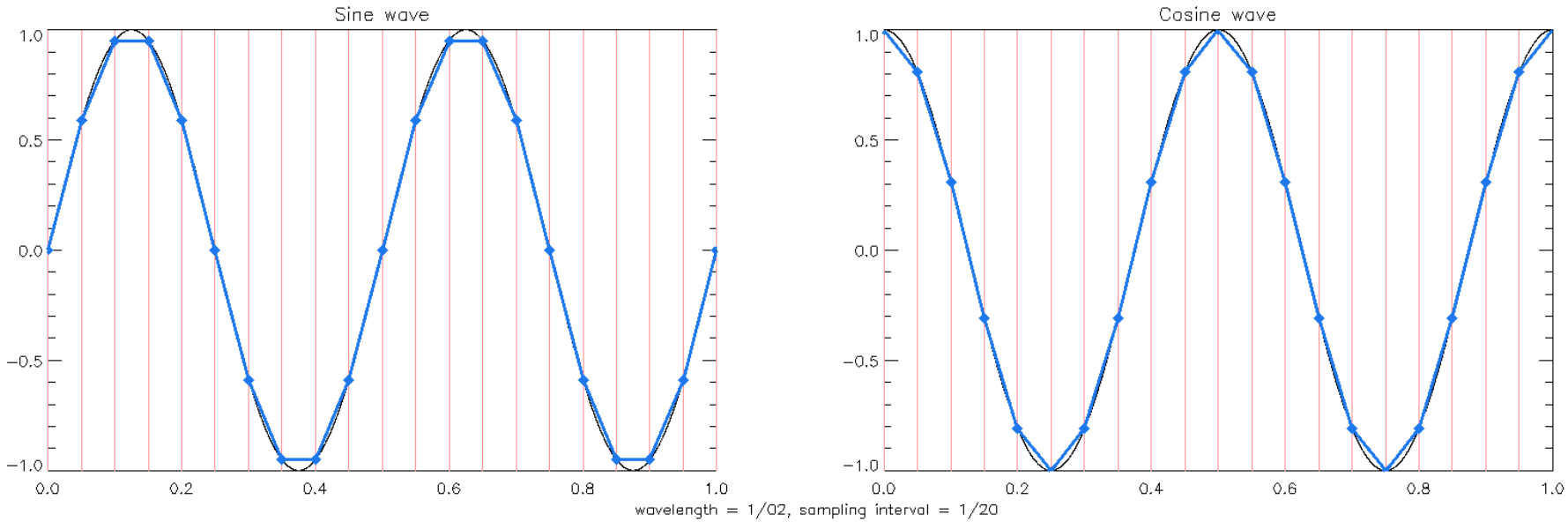
ENVS422: Data Analysis of Environmental Records Week 4

Sine/cosine waves, aliasing, and Fourier transforms

Sampling of sine and cosine waves

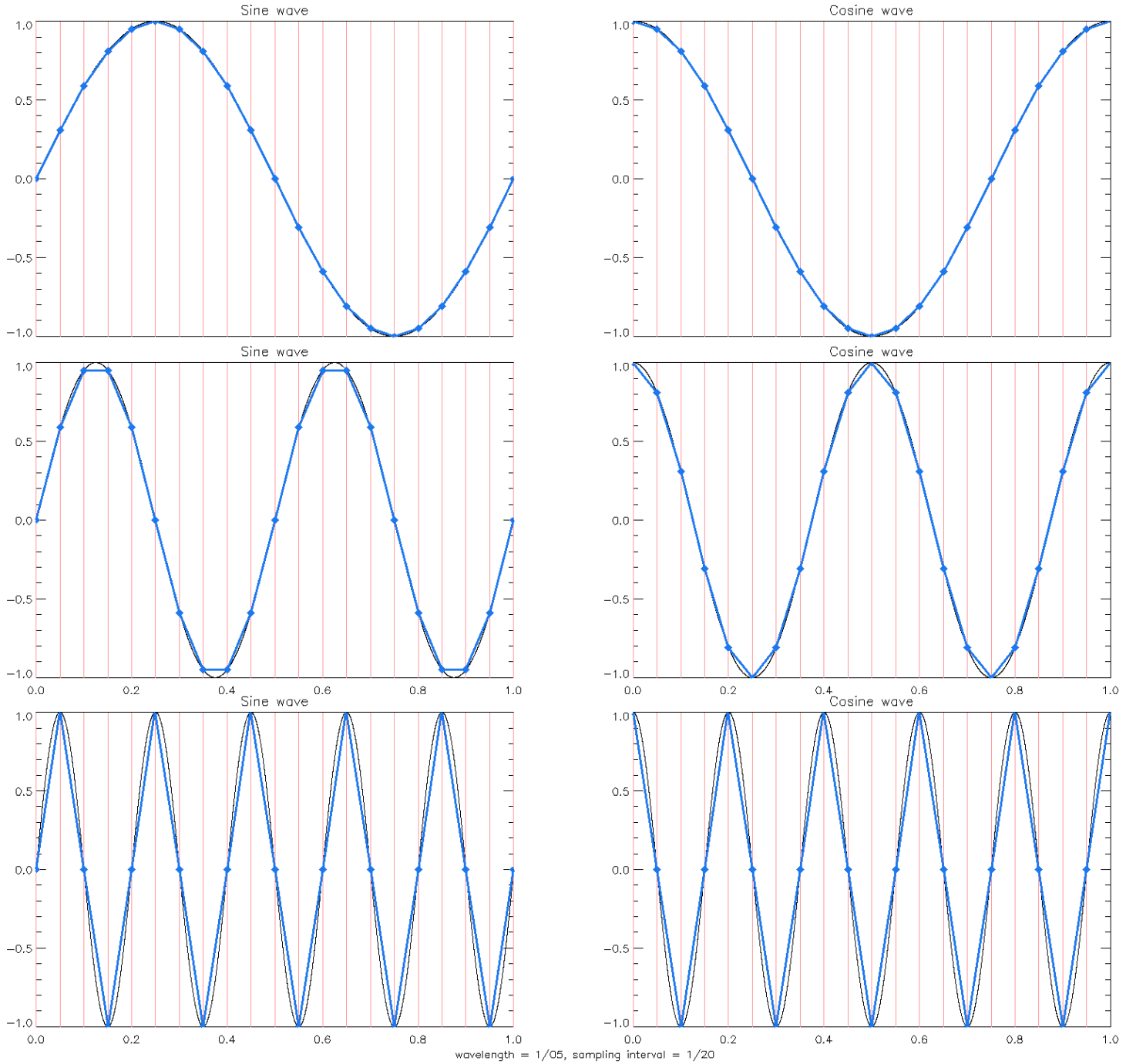


Black line shows a perfect sine or cosine wave

Pink lines show points in space (or time) where measurements are made

Blue line shows subsampled sine or cosine wave

We will stick with a constant sampling interval of $1/20^{th}$ of the total range, and see what happens as we change the wavelength



1

Very good

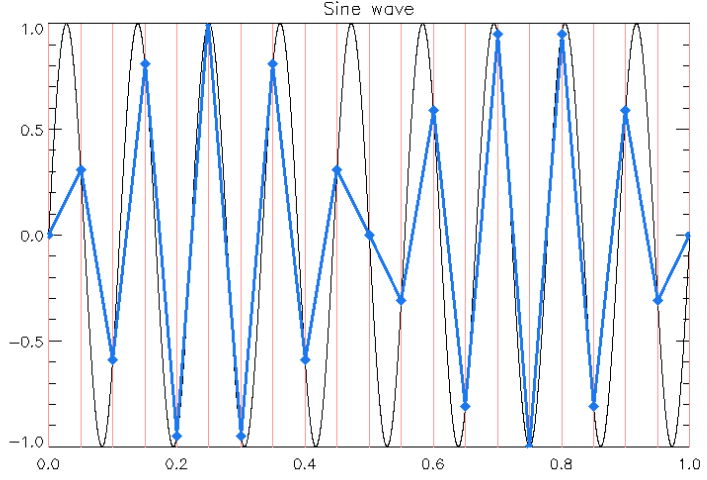
2

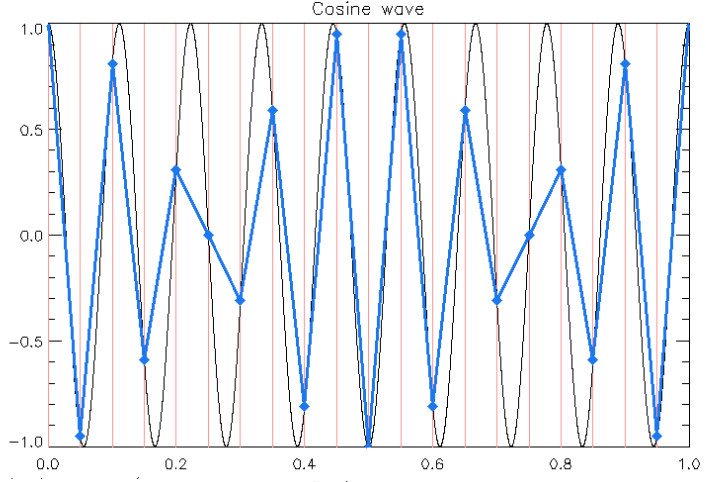
Good

5

Looking spiky

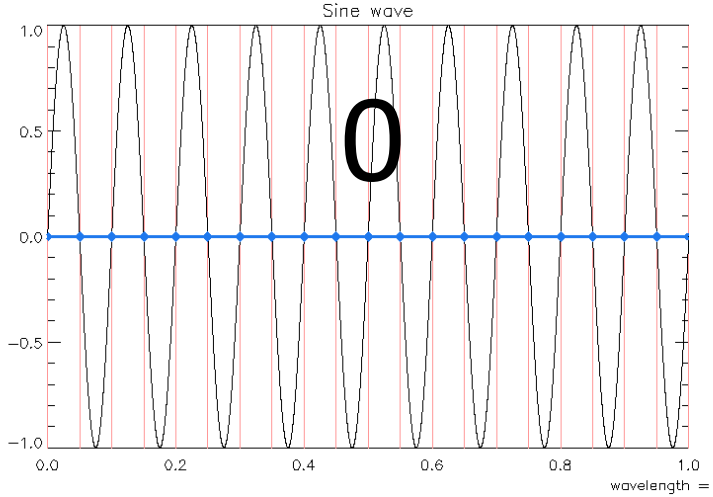
9

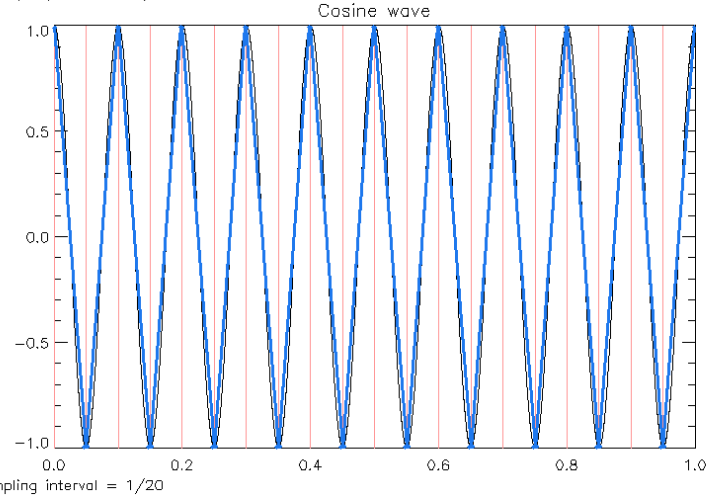




Very spiky, but still the right number of peaks and troughs (9)

0

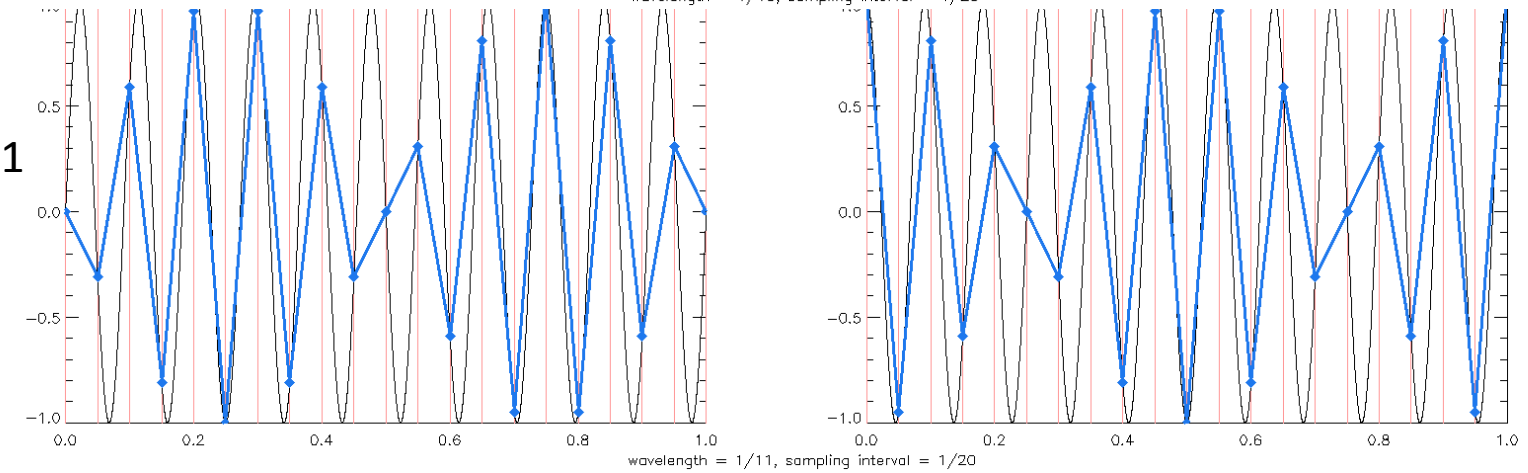




Hmm…

sample spacing = half the wavelength “Nyquist”

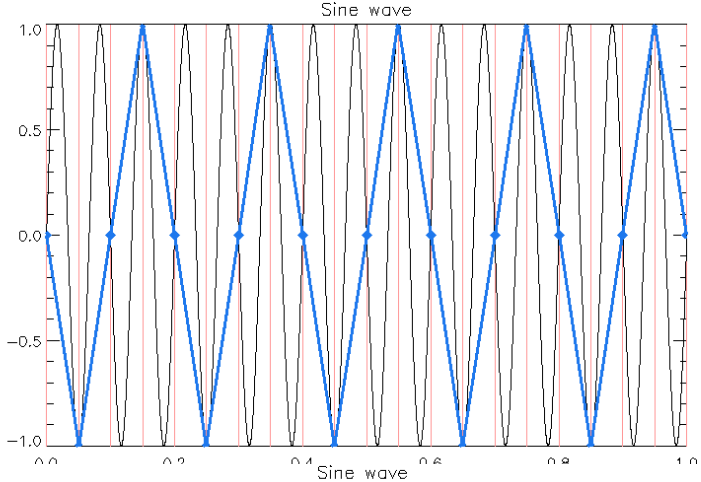
11

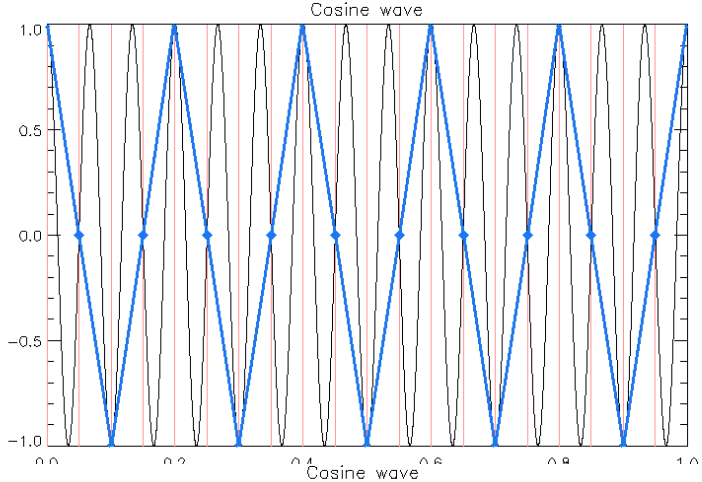


Should be 11 peaks and troughs, but only 9.

