NTNU – Trondheim
Norwegian University of
Science and Technology

The Fourier transform

1. Preliminaries and definitions
2. Heyser plots
3. An experiment with sound

Calculating with transforms

- The usual way this goes, is
  - Rewrite a function as a sum of other functions (without losing data)
  - Do something that is easier to do with the other functions than with the original
  - Add up all the other functions to get the original one back again (with some changes)
- The Fourier transform rewrites the original function as an endless pile of sine and cosine functions
  - In order to do it with a computer, we sacrifice a little bit of accuracy to get a finite (but very long) list of sines and cosines
  - At some point in math classes, somebody usually calls the sin/cos representation “the frequency domain”

Sines and cosines

- The notion that any function can be built out of sine and cosine waves has a kind of intuition
  - The ‘unit’ we’re working with spans the whole range from -1 to +1
  - It’s easy to amplify, stretch, and shift
- In order to reduce the number of things we need to think about at once, we can treat a pair of related sine/cosine functions as one complex function:

$$re^{i\omega} = r(cos\omega + isin\omega)$$

  - The *”frequency domain”* has to do with how we deal with the meaning of the omega

The transformation

- Given a function f(t), its transform F(k) is

$$F(k) = \int_{-\infty}^{\infty} f(t) e^{-i2\pi kt} dt$$

  - What we're doing is to multiply f with a wave along the entire t-axis, and adding up a number that represents the degree to which they overlap
  - Each k says how quickly the wave we're checking goes up and down
  - When we do it for a lot of different k-s, we get a function that says to what extent the oscillation rate of each k matches any oscillations in f(t)

The discrete transformation

- From the continuous transform, we can (in principle) integrate from -∞ to ∞ for infinitely many different k-s
- Computers don't do infinity very well, so to do this numerically, we can split the transform up into N pieces (N t-s and N k-s) that we're interested in
- It becomes

$$F(k) = \sum_{n=0}^{N-1} f(t_n) e^{-i2\pi \frac{k}{N} n}$$

- We can evaluate this with a for-loop from 0 to N-1

  ...and get N k-values out of it

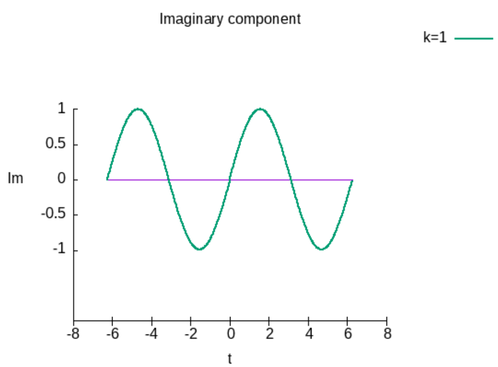
The frequency domain

- A natural question at this point is to wonder
  “what exactly does k=0, k=1, k=… <u>mean</u> here?”
- In a sense, it’s just a number – we haven’t attached any units to it
  - When we put in some values and calculate answers, the f(t) we know and love disappears in a pile of answers
  - We can put the answer-pile into the inverse transform formula and get f(t) back again, but how to interpret the pile of numbers?

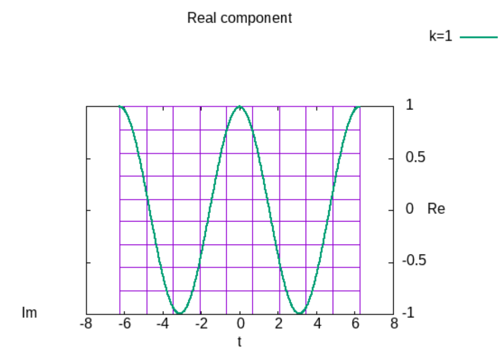
k=0 is easy, k=1 fits our N

- When k is 0, the 'wave' we're extracting from f(t) doesn't really wave at all, it becomes the constant 1
  - Thus, F(0) simply becomes the integral of f(t)
- When k is 1, the wave goes through 1 of its periods, stretched out to match the length of f(t)
- Remember, we've got two wave-shapes that can go up and down and back again in that interval:

Sine:

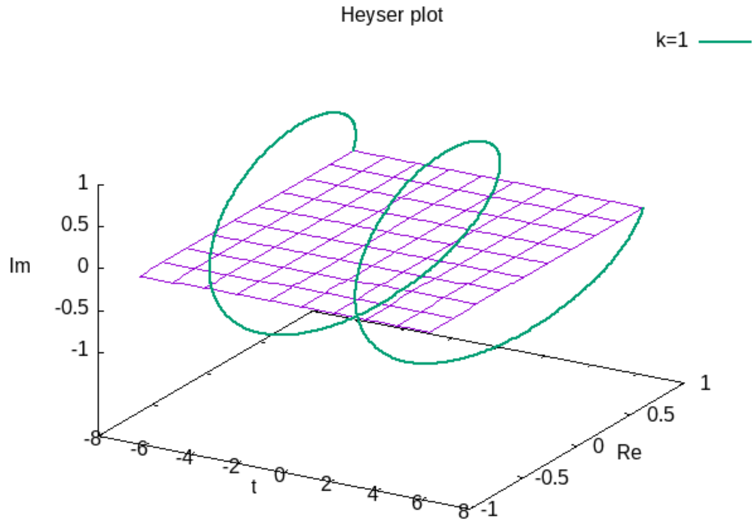


Cosine:



Heyser plots

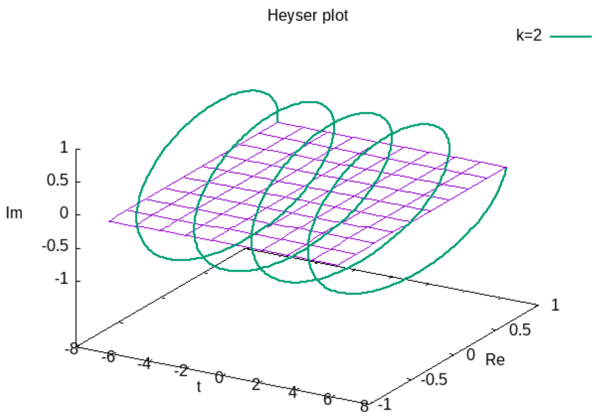
- Since we only want to deal with those two combined into one complex number, we can draw a 3D curve that captures both along the t-axis
- Here's such a plot that goes from -2π to 2π
  - Two full periods of sines and cosines



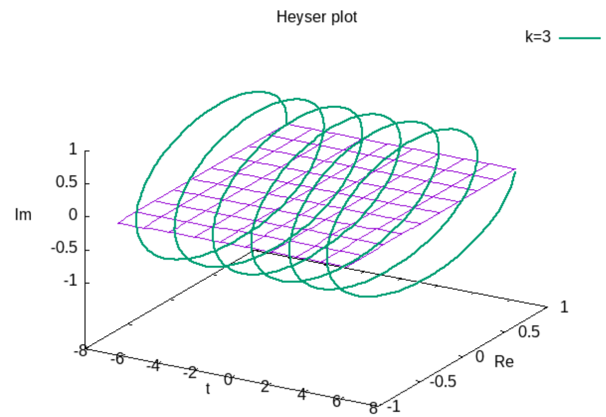
NTNU – Trondheim
Norwegian University of Science and Technology

k says how quickly the spiral turns

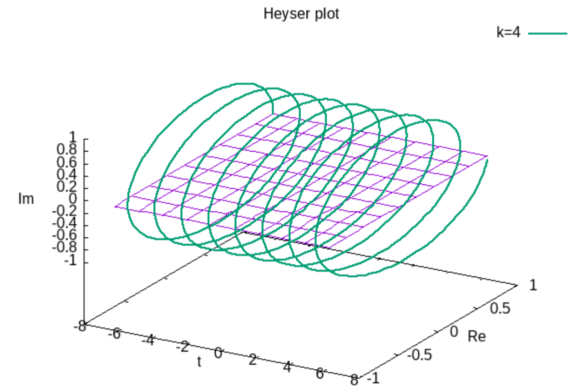
- Here are a few more k-s over the same interval:



k=2



k=3



k=4

Swing on the spiral

- When we multiply one of these with a real f(t), we get a complex number out
- For k=1
  - The real part says how well f matched with 1 period of cosine
  - The imaginary part says how well f matched with 1 period of sine
- For k=2
  - It’s the same, but with 2 periods of cosine/sine
- For k=3, it’s 3 periods

...etc...

We can now predict something

- If we feed this method an input that only contains a pure sine function,
  - Most k-s should have pretty low values
  - The k where the sine function perfectly strikes the sine-part of the spiral should have a remarkably high value (perfect match)
  - That spike will appear in the imaginary component of the answer
- Let’s try it!

Code archive contents

- There are a few more moving parts this time
- The mathematically interesting bits:
  - generate_signal.c *writes a simple sine wave into 'wave.dat'*
  - plot_wave.gp *draws a picture of it*
  - naive_dft.c *reads it, and writes transform in 'real.dat'+'imag.dat'*
  - plot_dft.gp *makes a picture that superpositions both components*
  - invert_dft.c *reads {real,imag}.dat and recovers wave in 'recover.dat'*
  - plot_recover.gp *makes a picture of the recovered waveform*
- These should be enough to demonstrate that we can transform a waveform and get it back again from its frequency spectrum

