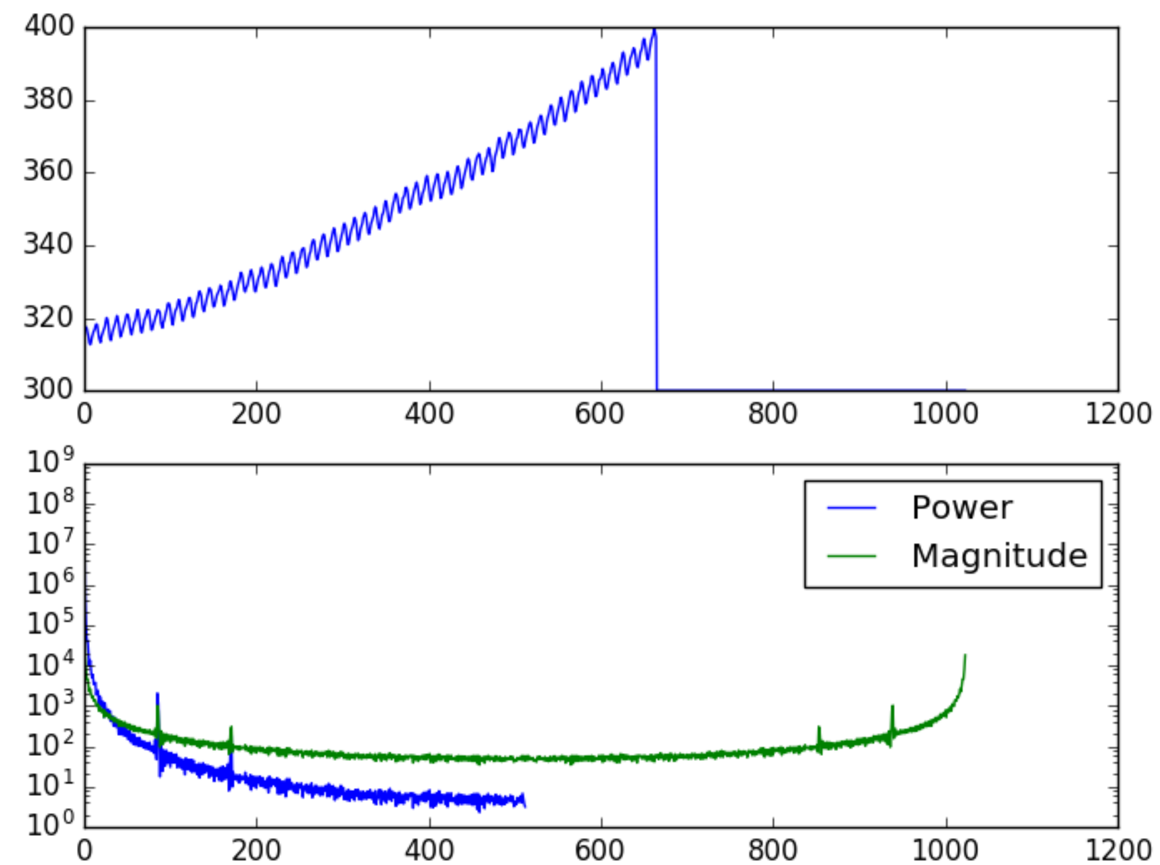
FFT : Padding

- Adding more cycles makes the peaks narrower and sharper, but if you have to pad, it adds these “echoes”