(11) looks like ±**(9)**!

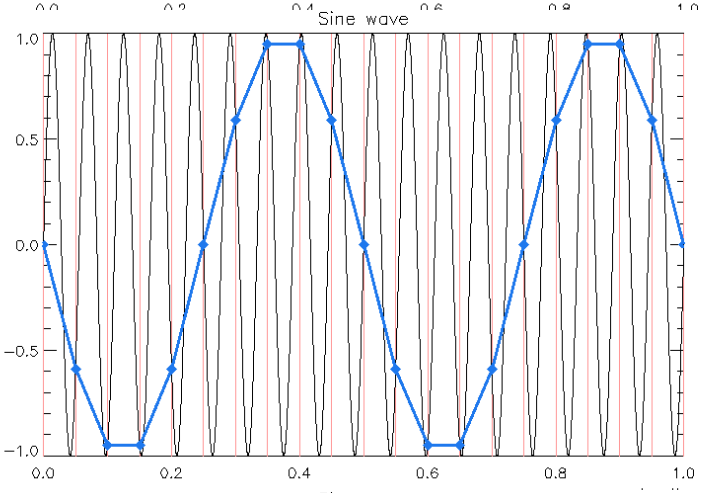
15

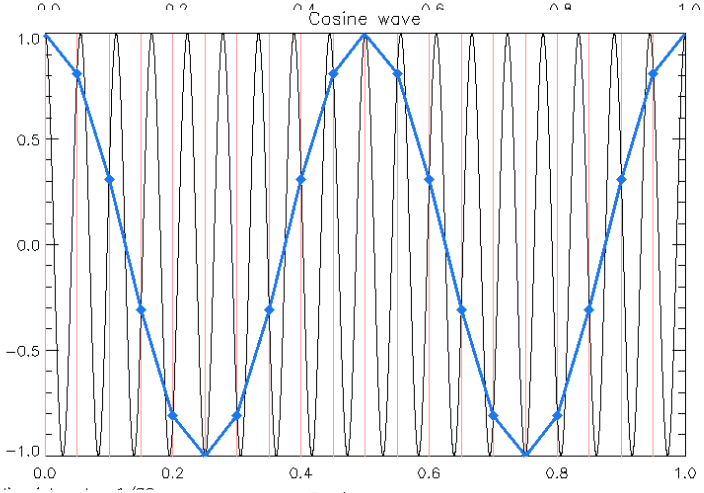




(15) looks like ±**(5)**

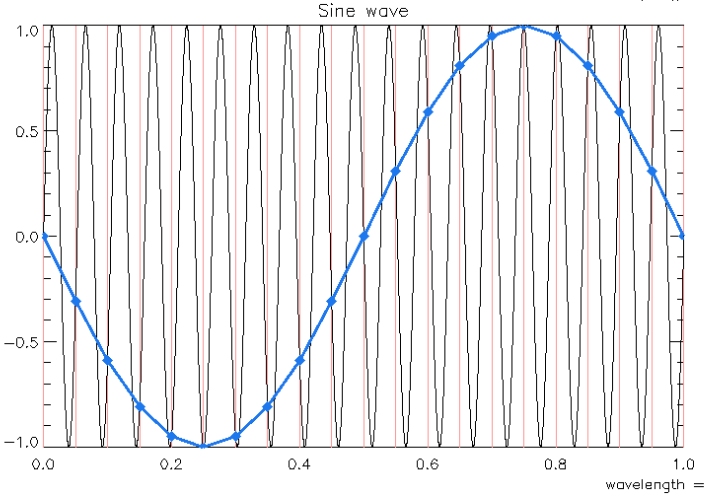
18

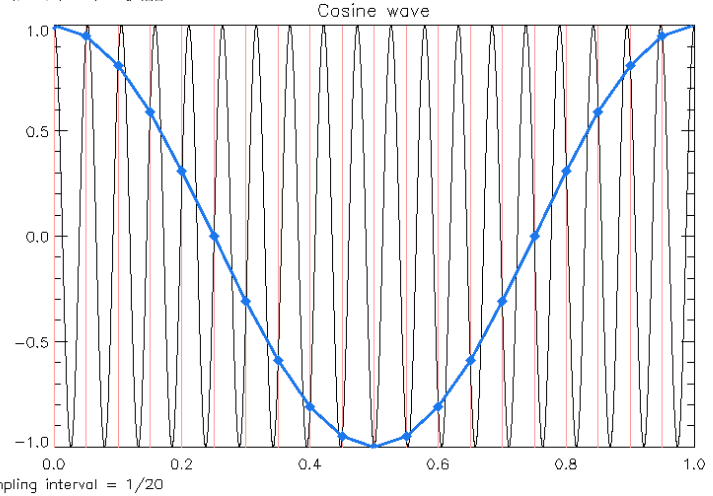




(**18**) looks like ±**(2)**

19

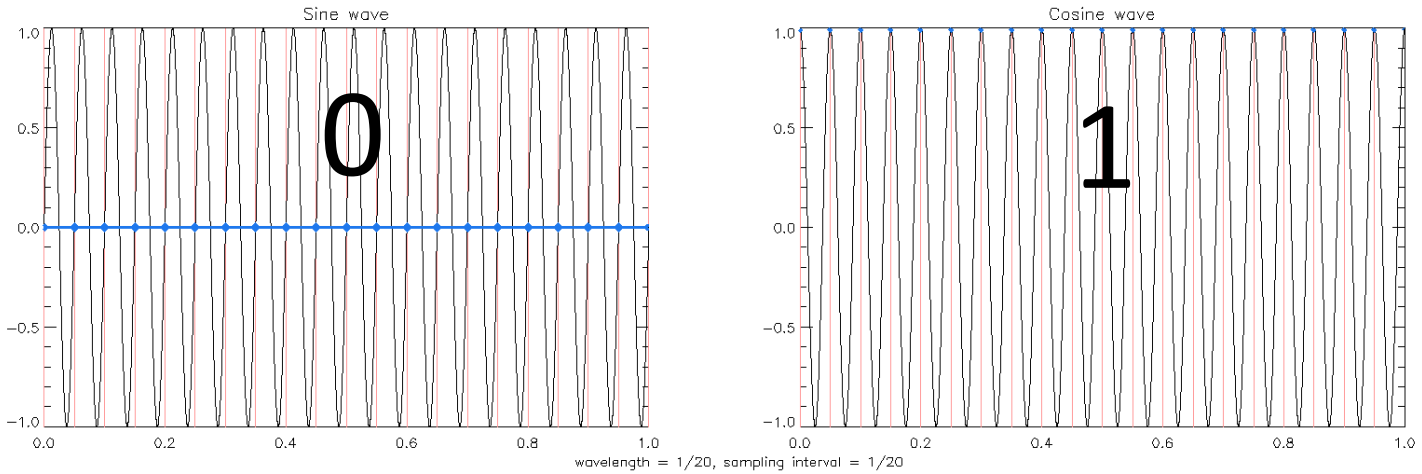




(19) looks like ±**(1)**

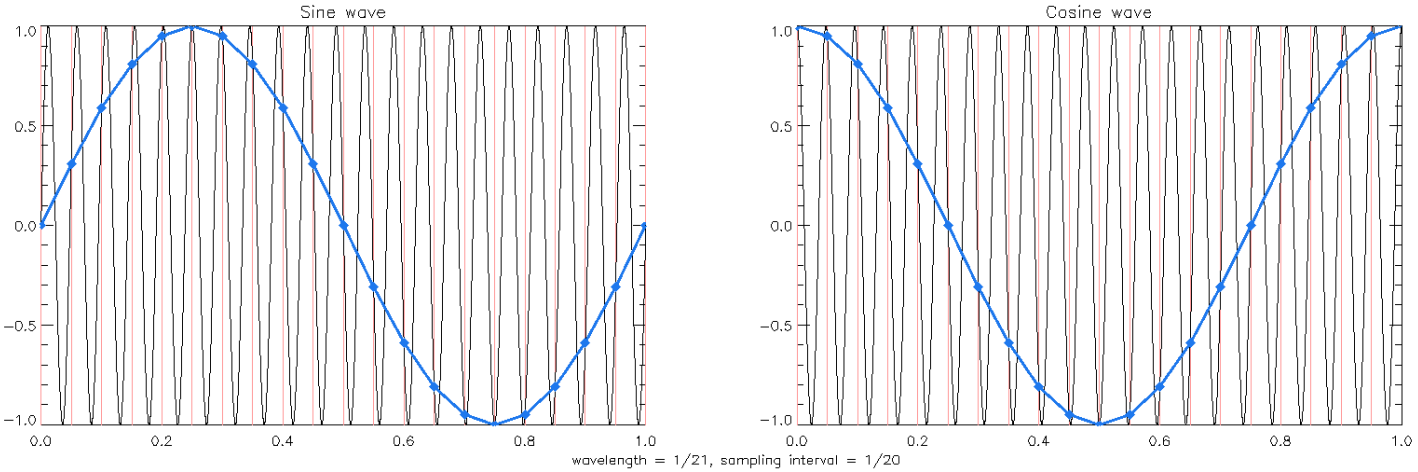
wavelength = 1/19, sampling interval = 1/20

20



(20) is like **(0)**, i.e. sine wave becomes zero, cosine becomes 1

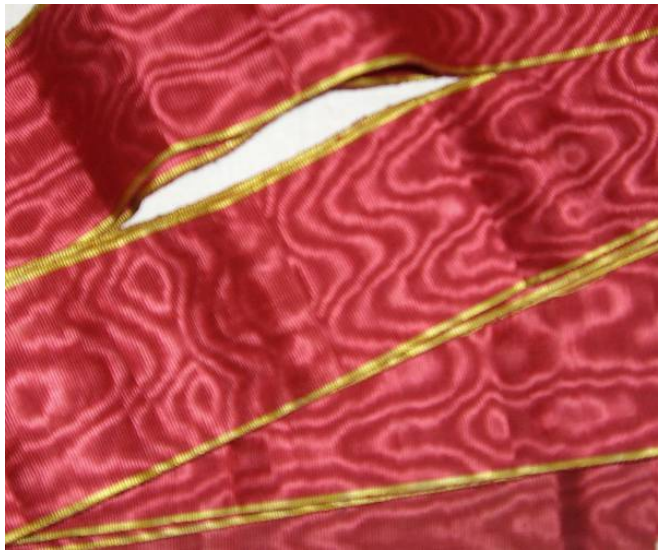
21



(21) is now the same as **(1)** or ±**(19)**

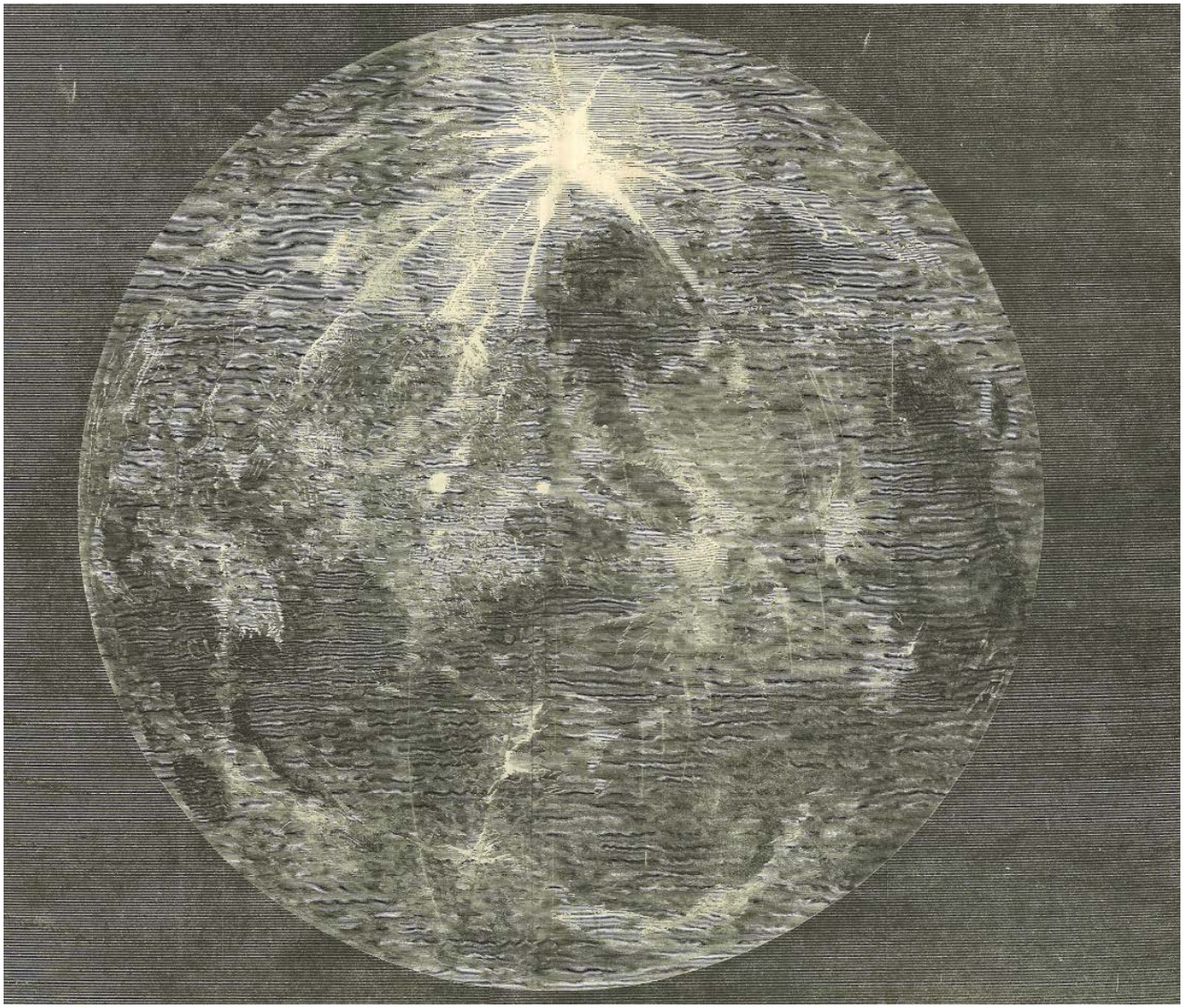
Aliasing

- This phenomenon - a short period or wavelength appearing as a long one because of the sampling - is called aliasing.
- You see it in Moiré silk (2 layers with virtually the same thread spacing, producing large scale patterns).
- Also on TV when people wear clothes with fine stripes.
- Also in satellite data when regular sampling separated by longer than the tidal period means tides appear at long periods (e.g. the Jason altimeter aliases semidiurnal tides to around 60 days).



A woodcut picture of the moon: shading using finely-spaced lines

How it appears when subsampled – aliasing of stripes produces large stripes



Smooth over the stripes before subsampling – no aliased stripes.