This is awfully slow

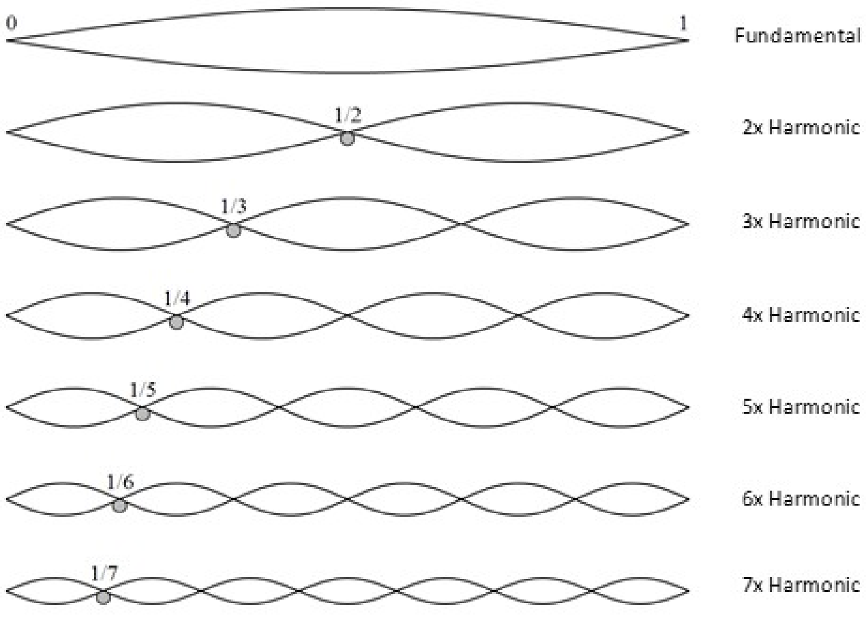
- naive_dft.c takes close to 6 minutes (on my laptop)
- Surely we can do something, spirals where k-s are multiples/factors of each other repeat a lot of the same stuff

Harmonics

- Waveforms do repeat themselves
  - At 2x frequency, we could reasonably expect that we could derive the 2nd half-wave from twiddling with the 1st half and flipping some signs

    ...and 3 quarter-waves from the 1st quarter at 4x

    … etc ...



Trying it once

- The naive translation runs in N^2 time
- If we halve the problem size, it should at least quarter the time
- odd_even_dft.c splits the problem into odd and even indices, and gets the second half of the transform from combinations of the first half

Trying it recursively

- cooley_tukey_fft.c halves the domain, then applies the same trick to halve it again, etc.
- The downside of this is that it requires the input length to be a power of 2
- The upside is that it's really, really fast
  - This demo-implementation is a bit silly for readability, but it has quite a striking effect still

Doing something fun with it

- I’ve included a program to convert waveforms into audio files
- Sound is a nice, 1-dimensional problem where we can separate parts of a wave form and work on it in the frequency domain
- Input files are 131072 values long
  - Chosen because it’s a power of 2
  - It’s also almost exactly 3 seconds of CD-quality mono sound (44100 values per second)
- NOTE: the length of the input determines how long 1 wavelength is at k=1
  - Since we have ~3 seconds, 3 full waves (k=3) corresponds to 1Hz
  - If we had 7 seconds, k=7 would be 1Hz

Code archive contents pt. II

- There are a few extras to make things tangible
  - sound.c *converts wave.dat to 'sound.wav' which can be played back through your speakers*
  - generate_a_major.c *generates an A-major triad as a superposition of 3 waves*
  - generate_a_minor.c *generates an A-minor triad as a superposition of 3 waves*
  - The input and output files are the same as with the default generate_signal thing that just makes a single sine wave
  - Just so we get some more interesting data to manipulate

Code archive dependency

- I have finally taken my own advice, and used a library
  - The inverse transformation is implemented using FFTW3
- Primarily, this saves me from writing it myself
  - It's not actually that hard when you have the forward version, but I'm sure we have enough code to look at anyway
- Secondarily, it serves as a demo of FFTW3 use

  ...and proves that the way our representation is laid out in memory is actually compatible with the rest of the world…

  (which was not the case with our pointer-massacre of sparse matrices)
- It requires you to install the library, though
  - The package is called *libfftw3-dev* on {Ubuntu/Mint/Debian...}

NTNU – Trondheim
Norwegian University of
Science and Technology

The Experiment

- The scripts major.sh, minor.sh, and shift.sh generate files and transforms and organizes everything

    ...just to remember the sequence, because filenames are hardcoded for simplicity

    - major.sh generates the A-major triad and saves it as sound
    - minor.sh generates the A-minor triad and saves it as sound
    - shift.sh
        - generates the A-major triad
        - transforms it into frequencies
        - moves the responses between k=750 and k=900 (250-300Hz) down by 47 index positions (roughly 15.55 Hz)
        - inverts the transform
        - makes sound out of the recovered signal
- Hurrah, we've turned a major chord into a minor using frequency analysis!
    - Try plotting the waveform of the major chord, you'll see how hard it would be to modify directly

There's so much more I'd like to say...

- If you feed these programs inputs that approach more than 22050Hz and look at the transform plots, you will rediscover the Nyquist sampling theorem
- We could have complex inputs
- We could have multi-dimensional inputs
- We could have things that fluctuate over space instead of time
- We could use it to solve differential equations

...but we don't have the time.

- Hopefully, though, you can look at the calculation from an additional perspective and invent things to use it for

(that is, provided that you hadn't tried all this from before)