Cutoff series at 256

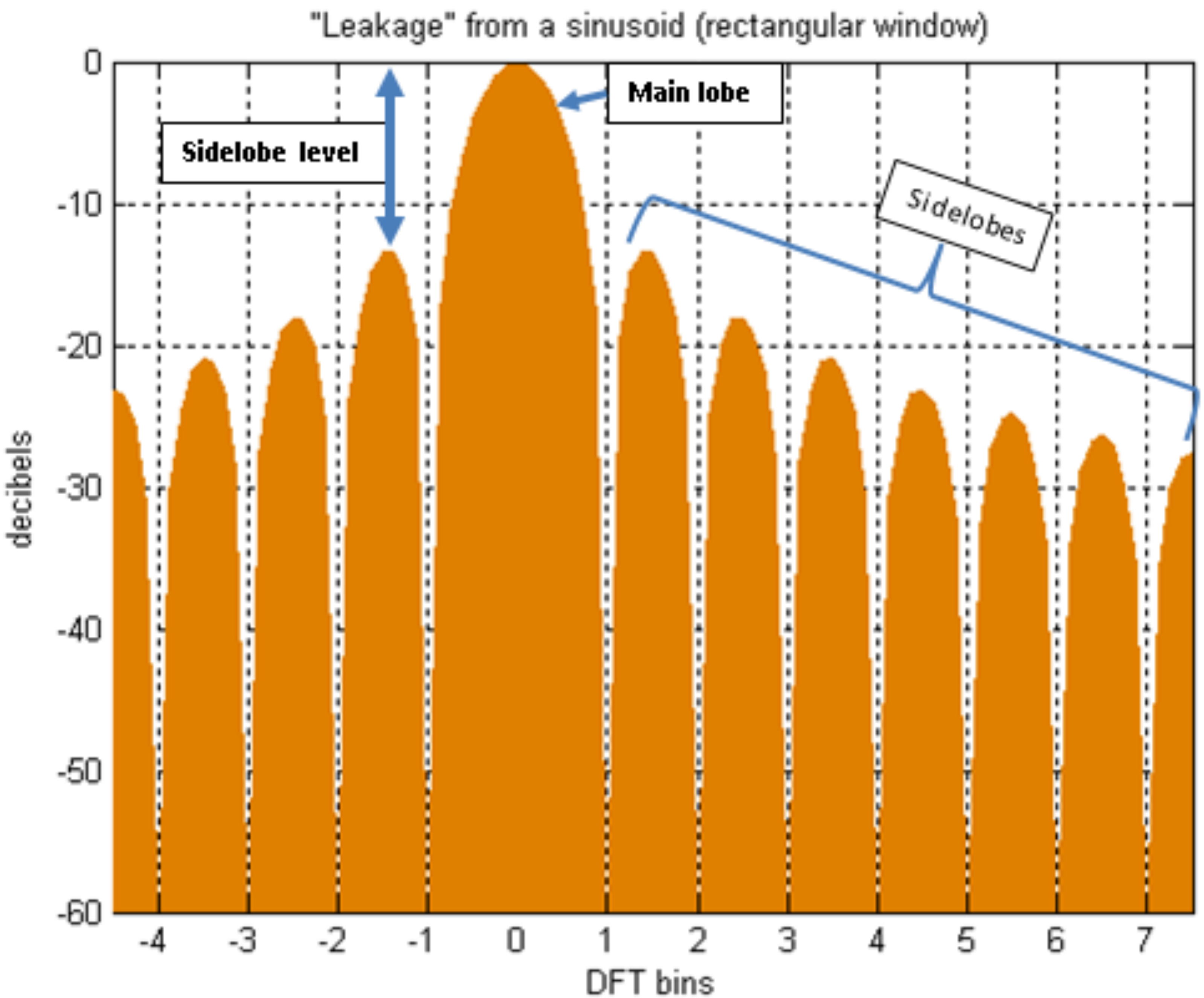


Pad with 300.0 ppm



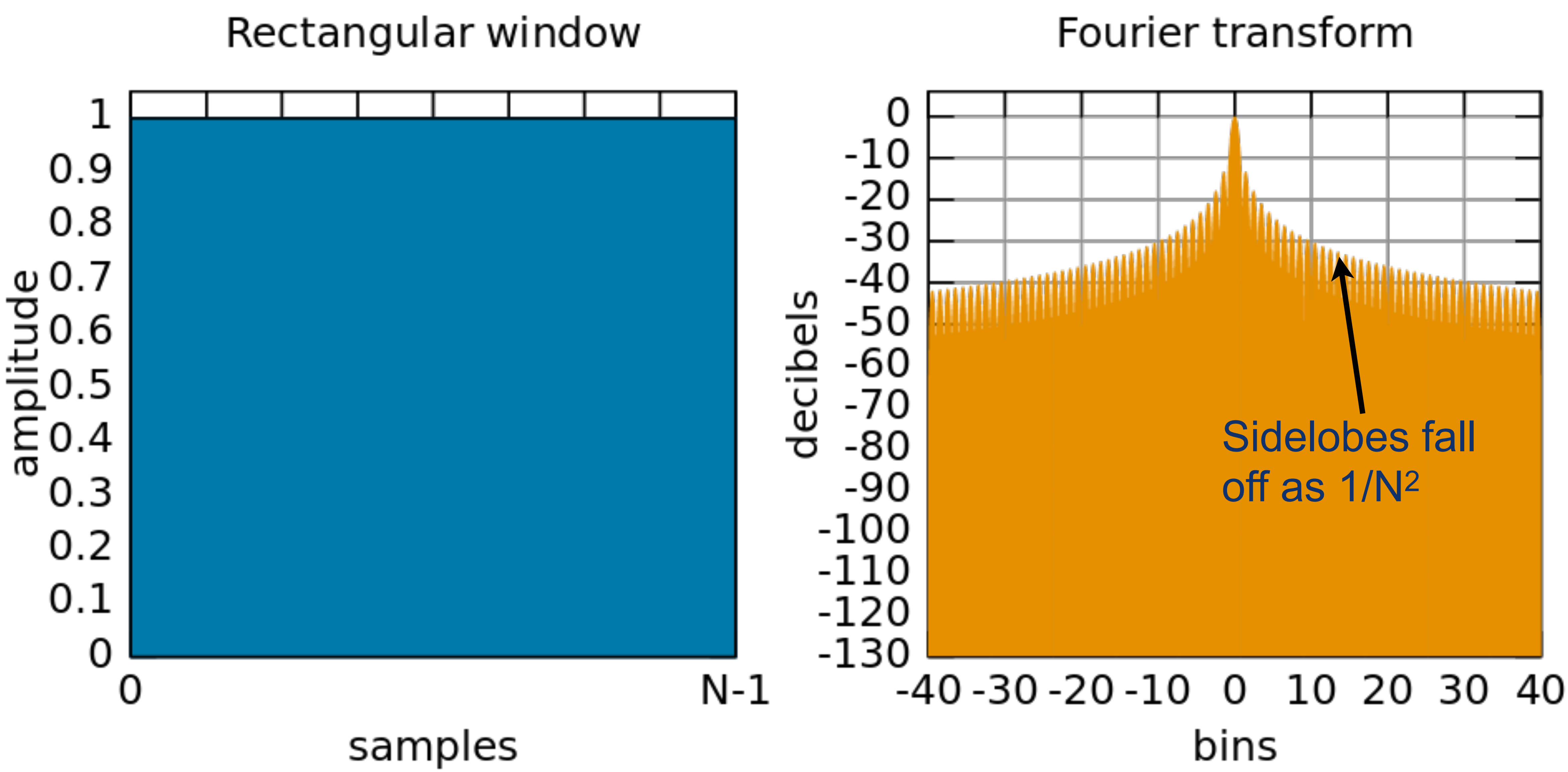
FFT : Windowing

- Can we get rid of this “ringing” ?
- This is related to “windowing” :
  - http://en.wikipedia.org/wiki/Window_function



FFT : Windowing

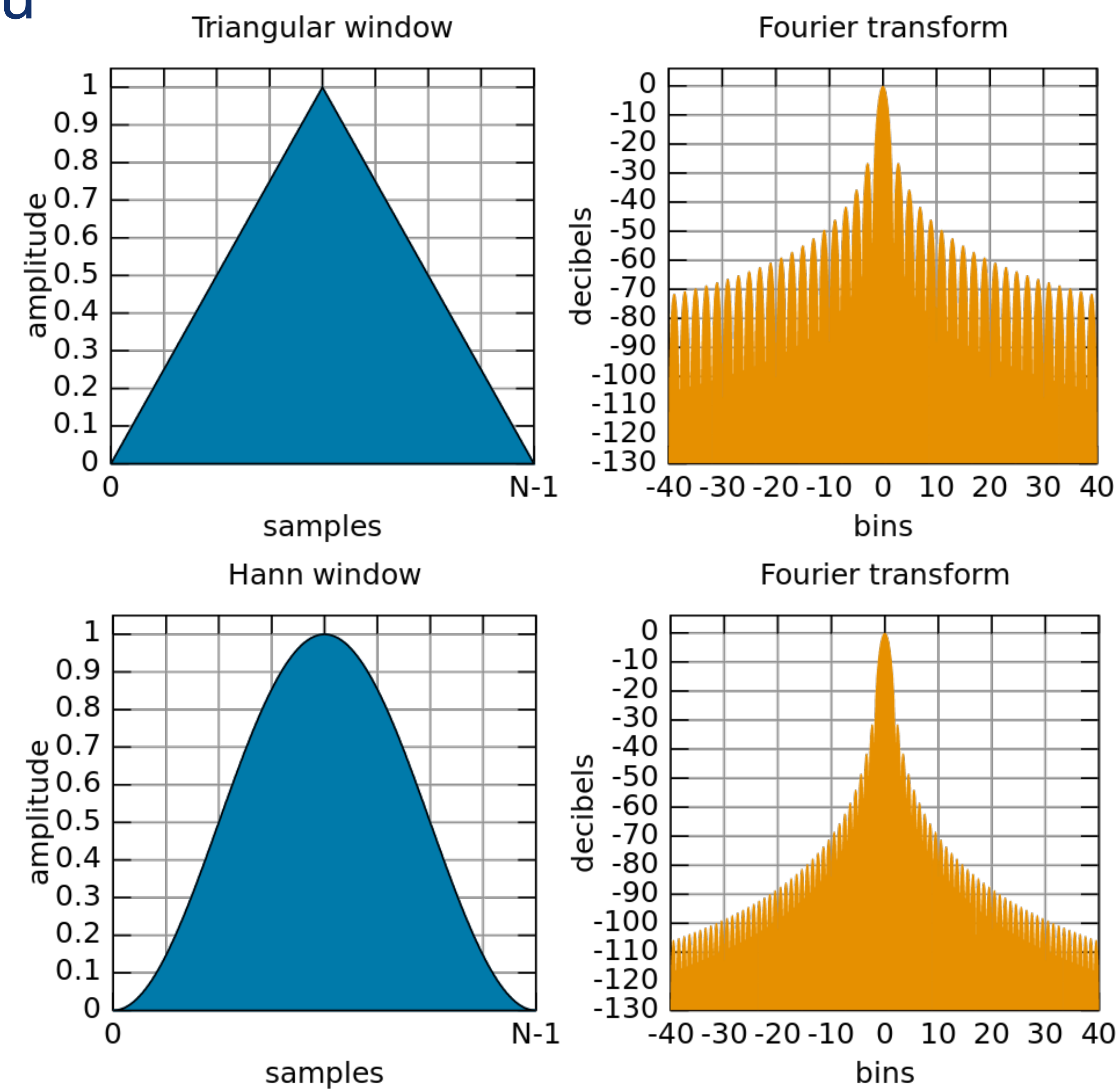
- For the "padding", this is equivalent to a rectangular window cut :