Aliasing

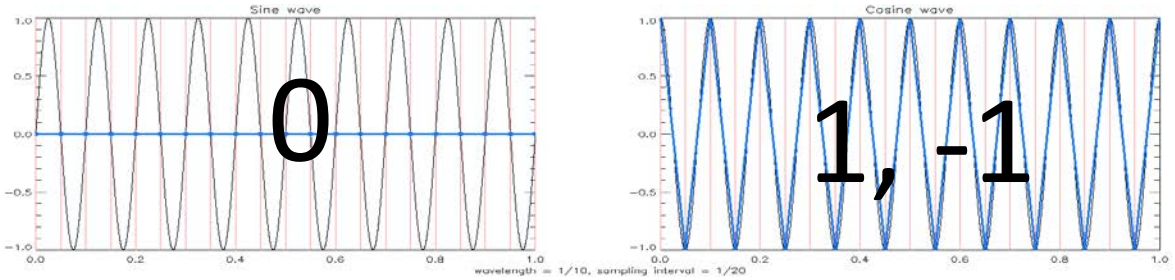
Aliasing is greatly reduced (though not exactly to zero) if your data samples are averages over the sampling period, rather than point values in time or space.

The only way to completely avoid aliasing is to have a “band width limited” signal. i.e. to know beforehand that your data contain no periods (or wavelengths) shorter than the Nyquist period, which is 2δt where δt is the sampling interval.

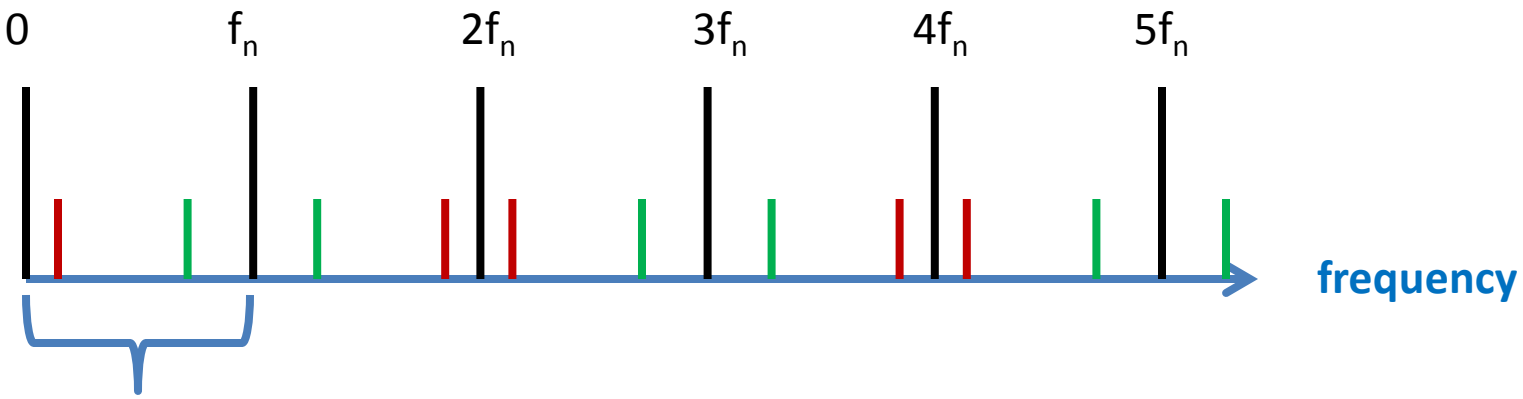
Put another way: frequency f = 1/period, and angular frequency ω = 2π/period, so this means no frequencies higher than the Nyquist frequency:

$$f_n = \frac{1}{2\delta t} \qquad \omega_n = \frac{\pi}{\delta t}$$

This is how the Nyquist frequency aliases:



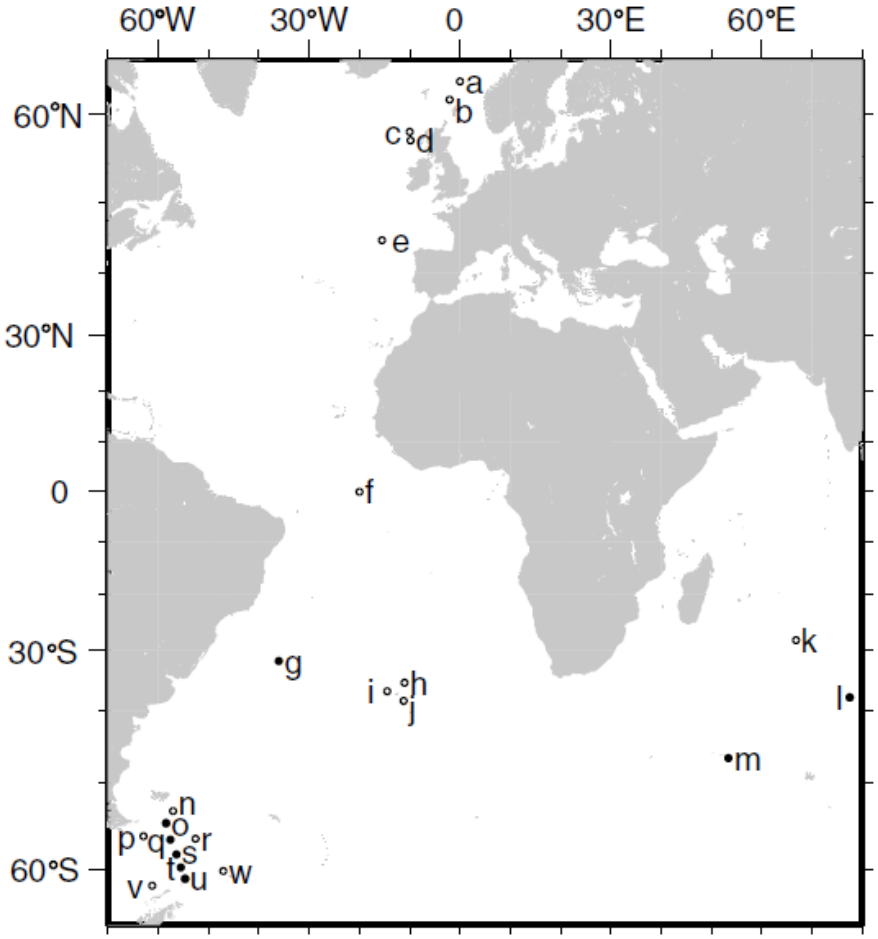
Frequencies higher than the Nyquist frequency appear as if reflected about the Nyquist frequency f_n , and offset by multiples of twice that frequency:



Red and green frequencies appearing in this interval could actually be due to any of the other red and green frequencies being aliased.

Usually aim to sample frequently enough, or with sufficient averaging, that there is relatively little energy at frequencies above f_n, though this isn't always possible.

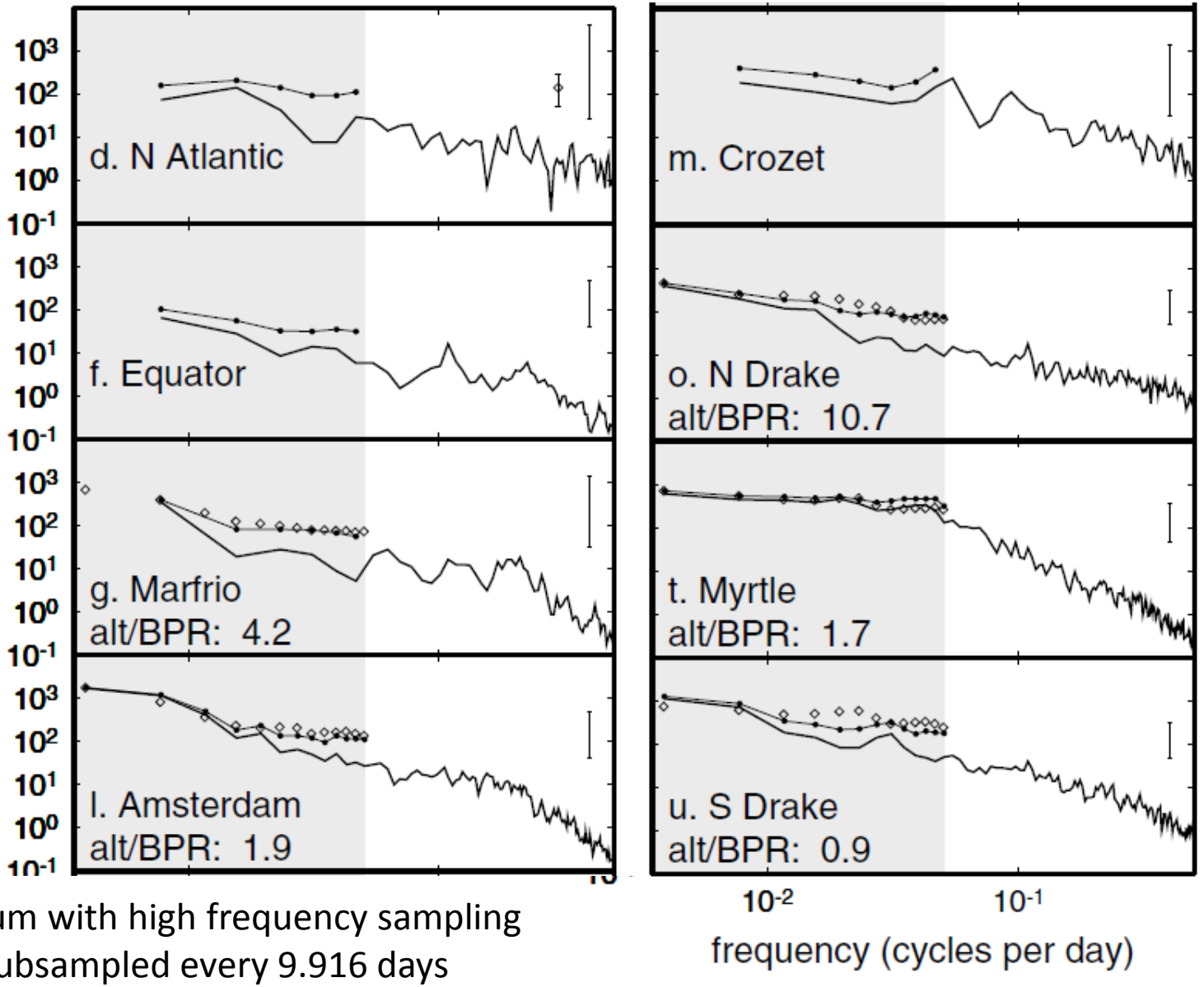
Gille & Hughes, 2001: Geophys. Res. Letters, 28(9), 1755-1758



Bottom pressure measurements from these points, with data every 15 minutes. Allows us to see how much signal is aliased in satellite sea level measurements with data every 9.916 days

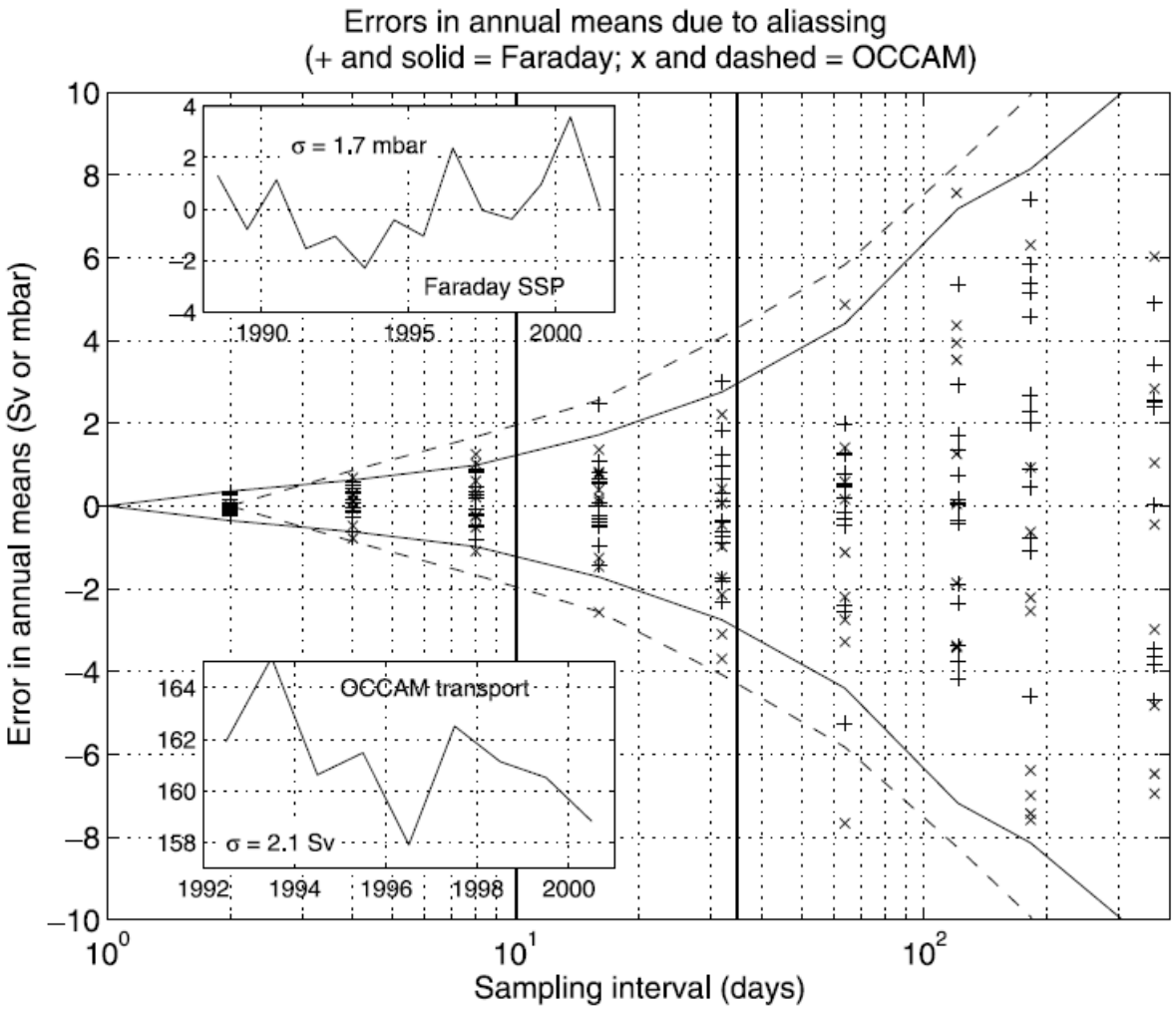
Gille & Hughes, 2001: Geophys. Res. Letters, 28(9), 1755-1758,

Steeply-sloping spectrum means less aliasing.