This is the form of the "ringing" you will observe in your transform, convolved with your desired transform!

FFT : Windowing

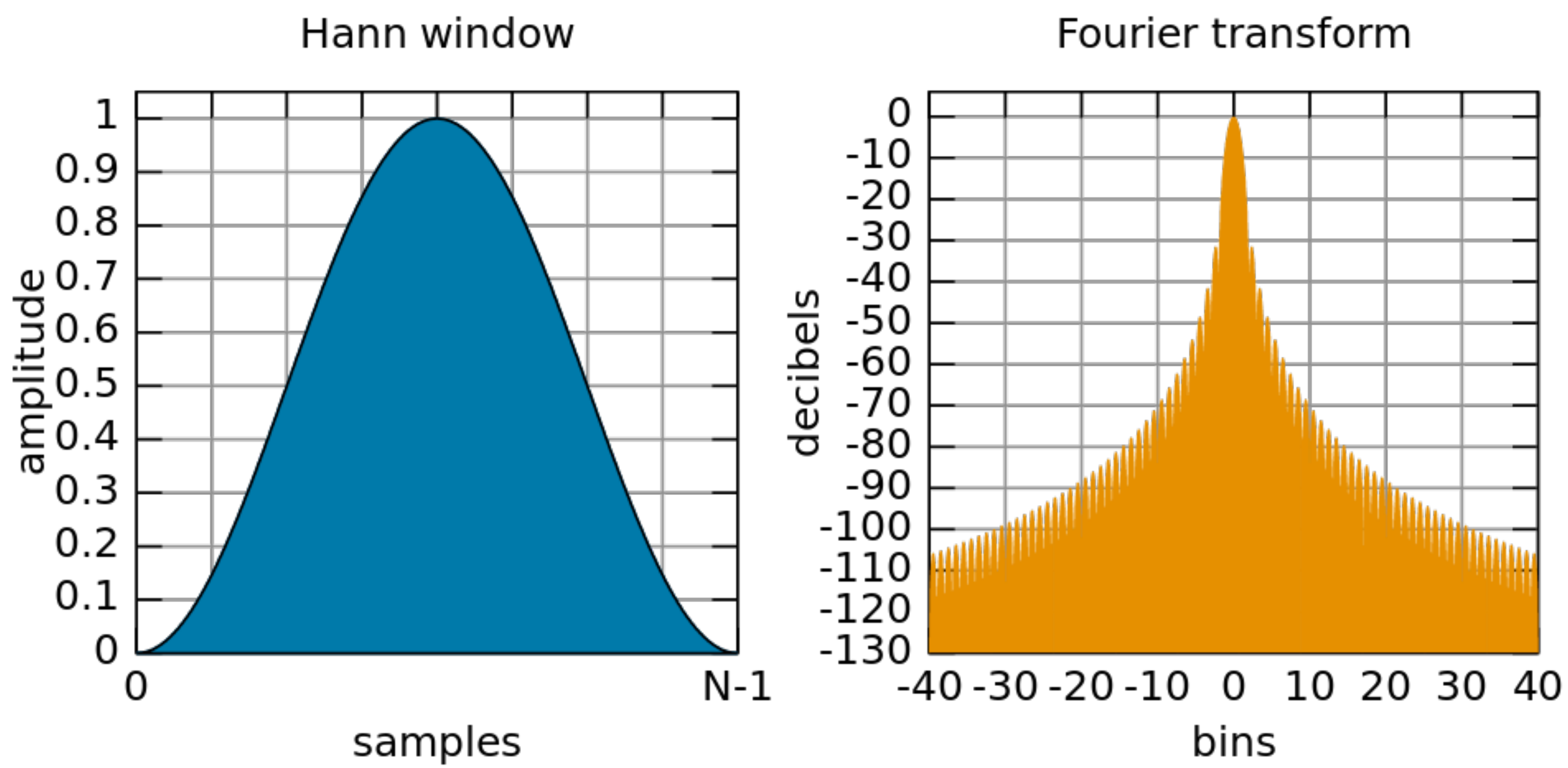
- There are many other possibilities that you may want to try, depending on your application
- Some examples :



FFT : Windowing

- To implement this :
  - You MODIFY the series in the time(/space) domain
  - This manifests in a cleaner signature in the frequency domain
- Example :

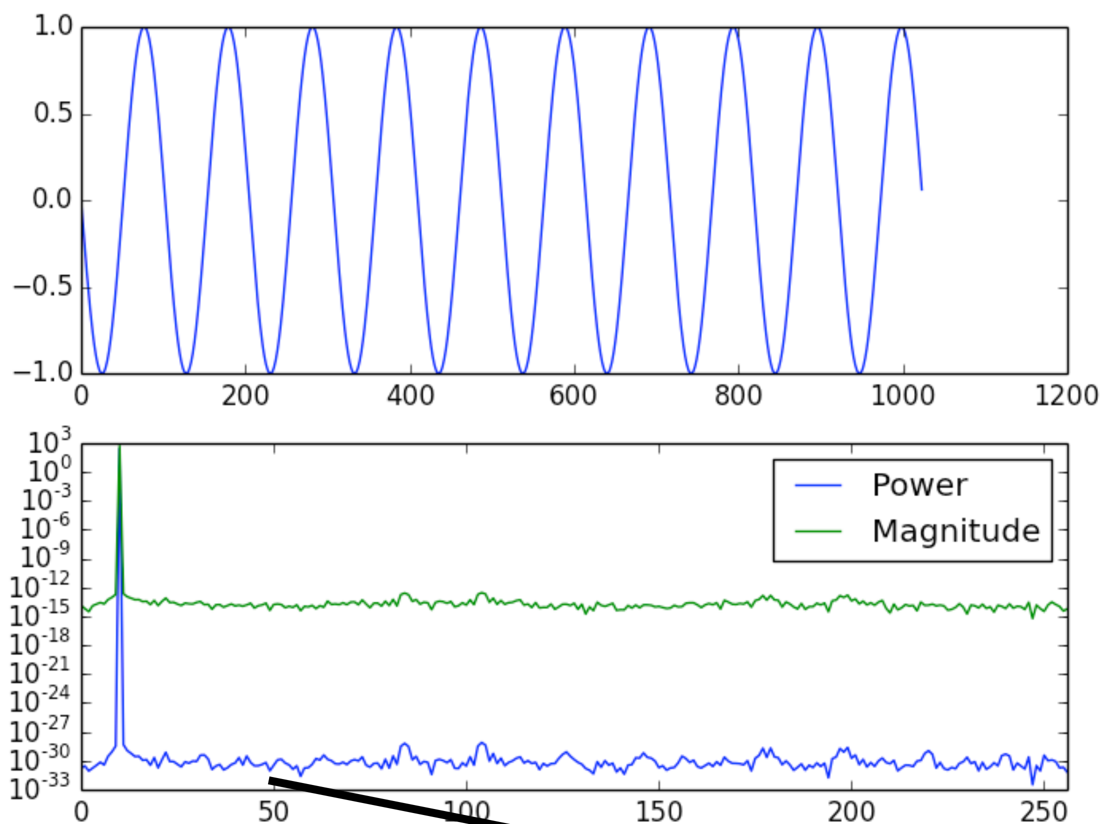
$$g_k = \sum_{j=0}^{N-1} W^{kj} f_j \left[\frac{1}{2} - \frac{1}{2} \cos\left(\frac{2\pi j}{N} \right) \right] .$$