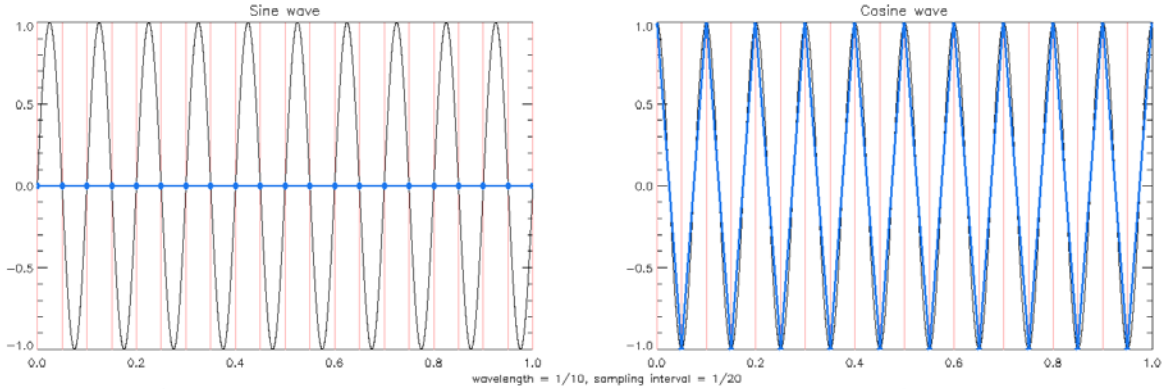


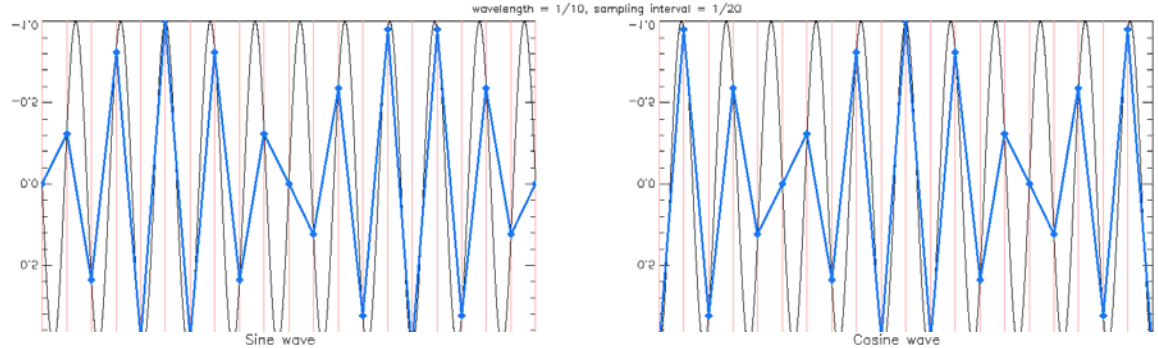
Line = true spectrum with high frequency sampling
Dots = with data subsampled every 9.916 days
Diamonds = satellite data measured every 9.916 days

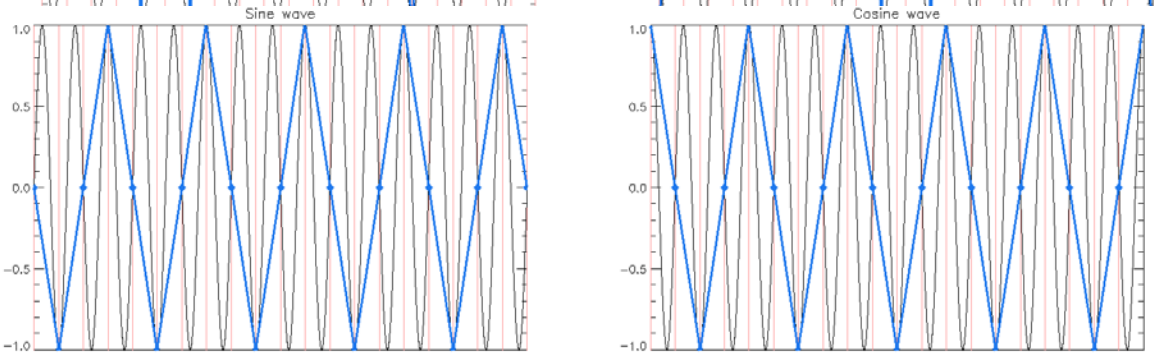
Meredith & Hughes 2005: Geophys. Res. Letters 32, L03609, doi:10.1029/2004GL022086

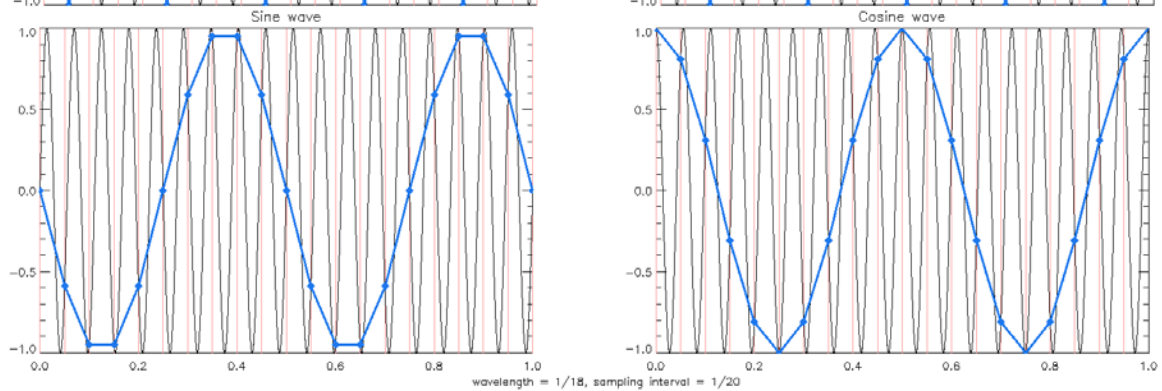


How often do you have to sample in order for the sampling error in annual means to drop below a certain level. For 1 Sv error in ACC transport, or 1 cm error in Antarctic sea level: about every 3 days.


Sine wave
Cosine wave
wavelength = 1/10, sampling interval = 1/20




Sine wave
Cosine wave


Sine wave
Cosine wave
wavelength = 1/18, sampling interval = 1/20

So, if we have some interval T of time (or space), there are an infinite number of sine and cosine waves that will fit a whole number of periods (or wavelengths) into that interval.

These have periods T/n where n is any positive integer. They have the forms

$$y = \sin\left(\frac{2\pi nt}{T}\right) \qquad \text{and} \qquad y = \cos\left(\frac{2\pi nt}{T}\right)$$

(actually, we can include n=0, because sin(0) = 0 and cos(0) = 1, so this allows us to have a constant too).

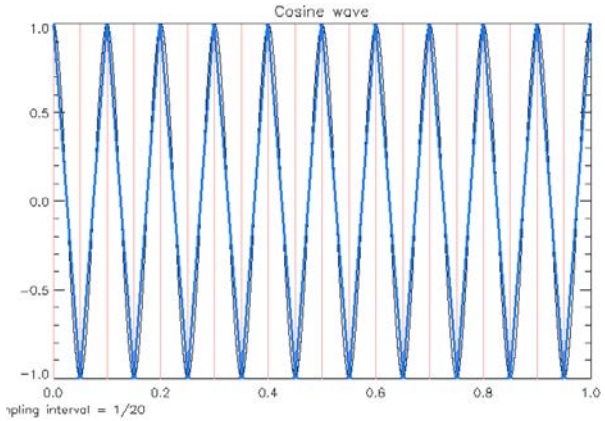
However, if we only have data from 20 regularly-spaced samples in that interval, all of those sines and cosines look like one of only 9 sines (wavenumbers 1 to 9) and 11 cosines (0 to 10, where 0 gives the constant).

This makes sense if you think that you have 20 numbers in your time series, so you should only need 20 basis functions to completely describe it.

Despite their uneven look, these N=20 subsampled sines and cosines are all orthogonal to each other, and all have variance 0.5 (except the constant term and the Nyquist cosine term, which each have variance 1), making them easy to normalize (multiply by $\sqrt{2/N}$, apart from the constant and Nyquist terms which should be multiplied by $\sqrt{1/N}$).

We have 20 numbers in our sample, and 20 orthonormal basis function to use to represent them.

These are a constant, sine and cosine of 9 different frequencies, and cosine of the Nyquist frequency:

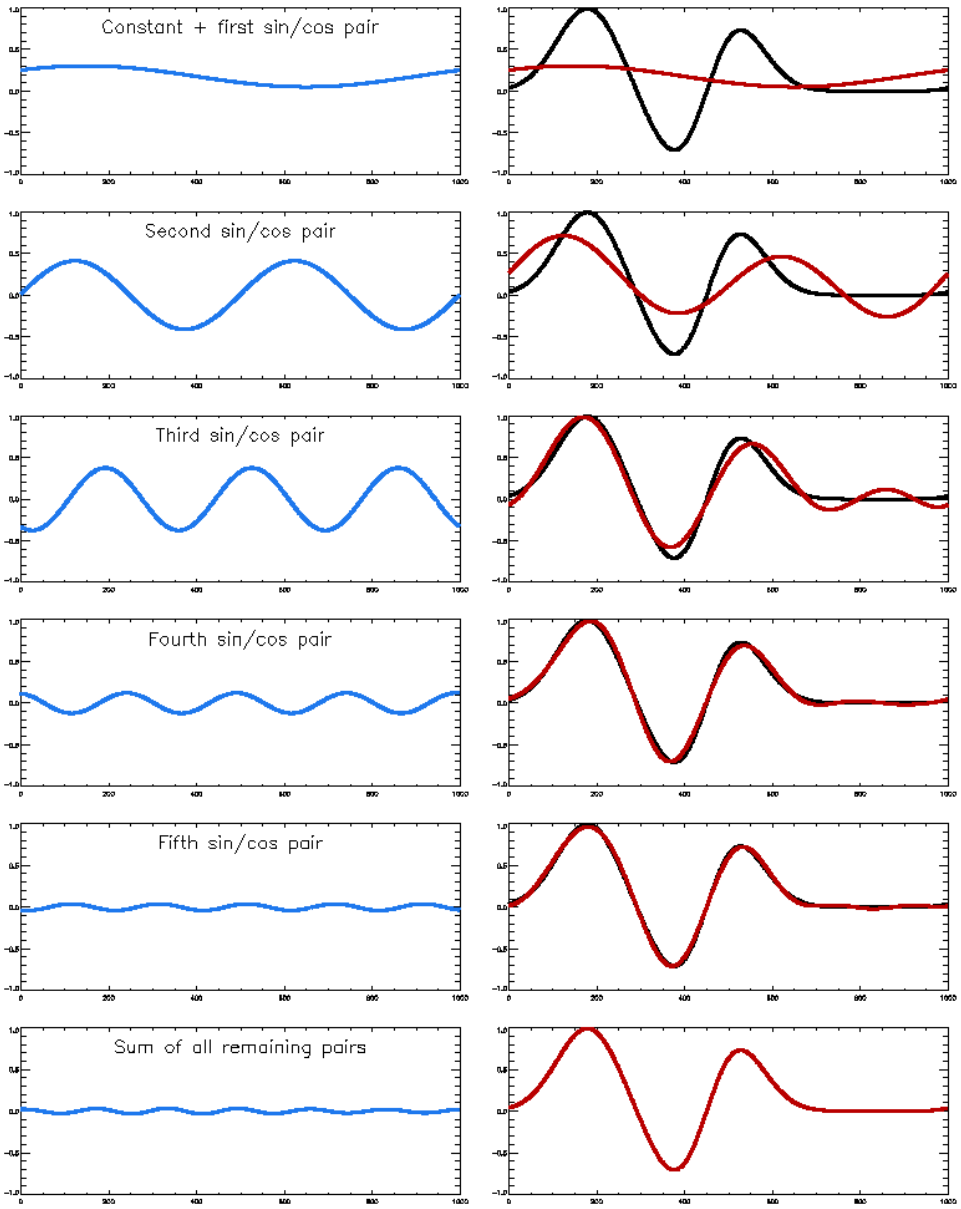


For any even number of samples N=2m, the analogous set of 2m basis functions exists: constant, m-1 sines, m-1 cosines, and cosine at the Nyquist frequency.

For any odd number N=2m-1, no sine or cosine wave fitting exactly in the total range happens to land exactly on the Nyquist frequency, so you have 2m-1 basis functions: a constant, m-1 sines, and m-1 cosines.

By expressing your time series as a sum of these sines and cosines, you can express your data as a function of frequency instead of as a function of time.

This is useful because many processes can be understood in terms of frequency and wavenumber (such as wave dispersion relations). You may be able to pick out different processes which happen in different frequency ranges.

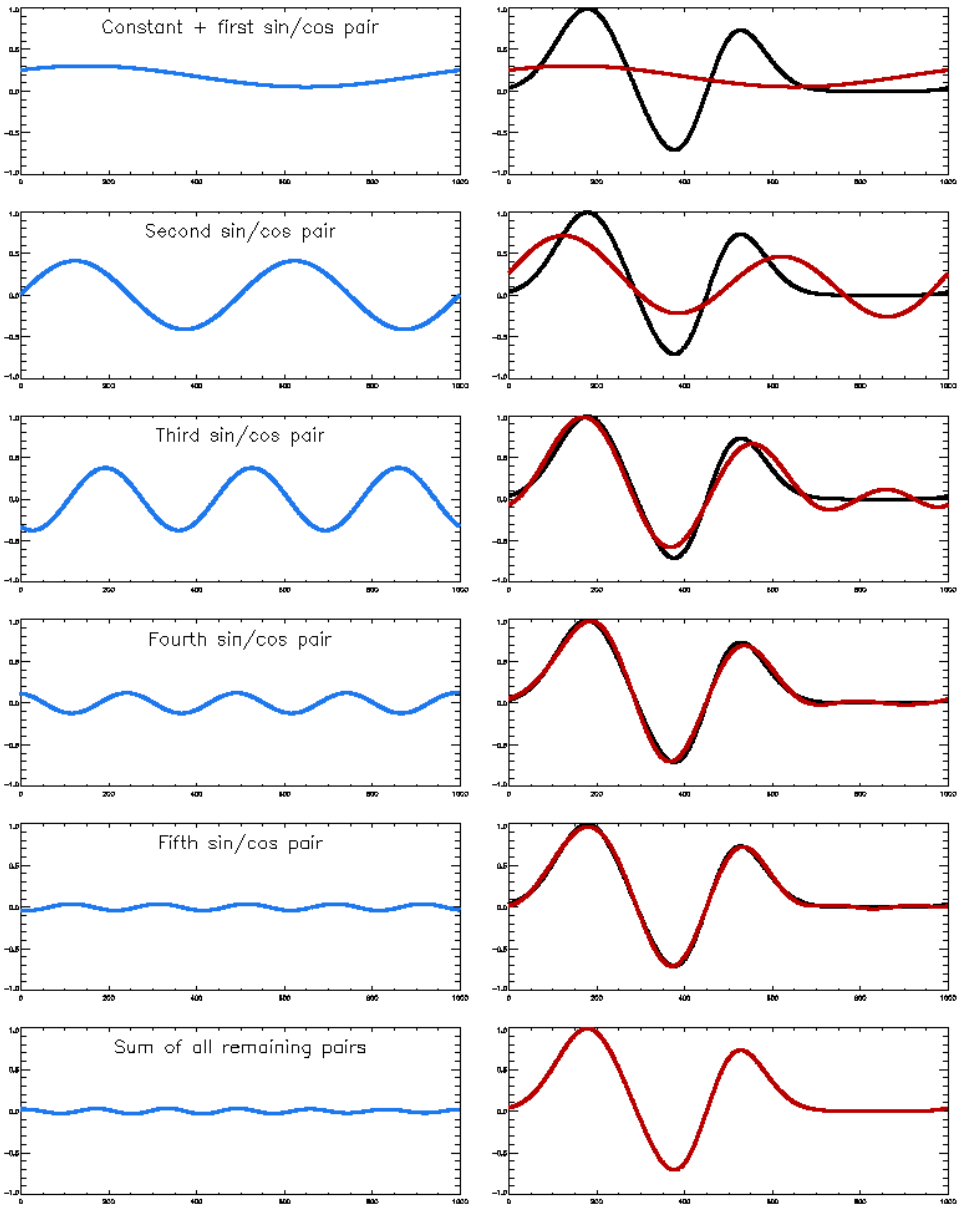


Black curve is a smooth time series y(t), 1000 points long, in this case.

Blue curves are approximations to the black calculated from individual pairs of sines and cosines

Red curves are sums of the blue curves, making successively better approximations to the black curve.