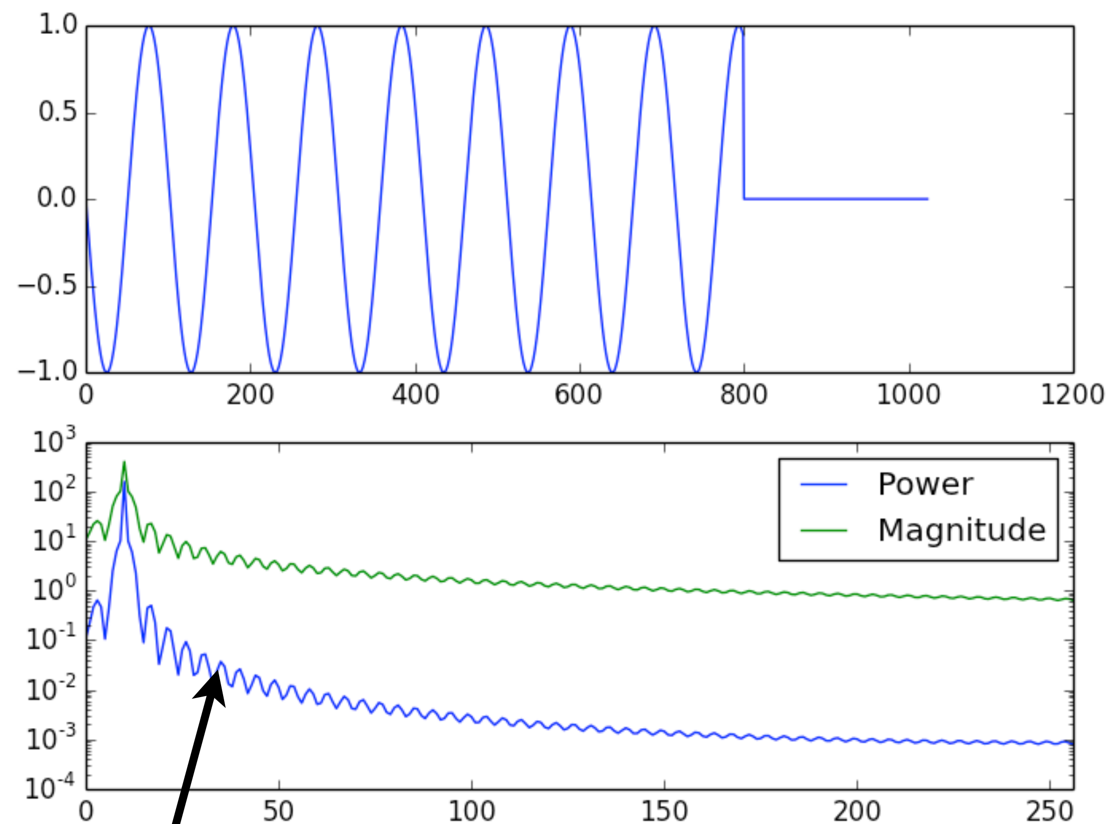
FFT : Windowing

- Take the effect of this from a “clipping” of our simple sinusoidal example

No padding



Padding with no window

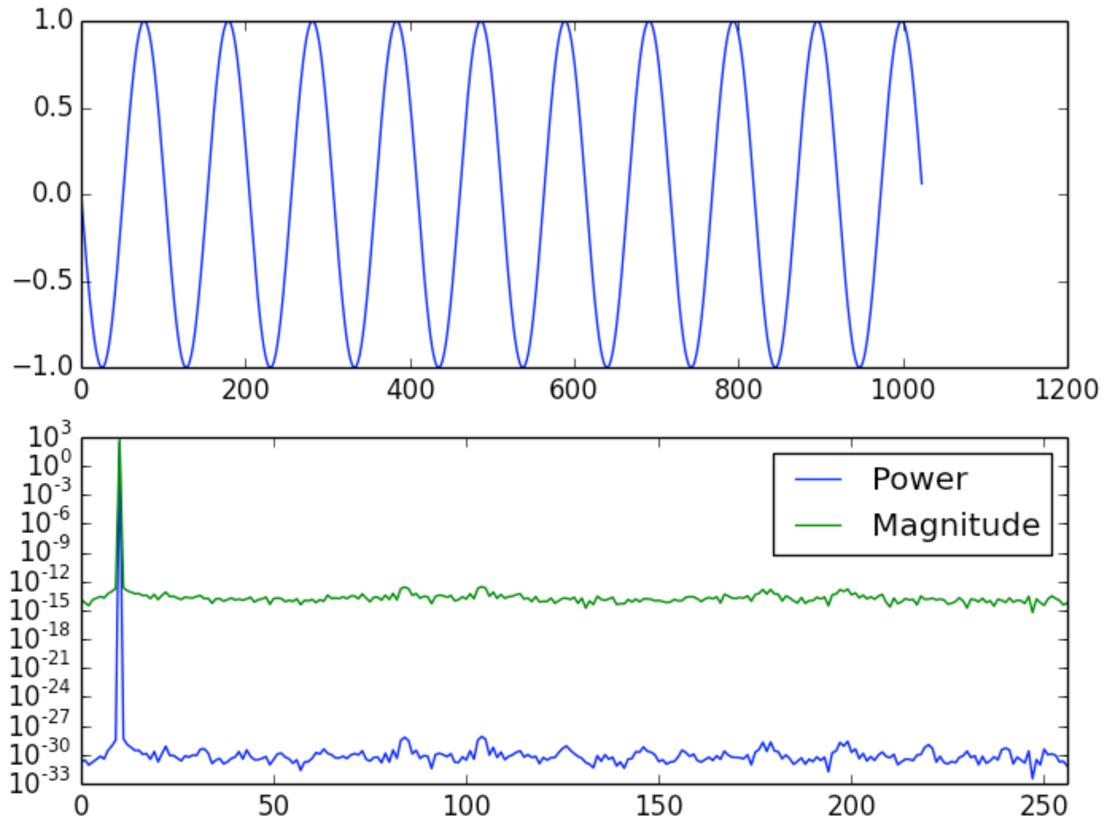


“Ringing” induced from the box window!