Amplitudes of each of the sines and cosines were calculated by a the dot product of the normalized sine curve with the time series y



i.e., in Matlab (red in text is a column vector, blue a row):

y *is the time series (column vector length 1000).*

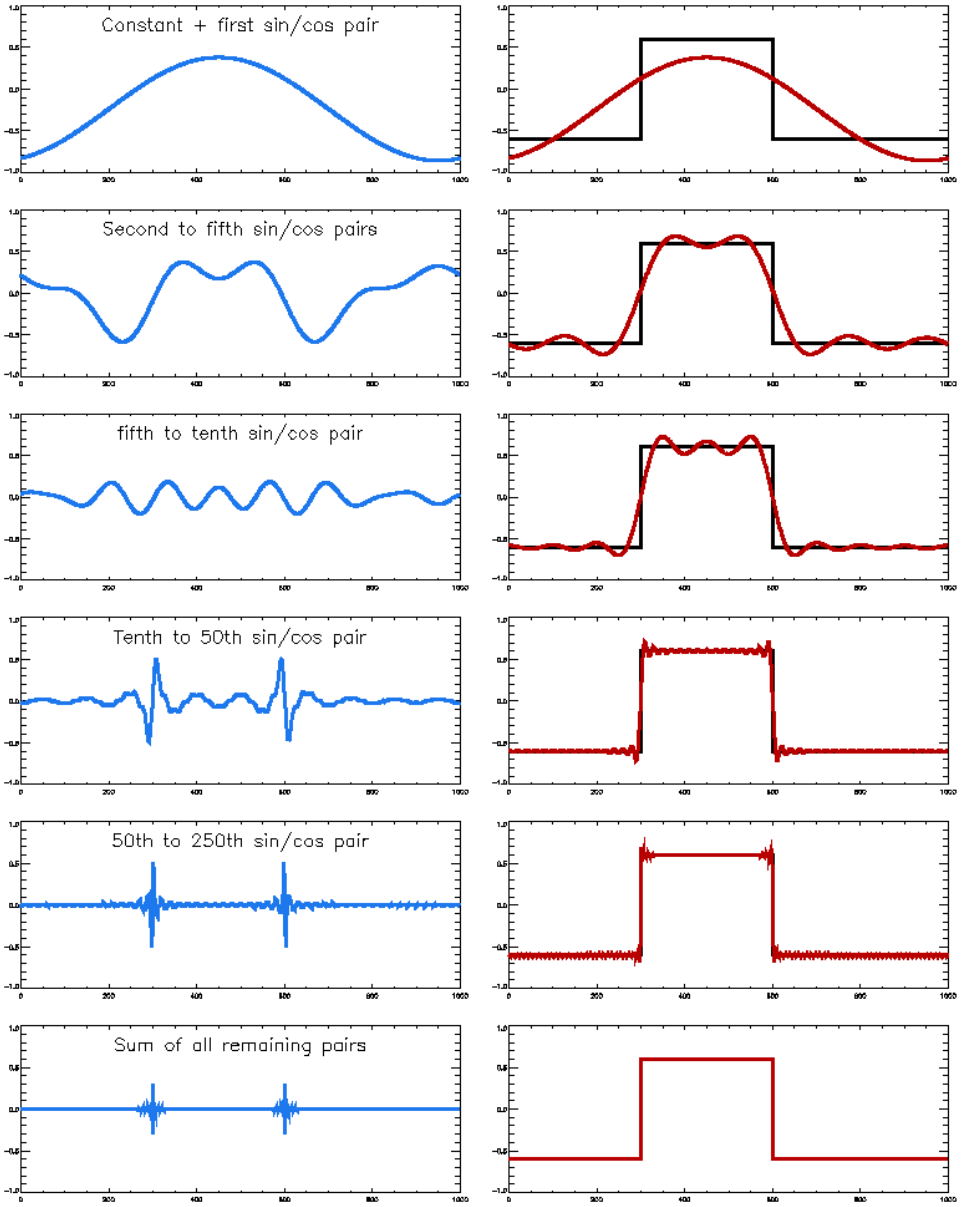
`t=[0:999]'` *represents the time, with a data span of* T=1000

`An=cos(2.0*π*n*t./T)` *is the nth cosine wave, not normalized.*

`an = An/sqrt(2.0*n)` *is now normalized (so* `an'*an` *= 1)*

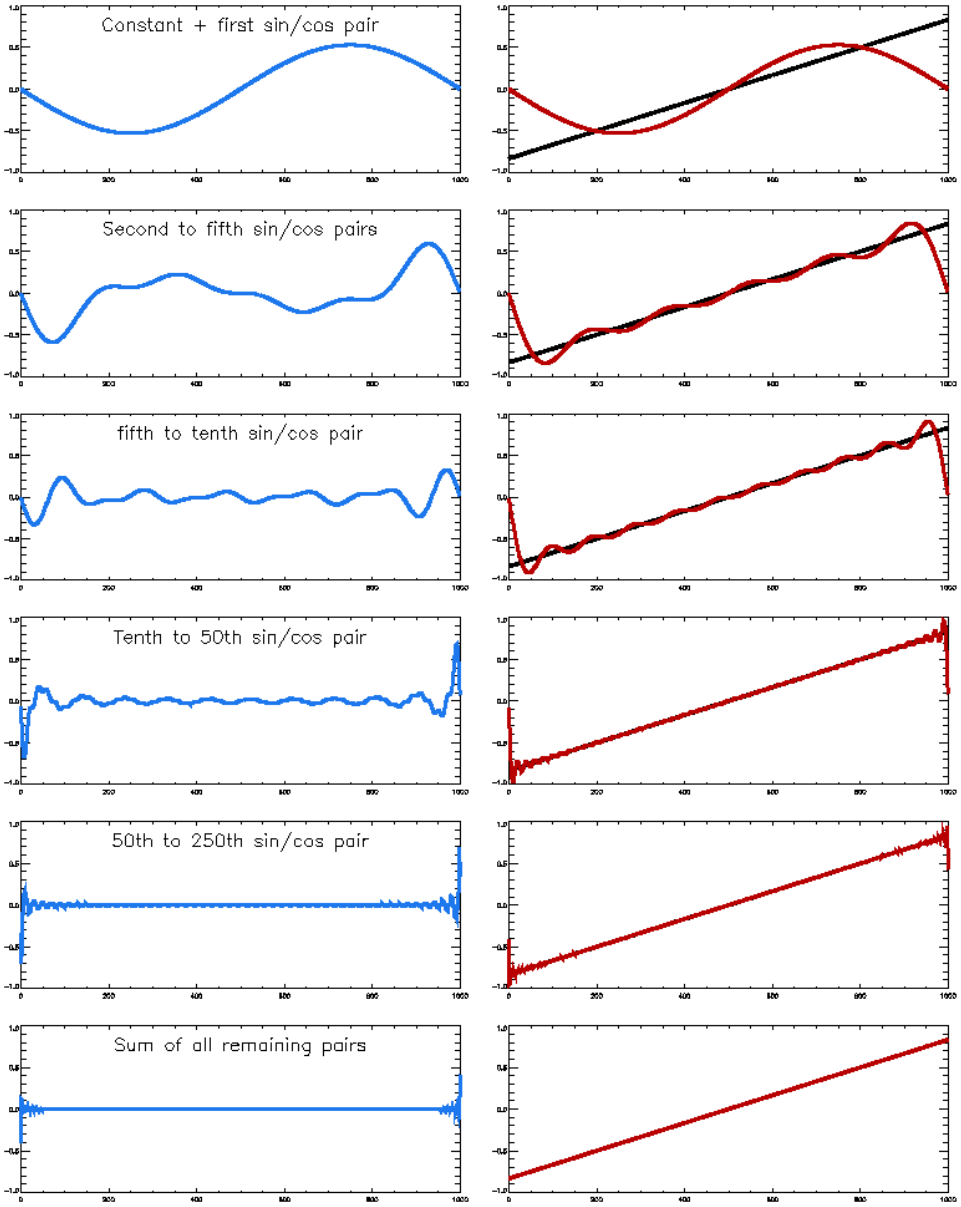
`cn=an'*y` *calculates the nth coefficient by the dot product of cosine wave with* y

`yn=cn*an` *is the bit of y reconstructed from the nth cosine wave. Red curves are these, together with sine waves (plus the 2 special cases), added up.*



Smooth curves can be well approximated by a small number of sine waves. Curves with sharp steps, like this one, need lots of sine waves (and still have problems near corners).

Using all the waves gives a perfect reconstruction (only in the discrete case, where there are a finite number), but even half the waves leave large errors near the steps, and radiating out from them (Gibbs' phenomenon).



You might thing a straight line – a linear trend - would count as “smooth”.

It doesn’t because all the sine and cosine waves are cyclic – they repeat indefinitely. So they try to reconstruct a repeating version of the trend, i.e. a sawtooth wave:

This contains sharp steps, so needs high frequency waves to make it.

In using the discrete Fourier transform, you are implicitly extending your time series with an infinite set of repetitions.

The Fourier transform of a time series consists of the list of coefficients of the waves of different frequency, in order of increasing frequency. There is a continuous FT for continuous data, but here we are considering the Discrete FT which is relevant for finite stretches of regularly sampled data (what you get in real world data)

If the time series is even, i.e. the same when reflected about the origin, only cosines are needed

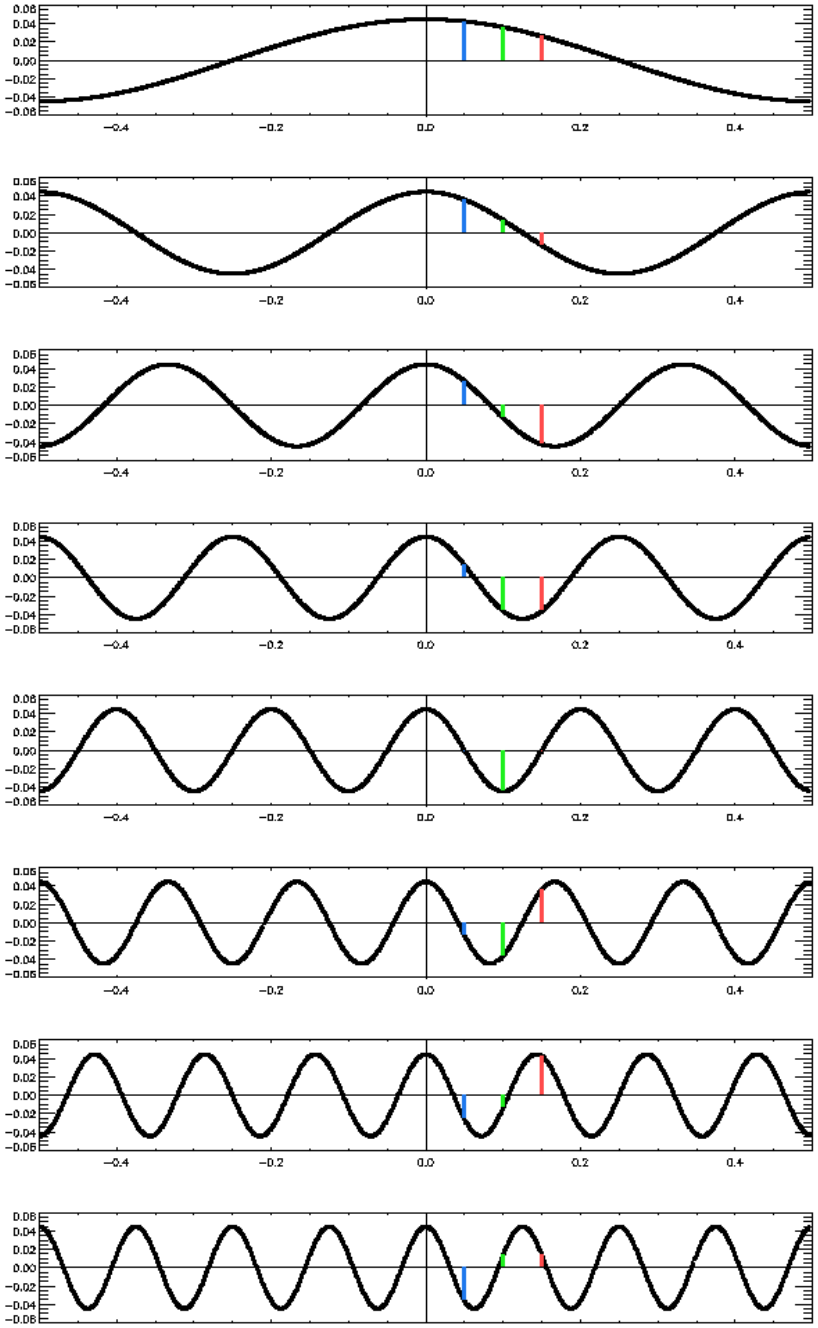
If the time series is odd, i.e. changes sign when reflected about the origin, only sines are needed

The transform is linear: the transform of the sum of 2 time series is the sum of the transforms of the individual time series

The Fourier transform of a pure sine or cosine wave (which fits an exact number of periods in the width of your time window), is a series of zeros, with a 1 at the frequency which matches that wave: a discrete delta function.

So far this is just like any other set of basis functions, although useful because it separates out different frequencies. There are 2 special properties of the FT which make it unusual:

1) If you apply it twice, you get back to where you started
2) There is an algorithm, the Fast Fourier Transform (FFT) which can do this decomposition much faster than other decompositions onto basis functions.



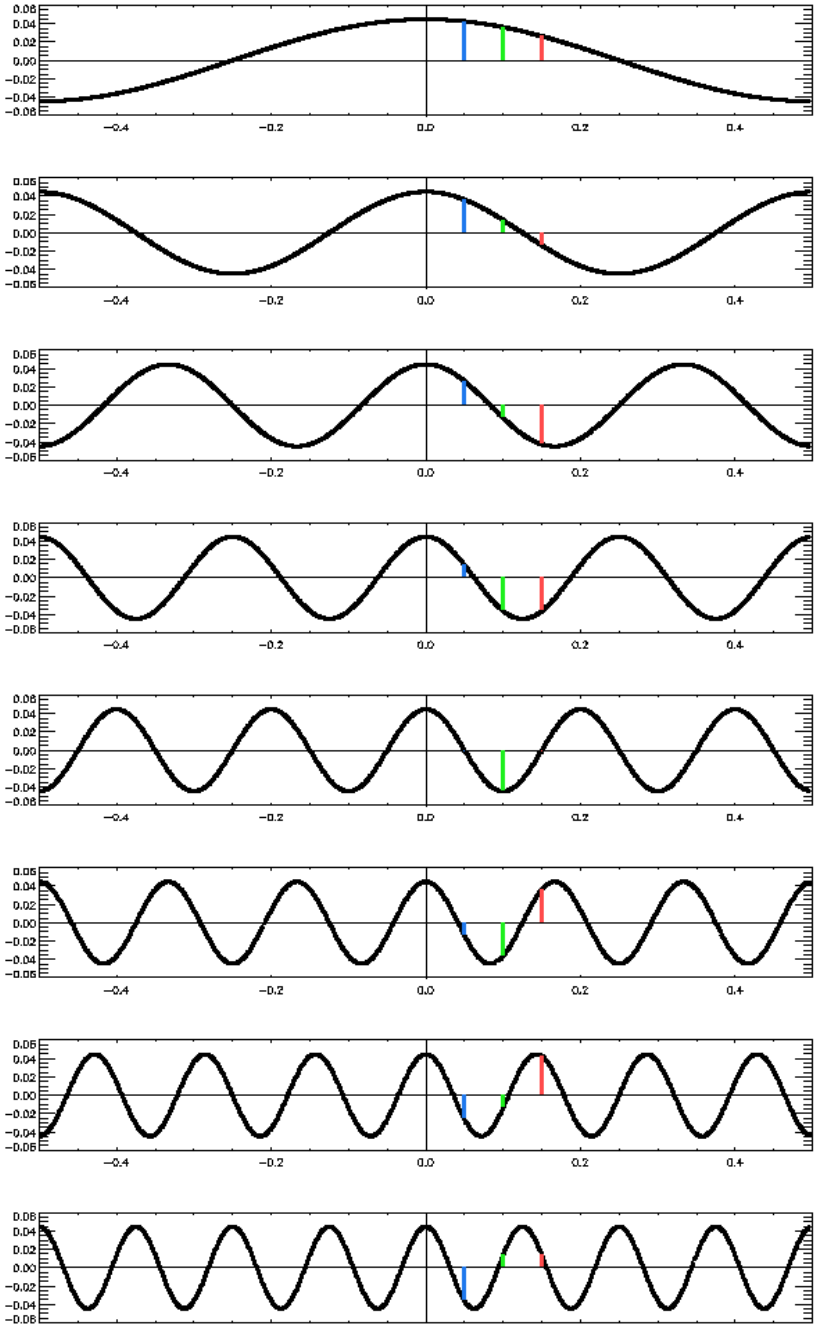
The time series you are calculating the FT of is zero everywhere, but 1 where the blue line is (a discrete delta function)

The dot product with the different cosine waves (i.e. the values of the FT coefficients) is given by the length of the blue line.