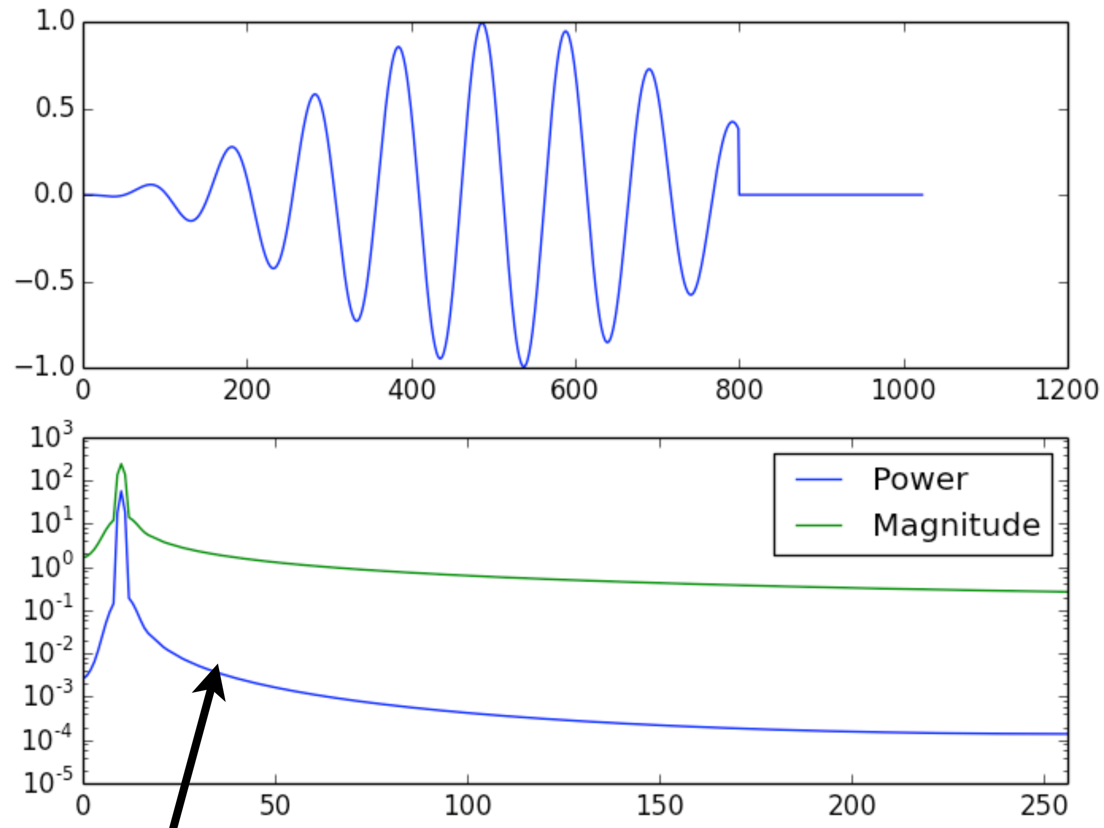
FFT : Windowing

- Take the effect of this from a “clipping” of our simple sinusoidal example

No padding



Padding with Hann window

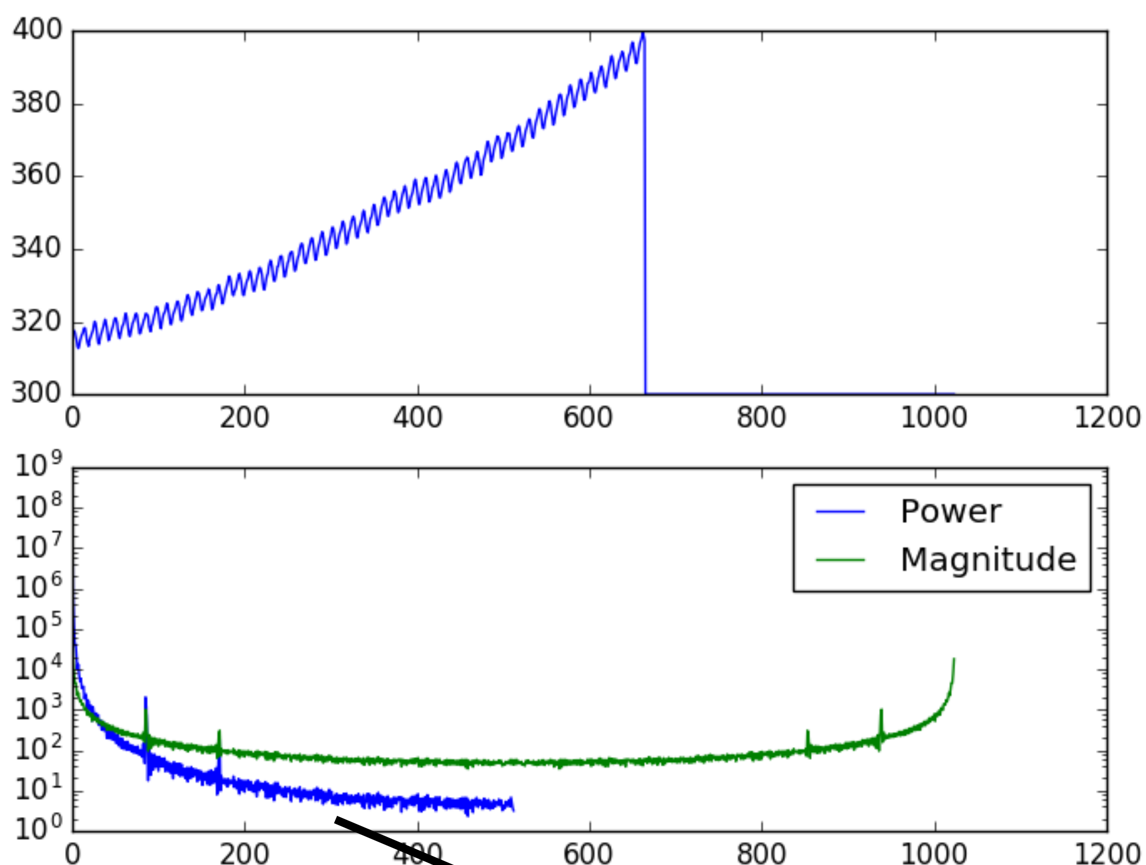


“Ringing” now much reduced!

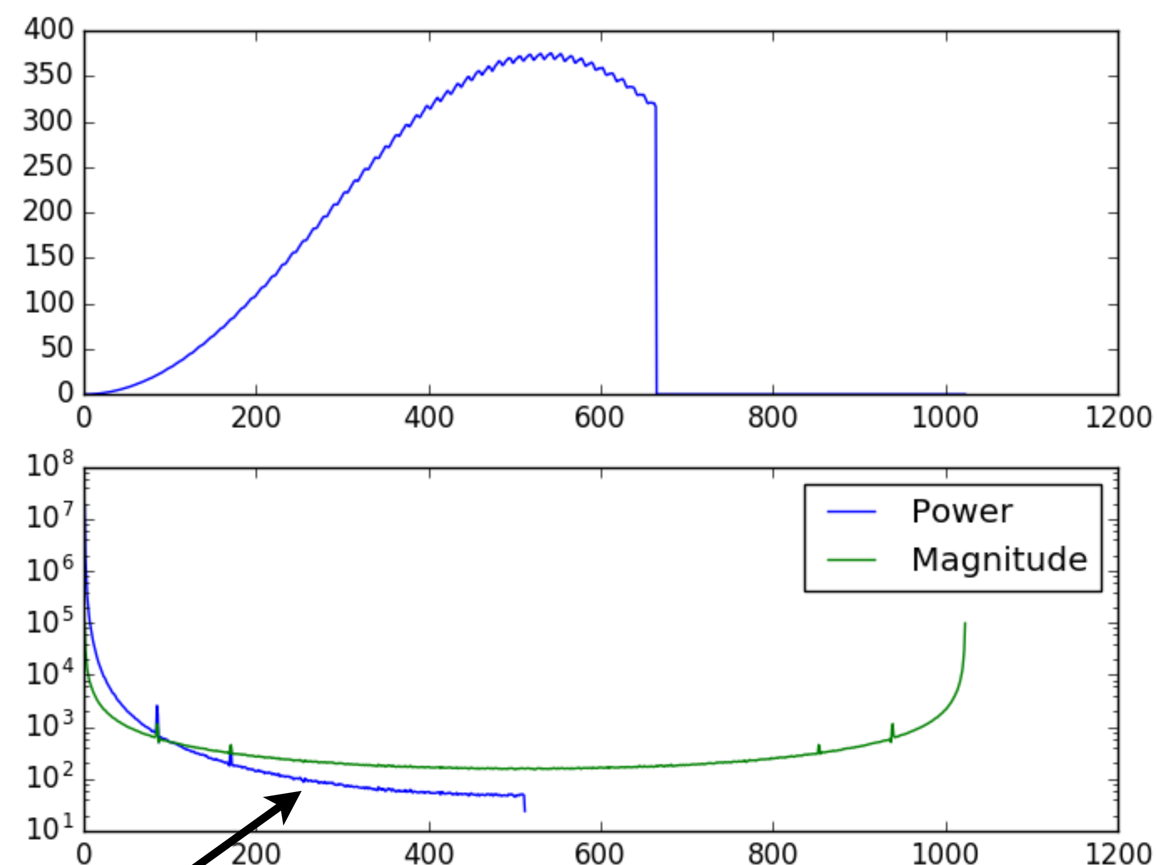
FFT : Windowing

- Now look at our actual CO2 data

Padding



Padding with Hann window



Considerably reduced "ringing" again!

FFT : Windowing

- So, for our example, and the Henn window :
  - From “fft_padding.ipynb”

Window function

```
x = array([ float(i) for i in xrange(N) ] )
if window :
    y = array([ math.sin(-2*math.pi*f* xi / float(N)) *(0.5 - 0.5 * math.cos(2*math.pi*xi/float(N-1)))  + m*xi  for xi in x ])
else :
    y = array([ math.sin(-2*math.pi*f* xi / float(N)) + m*xi  for xi in x ])
```

No window

- Don’t forget! In this case we added a linear term
  - Can “window” on this or not, if you want, but it depends on the use case