As the frequency gets higher, the waves squeeze closer to the origin, so the length of the blue line changes as the waves bunch up

Instead of thinking of the sine/cosine wave squeezing up by a factor 1, 2, 3… etc, you could think of the blue line moving to the right by 1, 2, 3… steps

Clearly, the length of the blue line would then oscillate as a sine wave as a function of frequency



For a delta function at a different position (green, red), the same happens but the oscillations are faster (the steps are bigger).

The FT of a delta function in time is a sine/cosine wave in frequency

The FT of a sine/cosine wave in time is a delta function in frequency.

If you have a time series, and calculate the discrete FT, you get a list of sine/cosine coefficients as a function of frequency

If you take that pattern as a function of frequency, and apply a discrete FT to that pattern (using sine/cosine waves as a function of frequency rather than time), you will get back to a function of time. You get back the same time series you started with (perhaps with a different amplitude depending on normalization).

This can make the FT a powerful tool for manipulating data, as well as for determining the frequency content of your time series.

FFT and complex numbers

All packages, Matlab included, use complex numbers to perform FFTs. This makes the logic of the algorithm more straightforward (sine and cosine terms are organized in a way which makes the transforms to frequency domain and back again exactly equivalent, apart from the question of normalization), deals naturally with the different normalization for the constant and Nyquist functions (all the others are naturally joined in pairs, so their two half sized normalizations add together), and makes it possible to deal with time series of complex inputs.

This doesn't need to bother you if you don't understand complex numbers, all you need to know is how to translate them.

A complex number can be thought of as a combination of an amplitude and an angle:

$$C = (a + ib) = Ae^{i\theta} = A\cos\theta + Ai\sin\theta)$$

where $A^2 = a^2$+b^2. One complex number represents one pair of numbers which tell you the amplitudes of the sine and cosine waves at one frequency.

real(C) = a, imag(C)=b

Translating a Matlab FFT

You have a time series y(t) of say N=100 points (column array, dimensions 100,0), spanning a length of time T (i.e. your points are at t=0, t=T/100, t=2T/100… t=99T/100), and you want to represent this as a sum of cosine and sine waves multiplied by coefficients a_n, b_n:

$$y = a_0\sqrt{1/N}\cos\frac{2\pi \times 0 \times t}{T} + b_0\sqrt{1/N}\sin\frac{2\pi \times 0 \times t}{T}$$

($\cos$ term: = 1; $\sin$ term: = 0)

$$+ a_1\sqrt{2/N}\cos\frac{2\pi \times 1 \times t}{T} + b_1\sqrt{2/N}\sin\frac{2\pi \times 1 \times t}{T}$$

$$+ a_2\sqrt{2/N}\cos\frac{2\pi \times 2 \times t}{T} + b_2\sqrt{2/N}\sin\frac{2\pi \times 2 \times t}{T}$$

…

$$+ a_{49}\sqrt{2/N}\cos\frac{2\pi \times 49 \times t}{T} + b_{49}\sqrt{2/N}\sin\frac{2\pi \times 49 \times t}{T}$$

$$+ a_{50}\sqrt{2/N}\cos\frac{2\pi \times 50 \times t}{T} + b_{50}\sqrt{2/N}\sin\frac{2\pi \times 50 \times t}{T}$$

($\sin$ term: = 0)

The square root factors ensure that each curve is normalized: Sum of squares of elements = 1

This is the Nyquist frequency, only present when the number of points is even (as here for n=100)

Translating a Matlab FFT

You type `yft=fft(y);`
and Matlab produces yft which is complex double with dimensions [n,1] where n is 100 here .

As it is a complex array, it contains N=100 pairs numbers (twice as many as you need if your time series was real).

The translation:

constant	real(yft(1,0)) = $a_0\sqrt{N}$	imag(yft(1,0)) = 0
	real(yft(2,0)) = $a_1\sqrt{N/2}$	imag(yft(2,0)) = -$b_1\sqrt{N/2}$
	real(yft(3,0)) = $a_2\sqrt{N/2}$	imag(yft(3,0)) = -$b_2\sqrt{N/2}$
	…	
	real(yft(50,0)) = $a_{49}\sqrt{N/2}$	imag(yft(50,0)) = -$b_{49}\sqrt{N/2}$
Nyquist (absent if n is odd)	real(yft(51,0)) = $a_{50}\sqrt{N}$	imag(yft(51,0)) = 0

Translating a Matlab FFT

real(yft(1,0)) = $a_0\sqrt{N}$ imag(yft(1,0)) = 0

real(yft(2,0)) = $a_1\sqrt{N/2}$ imag(yft(2,0)) = -$b_1\sqrt{N/2}$

real(yft(3,0)) = $a_2\sqrt{N/2}$ imag(yft(3,0)) = -$b_2\sqrt{N/2}$

…

real(yft(50,0)) = $a_{49}\sqrt{N/2}$ imag(yft(50,0)) = -$b_{49}\sqrt{N/2}$

real(yft(51,0)) = $a_{50}\sqrt{N}$ imag(yft(51,0)) = 0

so all the information is in the first 51=1+fix((N-1)/2) points.

What about the other 49 points? They contain the complex conjugates of what you find in points 2 to 50 (i.e. the same real part, and an imaginary part which is the same but with the opposite sign), listed in the opposite order.

real $(\sqrt{2}a_0 \; +a_1 \; +a_2 \; +a_3 \; +a_4 \; +a_5 \; \dots \; +a_{49} \; \sqrt{2}a_{50} \; +a_{49} \; \dots \; +a_5 \; +a_4 \; +a_3 \; +a_2 \; +a_1)\sqrt{N/2}$

imaginary $(0 \; -b_1 \; -b_2 \; -b_3 \; -b_4 \; -b_5 \; \dots \; -b_{49} \; 0 \; +b_{49} \; \dots \; +b_5 \; +b_4 \; +b_3 \; +b_2 \; +b_1)\sqrt{N/2}$

Whatever value of N, the number of complex values you need before they start repeating is $\text{fix}\left(\frac{N+1}{2}\right)$, where fix rounds the number towards zero. This gives 51 (a_0 and b_0 to a_{50} and b_{50}) for both N=100 and N=101 (a_{50} and b_{50} are repeated for N=101, not for N=100).

Translating a Matlab FFT: Normalization

$$(\sqrt{2}a_0 \; +a_1 \; +a_2 \; +a_3 \; +a_4 \; +a_5 \; \dots \; +a_{49} \; \sqrt{2}a_{50} \; +a_{49} \; \dots \; +a_5 \; +a_4 \; +a_3 \; +a_2 \; +a_1)\sqrt{N/2}$$
$$(0 \quad -b_1 \; -b_2 \; -b_3 \; -b_4 \; -b_5 \; \dots \; -b_{49} \quad 0 \quad +b_{49} \; \dots \; +b_5 \; +b_4 \; +b_3 \; +b_2 \; +b_1)\sqrt{N/2}$$

There are various ways to normalize an FFT. Matlab multiplies by $\sqrt{N}$. This has the effect that fft(fft(x)) = Nx. To cope with this, it supplies an inverse fft which divides by $\sqrt{N}$:
ifft(ifft(x)) = x/N. With this, you can go "forwards" with fft, and "backwards" with ifft:
ifft(fft(x)) = x

Note, the x you end up with won't be exactly what you started with for 2 reasons. 1) the accuracy isn't perfect. 2) both fft and ifft produce complex numbers. The imaginary part should be very small, so you'll need real(x) to return to an exactly real time series.

Alternatively you can use ifft(fft(x),'symmetric'), which gives you a real function directly.

You could avoid this difference by always using fft, and always dividing by $\sqrt{N}$

i.e. N = size(x,1) the number of elements per column of x
xft = fft(x)/sqrt(N);
xx = real(fft(x))/sqrt(N) xx should be equal to x (to machine precision).

NB if x is a 2D array, fft will perform separate ffts along each column.

Translating a Matlab FFT: Normalization

Advantages of dividing by sqrt(n):

Forward and backward transforms are the same.

The sum of the squares of the elements stays the same:

```
>> sum(x'*x)

ans =

  150.0000

>> xft=fft(x)/sqrt(N);
>> sum(xft'*xft)

ans =

  150.0000
```

But note, this is using all of the coefficients, including the repeated ones, and xft' is a complex conjugate transpose, not just a transpose – irrelevant for real arrays, but makes a difference for complex arrays. The sum you are actually doing here is

$$N(a_0^2 + a_1^2 + b_1^2 + a_2^2 + b_2^2 + \cdots + a_{ny}^2)$$

where a_{ny} is the coefficient of the Nyquist frequency wave, if it exists.
Because these are coefficients multiplying the normalized waves (i.e. sine and cosine waves with amplitude by $\sqrt{2/N}$ and constant and Nyquist wave with amplitude $\sqrt{1/N}$), the sums of their squares adds up to the variance of the time series. Variance times N is the sum of the squares of the elements in x.

Translating a Matlab FFT: Normalization

How come the sum you are actually doing here is this?

$$N(a_0^2 + a_1^2 + b_1^2 + a_2^2 + b_2^2 + \cdots + a_{ny}^2)$$

You are summing the squares of all the elements in this:

$$(\sqrt{2}a_0 \; +a_1 \; +a_2 \; +a_3 \; +a_4 \; +a_5 \; \ldots \; +a_{49} \; \sqrt{2}a_{50} \; +a_{49} \; \ldots \; +a_5 \; +a_4 \; +a_3 \; +a_2 \; +a_1)\sqrt{N/2}$$
$$(0 \quad -b_1 \; -b_2 \; -b_3 \; -b_4 \; -b_5 \; \ldots \; -b_{49} \quad 0 \quad +b_{49} \; \ldots \; +b_5 \; +b_4 \; +b_3 \; +b_2 \; +b_1)\sqrt{N/2}$$

which is the sum of

$$(2a_0^2 \; a_1^2 \; a_2^2 \; a_3^2 \; a^2_4 \; a_5^2 \; \ldots \; a_{49}^2 \quad 2a_{50}^2 \; a_{49}^2 \; \ldots \; a_5^2 \; a_4^2 \; a_3^2 \; a_2^2 \; a_1^2)(N/2)$$
$$(0 \quad b_1^2 \; b_2^2 \; b_3^2 \; b_4^2 \; b_5^2 \; \ldots \; b_{49}^2 \quad 0 \quad b_{49}^2 \; \ldots \; b_5^2 \; b_4^2 \; b_3^2 \; b_2^2 \; b_1^2)(N/2)$$

and, gathering together the pairs, this becomes the sum of

$$(a_0^2 \; a_1^2 \; a_2^2 \; a_3^2 \; a^2_4 \; a_5^2 \; \ldots \; a_{49}^2 \; a_{50}^2)N$$
$$(0 \quad b_1^2 \; b_2^2 \; b_3^2 \; b_4^2 \; b_5^2 \; \ldots \; b_{49}^2 \quad 0)N$$

… as above.

Matlab FFT summary

- Time series with N points.
- If N is odd, then this is the sum of a constant, M = (N-1)/2 cosines, M = (N-1)/2 sines
- If N is even, it is the sum of a constant, M = N/2 - 1 cosines, M = N/2 - 1 sines and an extra cosine at the Nyquist frequency (alternating ±1 every grid point).
- The curves are sines and cosines of $2\pi nt/T$ where n ranges from 1 to M, and includes 0 for the cosine term (i.e. a constant), and also M+1 for the cosine term if N is even (the Nyquist term). T is the total length of the time series, i.e. $T=N\delta t$
- To be normalized so the sum of squares of elements in each basis vector is 1, the sines and cosines have amplitude $\sqrt{2/N}$, and the constant and Nyquist curve have amplitude $\sqrt{1/N}$
- xft=fft(x) returns a complex vector of length N. The real part of this which contains $\sqrt{2}\ \times$ the coefficient of the constant term in the first element, the coefficients of the cosine terms in the next M elements $\sqrt{2}\ \times$ the coefficient of the Nyquist term in the next (if N is odd), and then repeats the coefficients of the cosine terms in reverse order, all multiplied by $\sqrt{N/2}$
- The imaginary part contains 0 in the first element, minus the coefficients of the sine terms in the next M elements 0 in the next if N is odd, and then repeats the coefficients of the sine terms in reverse order (and without the minus sign), all multiplied by $\sqrt{N/2}$

Matlab FFT summary

- The variance explained by each coefficient is the square of that coefficient (with our normalized coefficients).
- For the corresponding terms in the Matlab FFT, it is (1/N) times the square of the term (1/2N in the case of the constant and Nyquist terms).
- In Matlab xft=fft(x); xr=fft(xft) produces xr which is larger than x by a factor of N. Dividing by $\sqrt{N}$ after each fft would avoid this, and would renormalize xft so that xft'*xft = x'*x, i.e. the sum of the squares of the coefficients is the same as the sum of the squares of the elements in x.
- In short: you can translate between time series and a list of coefficients of frequency, and back again by using fft. Divide by $\sqrt{N}$ if you want the sum of squares of elements to stay constant, (or use fft and ifft to go "forwards" and "backwards", and remember that the Nyquist and constant coefficients behave differently from the rest (you can avoid worrying about these if you subtract means first, and only use odd-length time series, but see next point). When you need to remember the details, look them up!
- A final technical point: FFTs can, in principle, be designed to work fast with any length of data. In practice, many algorithms prefer data with length a power of 2 or a product of small prime numbers (e.g. 2, 3, 5).

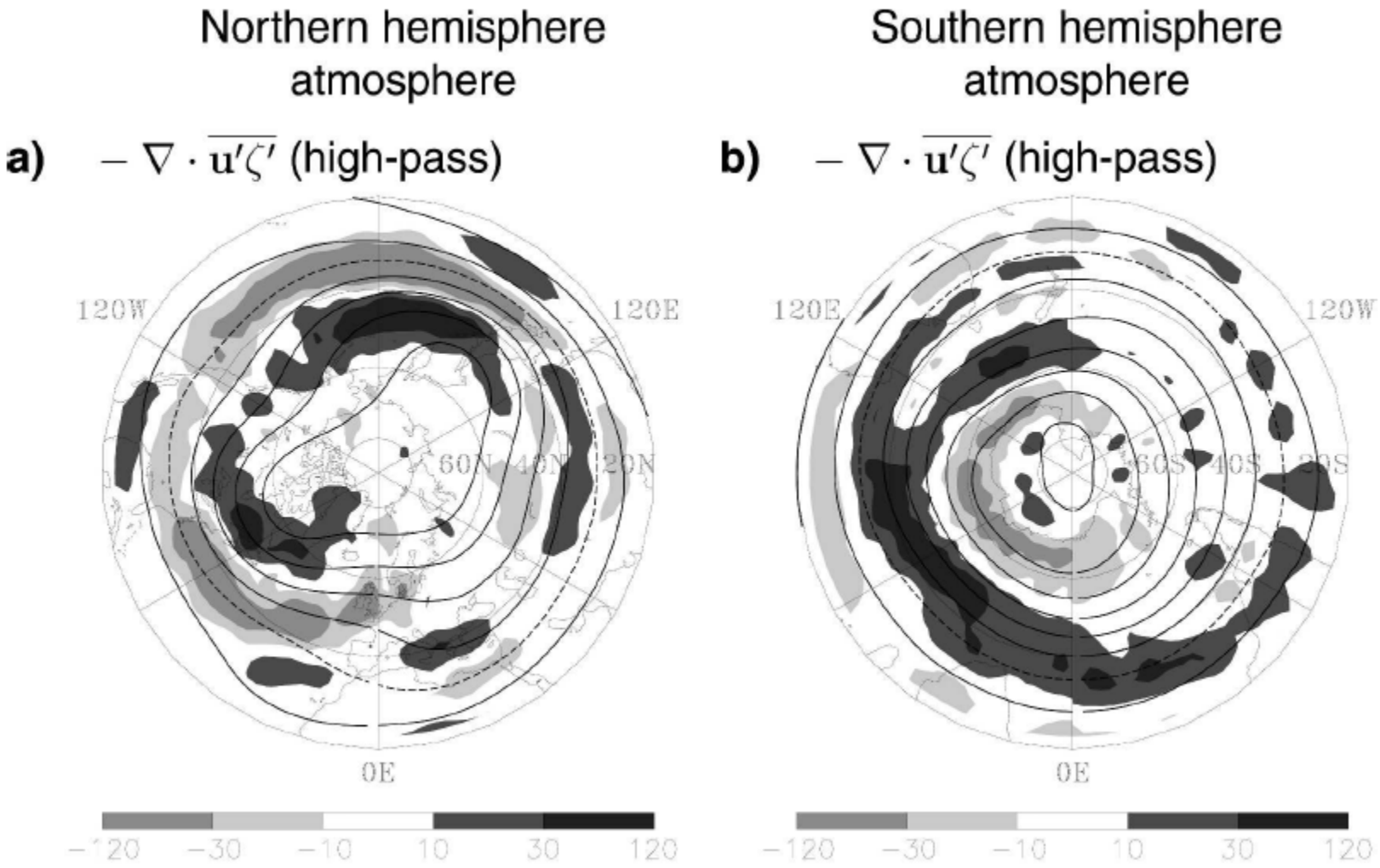
Matlab FFT: more dimensions

- xft=fft(x) – if x is a column, will just calculate the FFT. If x is multidimensional, it will calculate FFTs for each column independently. ifft(fft(x),'symmetric') gets back to x.
- xft=fft2(x) takes a 2 dimensional array and performs a 2D FFT. i.e. it transforms time into frequency, and space into wavenumber. ifft2(fft2(x),'symmetric') gets back to x. This is the same as doing 1D FFTs in time first, and then in space, or vice versa.
- xft=fftn(x) takes an n-dimensional array x and performs an n-dimensional FFT.
- ifftn(fftn(x),'symmetric') gets back to x.

What can we do with this tool?

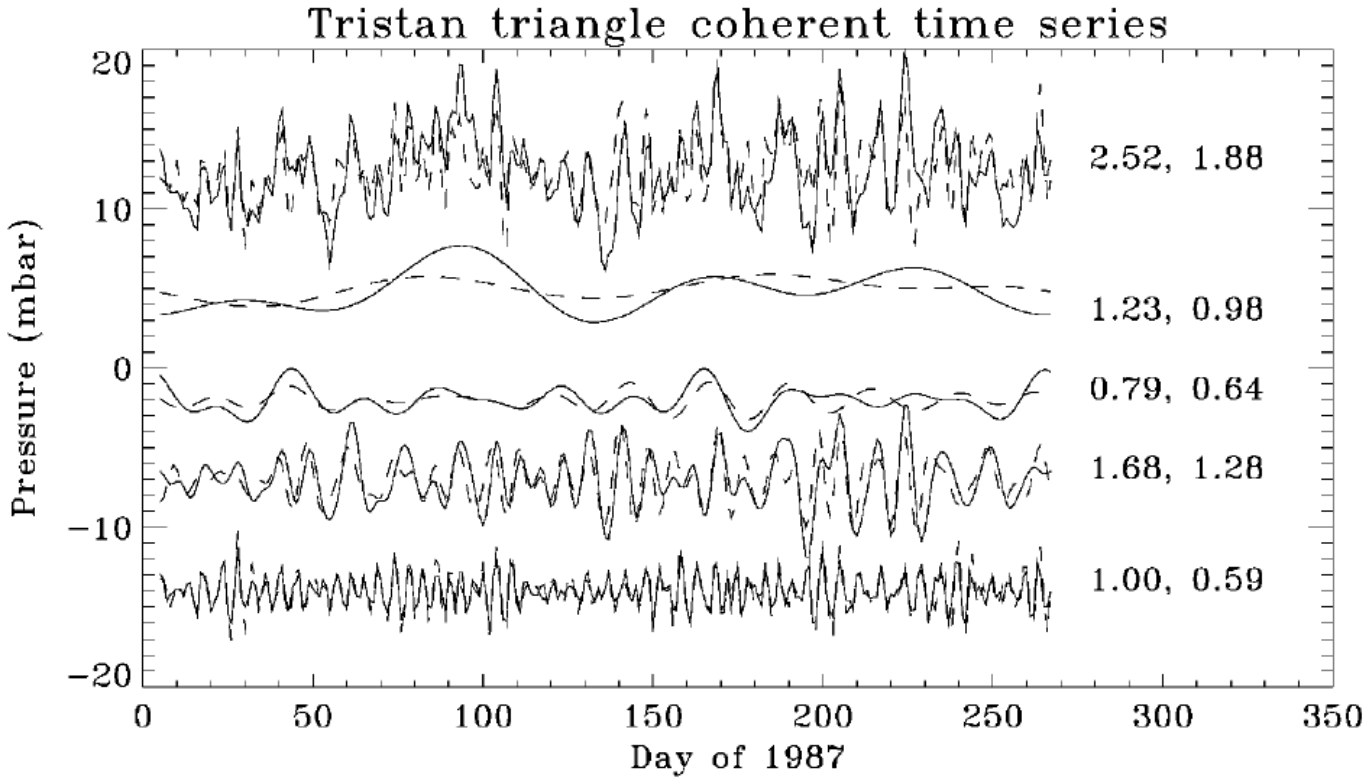
- Filter data to focus on certain frequencies/wavenumbers
- Calculate spectra to see what frequencies contain most energy
- Do cross-spectral analysis (see how the relationship between two time series depends on frequency).
- Shift data in time by a fraction of a time step (only if there is no energy at frequencies above the Nyquist limit).
- Lots of other clever things…

Williams et al., 2007: J. Phys. Oceanogr., 2267-2289.



Determining how eddies interact with the mean flow (in the atmosphere here). We define atmospheric “eddies” as variability at periods shorter than 7 days.

How well does a model reproduce observations?

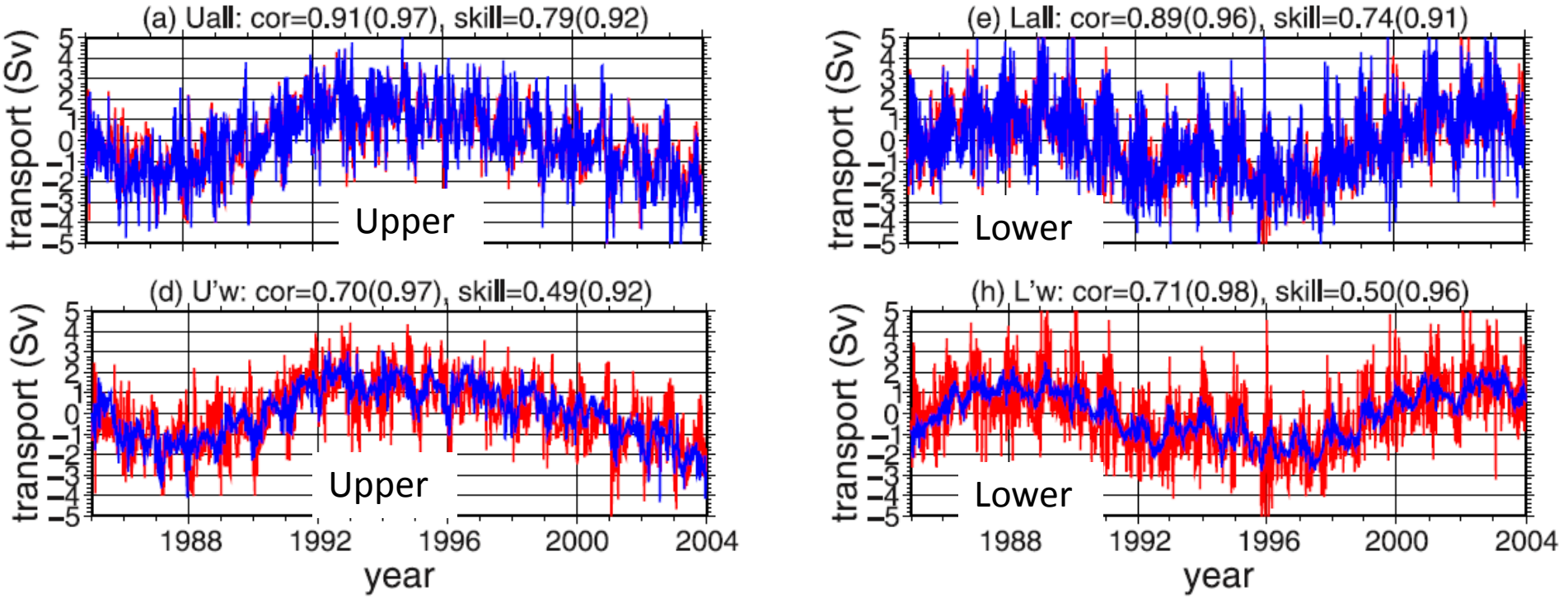


Solid line is measurements, dashed line is from an ocean model. Numbers show standard deviation of measurement, and of measurement after subtracting the model prediction.

At 2-6 days, the model explains $100(1-0.59^2/1.00^2) = 65$ % of the measured variance
This decreases to 42% at 6-20 days, 34% at 20-60 days, and 36% at longer than 60 days, compared with 44% overall. We can see how the model does a better job at some frequencies than others.

Bingham & Hughes, 2008: J. Geophys. Res. 113, C09008, doi:10.1029/2007JC004679

Model study – how well can we measure changes in the upper and lower branch of the MOC if we only have boundary pressure measurements

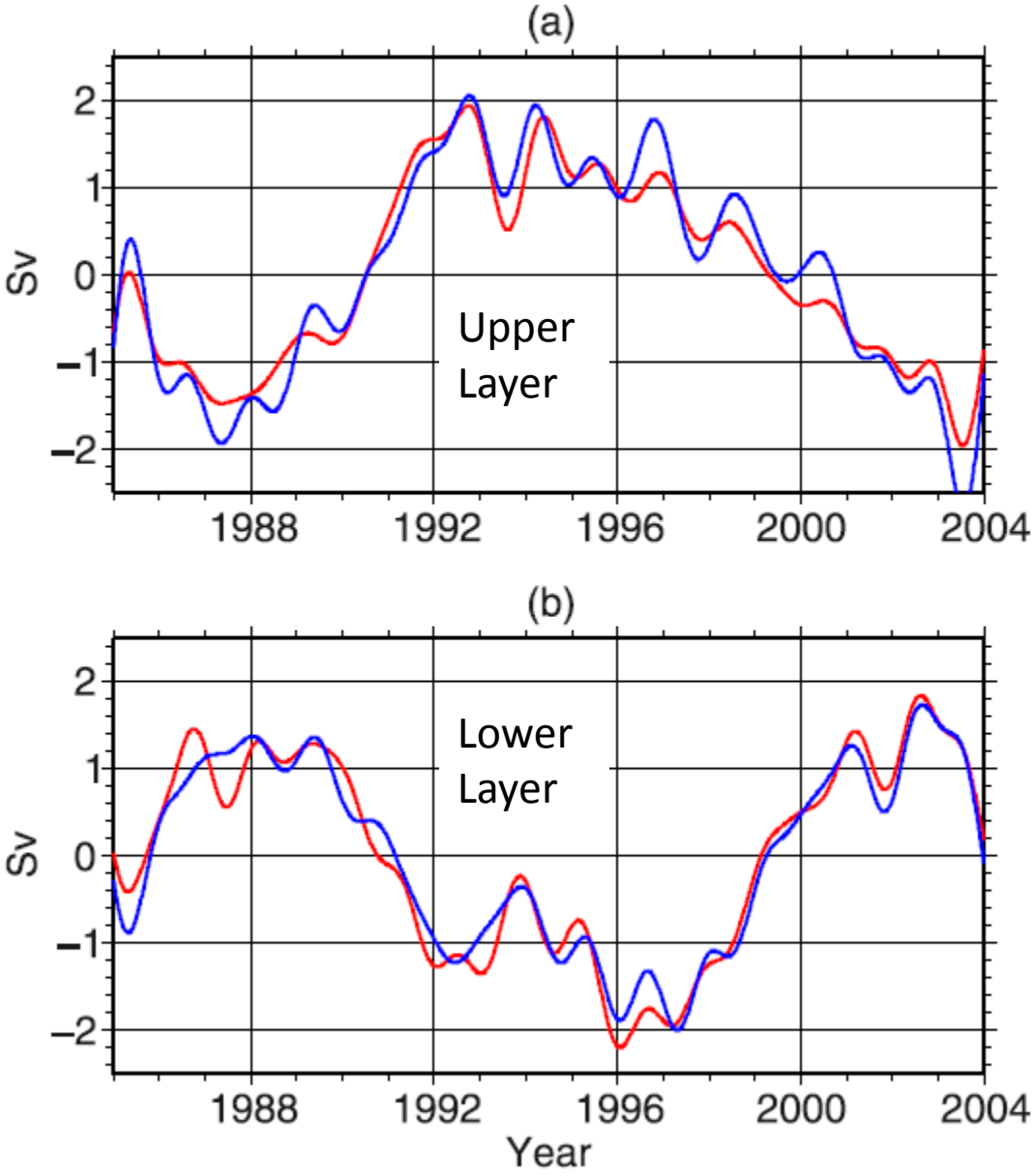


Red is model truth, blue is reproduction from pressure "measurements", if we have them (top) everywhere, or (bottom) just on the western continental slope.

cor = correlation, skill = fraction of variance explained.
Numbers in brackets are after low pass filtering (periods > 400 days)

Having only the western boundary works as well as (or better than) having both, for long periods, but not for short periods

Bingham & Hughes, 2008: J. Geophys. Res. 113, C09008, doi:10.1029/2007JC004679



Low pass filtered.