Inverse FFT

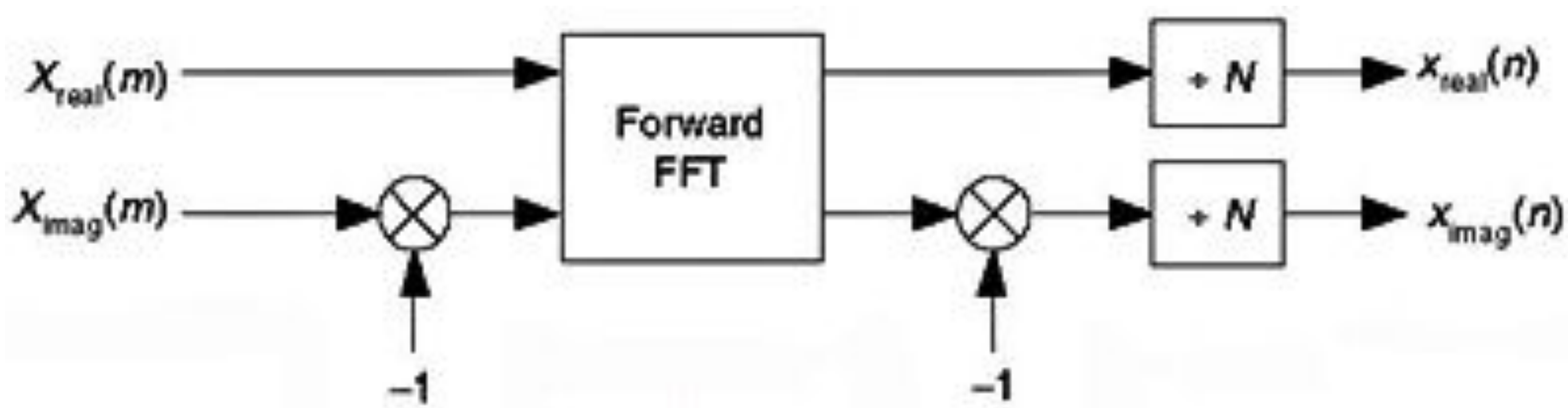
- In order to get your signal properly "cleaned up", we need to also know the inverse Fourier transform (IFT):

F.T. $$g_k = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} W^{kn} f_n \ ,$$

I.F.T. $$f_n = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} W^{-nk} g_k \ ,$$

Inverse FFT

- A few tricks to compute this :
  - http://www.embedded.com/design/embedded/4210789/DSP-Tricks--Computing-inverse-FFTs-using-the-forward-FFT
- The easiest way :



Inverse FFT

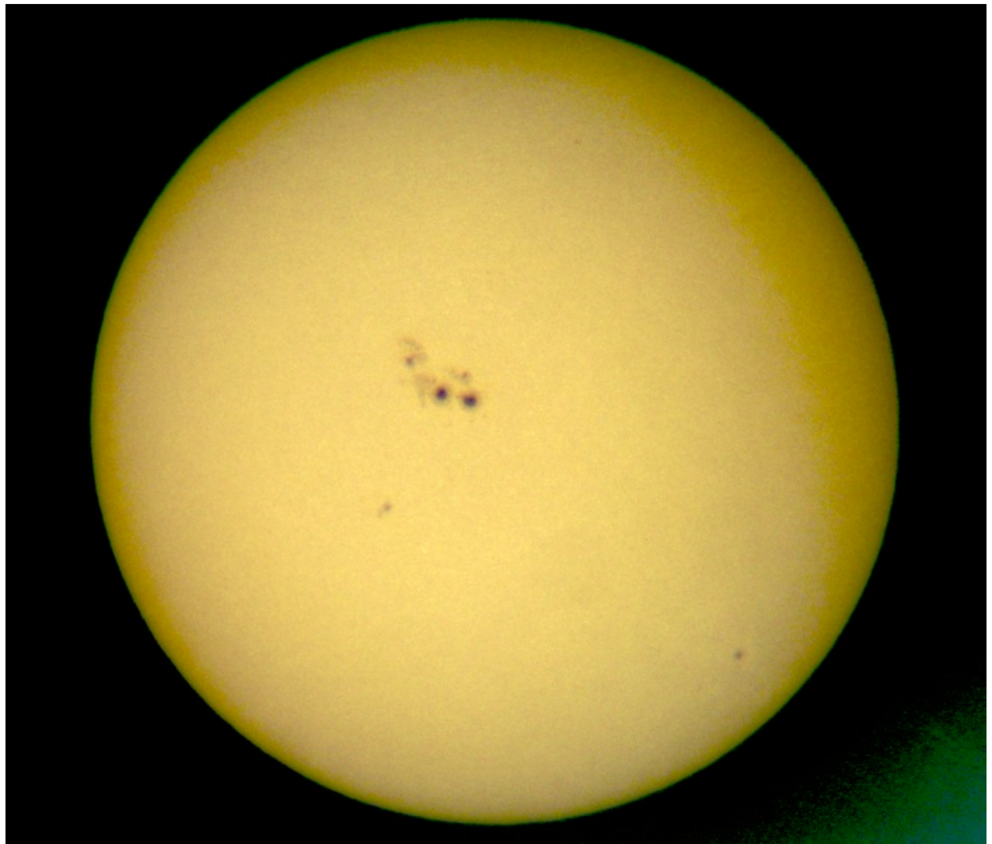
- So, in pseudocode :
  - Compute conjugate
  - Compute FFT
  - Compute conjugate again
  - Divide by N

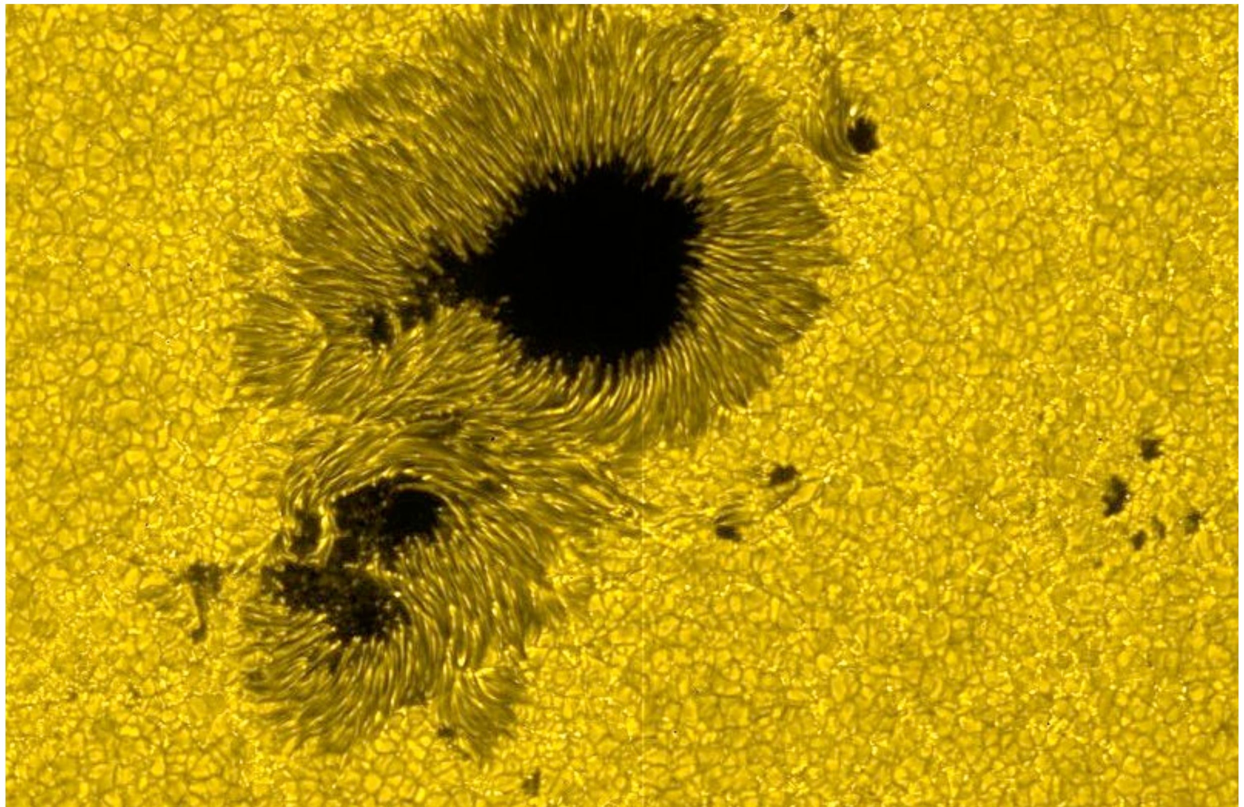
- In python :

```
def ifft(x) :
    # conjugate the complex numbers
    x = conj(x)

    # forward fft
    X = fft( x );

    # conjugate the complex numbers again
    X = conj(X)

    # scale the numbers
    X = divide(X, len(X))

    return X
```