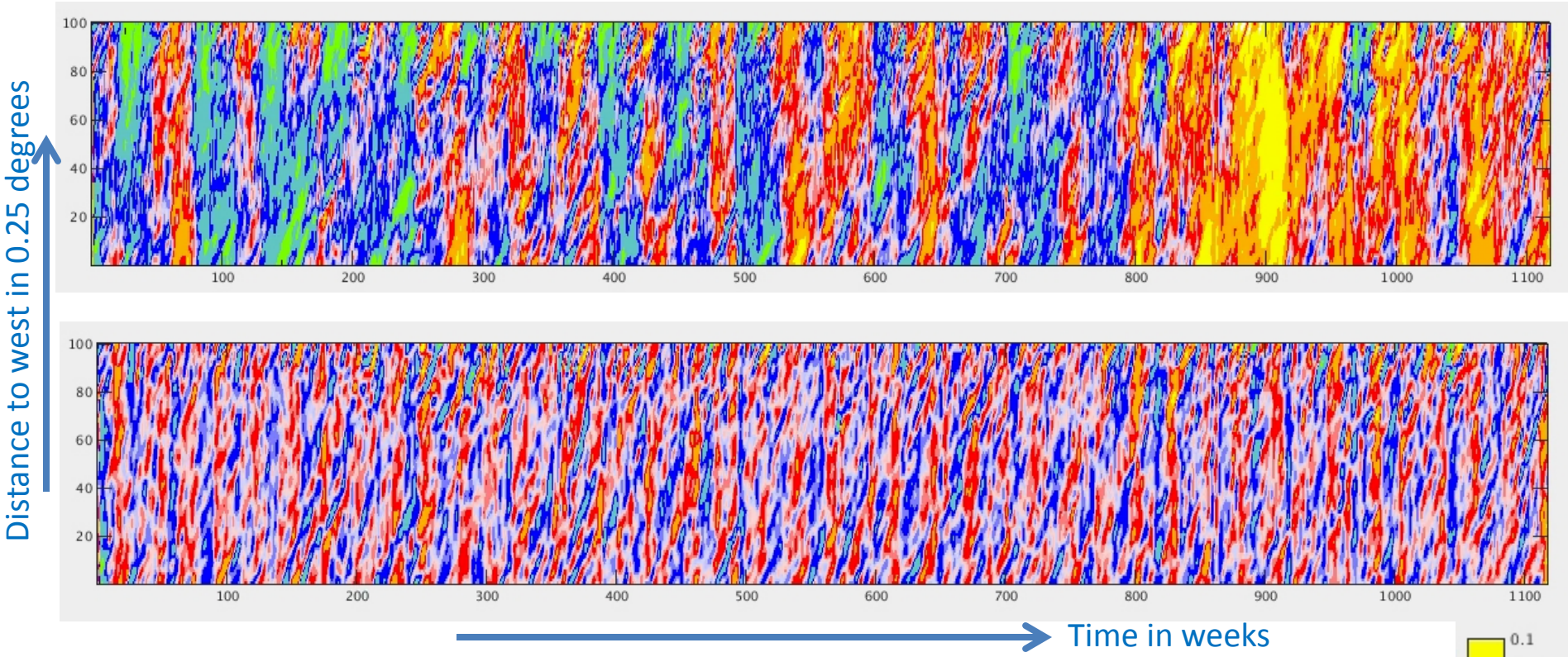
"Truth"

from western boundary "measurements"

```
function [hb,hfilt]=altim_filt(pmin,pmax)
%
% filters to pass only periods pmin to pmax from altimetry (in days)
%
[h,hr]=readavisolat(300);                % 1117 points in time, every 7 days
hb=hr(:,1301:1400);                      % select 100 longitudes
hft=fft(hb);                             % FFT of each time series
hfreq=[0,1:558,558:-1:1]'./(1117*7);     % frequency in cycles per day
hper=1./hfreq;                           % period in days
                                         % Note, first one is infinity
ffilt=1+hper*0;                          % make the filter, fill with ones
ffilt(hper < pmin)=0;                    % set to zero periods < pmin
ffilt(hper > pmax)=0;                    % set to zero periods > pmax
hftf=hft;                                % make a copy of the FFT
for i=1:100;
    hftf(:,i)=hftf(:,i).*ffilt;          % Multiply the FFT copy by the filter
end
hfilt=ifft(hftf,'symmetric');            % inverse FFT to obtain 100 time series
end
```

Red bit is where I work out which frequencies and periods each of the Fourier coefficients will correspond to. There is the constant (zero freq), then the wave with period equal to the entire time series (freq 1/T), and then 2,3,4… times this, then reverse (odd number of points, so no Nyquist). Period is 1/frequency, but taking care with f=0 (to avoid an infinity).

Filtering: simple example



Top panel shows a time/longitude section of sea level anomaly (metres) from altimetry (in the tropical Atlantic).

FFT each time series, filter out periods longer than 270 days or shorter than 30 days, inverse FFT

Lower panel show the result – only data at periods 30 to 270 days.

Removed a lot of annual and longer variability which was hiding propagating features, but there are still a lot of vertical stripes – coherent across the entire distance, but not visibly propagating – which are hiding the propagating features

Try a 2D filter, keeping only wavelengths between, say, 5 degrees and 12.5 degrees (1/5 and 1/2 of the domain length), as well as periods between 30 and 270 days.

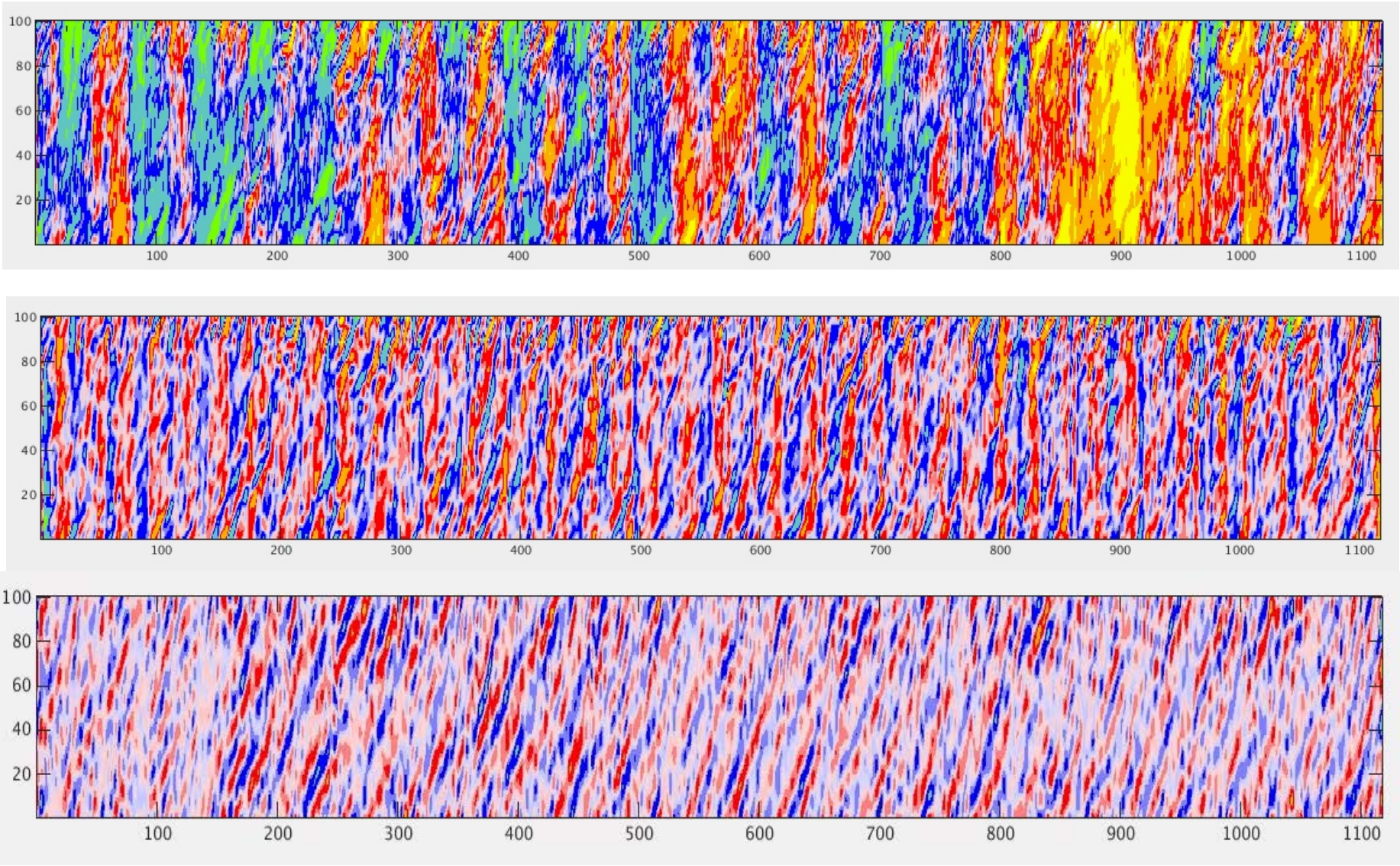
Applying a 2D filter, using 2D FFT, is exactly equivalent to applying a 1D filter in time, and then applying a 1D filter in space, if the 2D filter is made from the product of the two 1D filters (other filters which can't be made of such products are also possible).

```
function [hb,hfilt]=altim_filt2d(pmin,pmax,wlmin,wlmax)
%
% filters to pass only periods pmin to pmax (in days)
%               and wavelengths wlmin to wlmax (in degres) from altimetry
%
[h,hr]=readavisolat(300);                 % 1117 points in time, every 7 days
hb=hr(:,1301:1400);                       % select 100 longitudes
hft=fft2(hb);                             % 2D FFT in space and time
hfreq=[0,1:558,558:-1:1]'./(1117*7);      % frequency in cycles per day
hper=1./hfreq;                            % period in days (first is infinity)
ffilt=1+hper*0;                           % Create 1D filter, 1 for frequencies to
ffilt(hper < pmin)=0;                     % pass, zero elsewhere
ffilt(hper > pmax)=0;
hwn=[0,1:50,49:-1:1]'./(100*0.25);        % wavenumber in cycles per degree
hwl=1./hwn;                               % wavelength in days (first is infinity)
gfilt=1+hwl*0;                            % Create 1D filter, 1 for wavelengths to
gfilt(hwl < wlmin)=0;                     % pass, zero elsewhere
gfilt(hwl > wlmax)=0;
fgfilt=ffilt*gfilt';                      % Combine 1D filters into 2D filter
hftf=hft.*fgfilt;                         % Apply filter to 2D FT
hfilt=ifft2(hftf,'symmetric');            % Apply inverse 2D FT
end
```

It can be tricky to cope with the different cases (odd/even length), and remember how to relate position in the FFT to frequency or period.

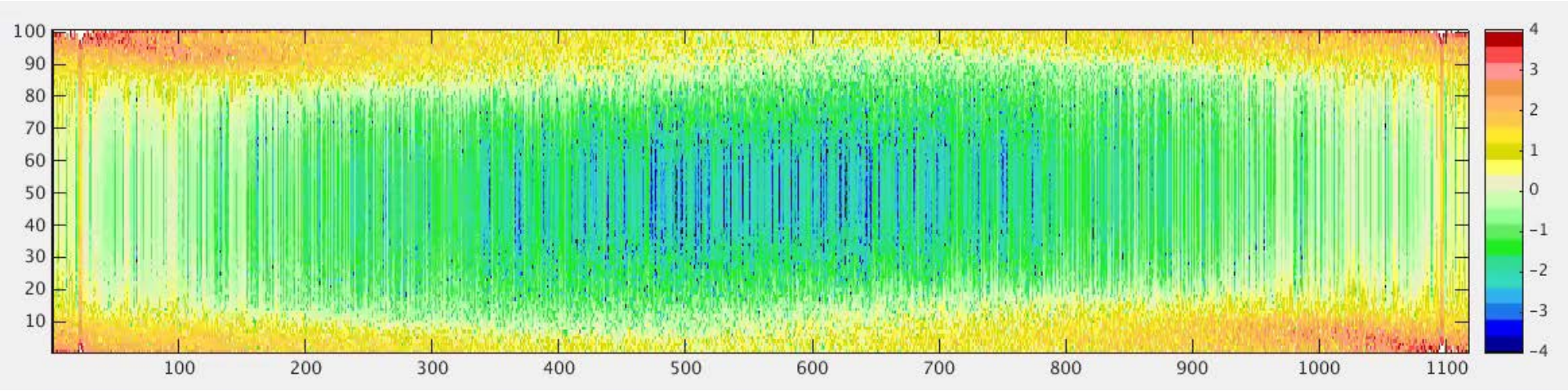
This code will create the appropriate list of periods and frequencies. Given a column array (or 2D array of multiple columns), hin, with time series in the columns, and also given dt = the time interval between data points (in units that suit you, e.g. years), it will calculate a list of frequencies (units e.g. cycles per year) and periods (units e.g. years) associated with the corresponding places in the FFT of hin.

```
function [freq,period]=fouper(hin,dt)

[N,M]=size(hin);  % N=length of time series, M=number of time series
ny=mod(N+1,2);    % ny = 1 for even length (Nyquist exists),
                  %      0 for odd (no Nyquist)
T=N*dt;           % total time period spanned by data
N2=(N-1-ny)/2;    % number of sine or cosine waves,
                  % excluding mean and Nyquist
freq=[0:N2+ny,N2:-1:1]'./T; % frequencies (cycles per unit time)
                            % associated with fft values.
                            % Zero, 1/T, 2/T ... and reversed.
period=1./freq;   % Note, first period will be +infinity

end
```

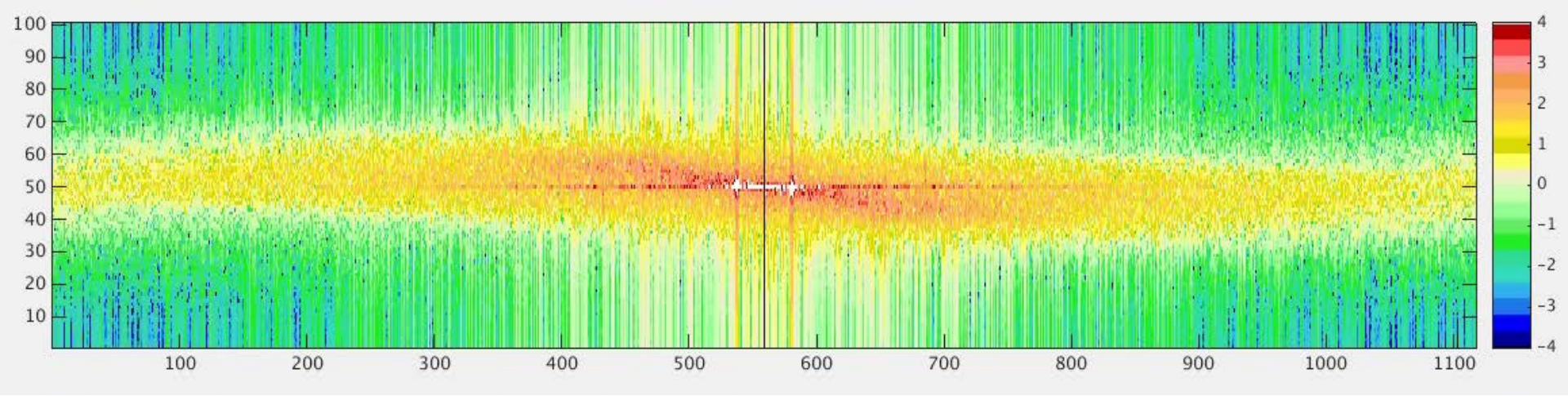


Now the vertical stripes are greatly reduced, and the propagating features really stand out. BEWARE: if you pick only a particular period and wavelength, that is what you will get out. Here, filtered random noise would give similar stripes, but tilted equally often in the opposite direction.

Log_{10} of the amplitude squared coefficients from the 2D FFT of the original image



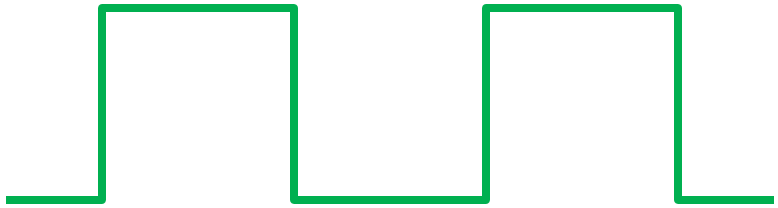
The same, but shifted so that the low frequencies/long wavelengths are in the middle



There's a Matlab function to do this shift: xfts=fftshift(xft).

In this 2D case, preferential propagation direction shows up as the asymmetry (twist) in the energy.

What we are doing here is known as “boxcar filtering”, because your filter is either 1 or zero, so it looks like a boxcar:



This has some disadvantages. Next week, we will cover more subtle kinds of filter, estimation of spectra, cross-spectra and (lagged) autocorrelation and cross-correlations.