Finally back to some physics

- Can also use the FFT to take a look at sunspots
- They have been known for a long time (364 BC from comments from Chinese astronomer Gan De
- Magnetic activity causes a temperature decrease locally, manifests in a slightly darker spot

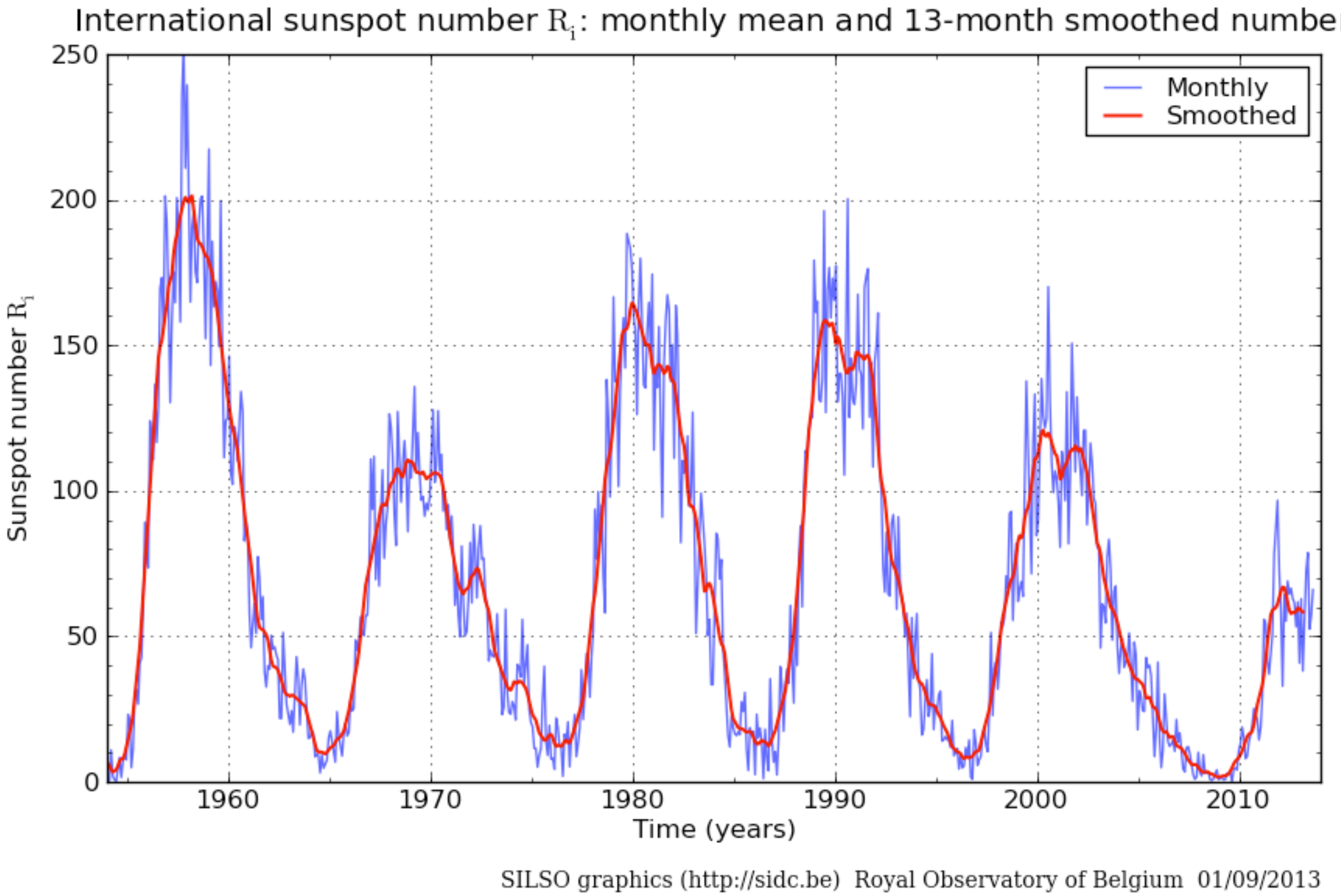




Sunspots

- Can get some data on sunspots from the SIDC (Solar Influences Data Analysis Center):
  - http://sidc.be
- Can find some data :https://www.sidc.be/silso/newdataset
- For instance :

http://www.sidc.be/sunspot-index-graphics/wolfmms.php



Sunspots

- To get the data :
  - http://www.sidc.be/DATA/monthssn.dat
- The format is :
- Looks like :

Year+Month	(in decimal)	Sunspot number	Sunspot number (smoothed)
174901	1749.049	58.0	
174902	1749.129	62.6	
174903	1749.210	70.0	
174904	1749.294	55.7	
174905	1749.377	85.0	
174906	1749.461	83.5	
174907	1749.544	94.8	81.6
174908	1749.629	66.3	82.8
174909	1749.713	75.9	84.1
174910	1749.796	75.5	86.3
174911	1749.880	158.6	87.8
174912	1749.963	85.2	88.7
175001	1750.048	73.3	89.0

Sunspots

- So let’s have some fun with that!
- Say we want to have the data, but get rid of the high-frequency jiggles
- This is not the smoothing that they apply (they apply a Kalman filter) but we’ll use the FFT, a transform, and the IFFT instead

Hands on!

- Sunspots!

- Exceptions :
  - http://docs.python.org/2/tutorial/errors.html
  - http://docs.python.org/2/library/exceptions.html#bltin-exceptions