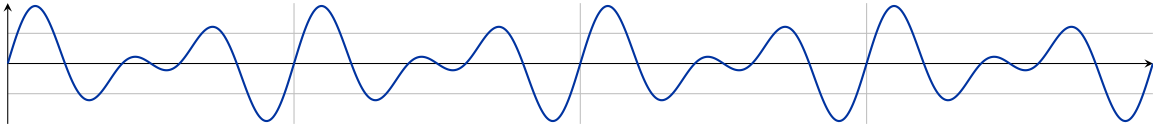# The Fast Fourier Transform

Zachary Todd Edwards
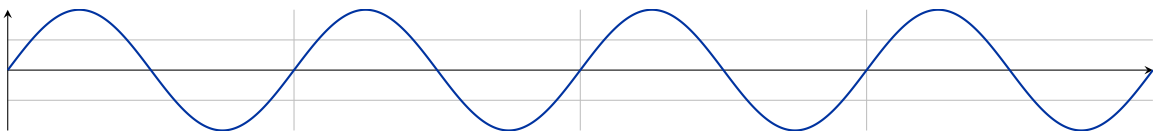
July 10, 2023
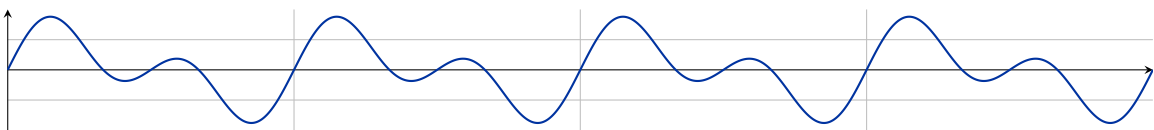
1   A Hard Problem

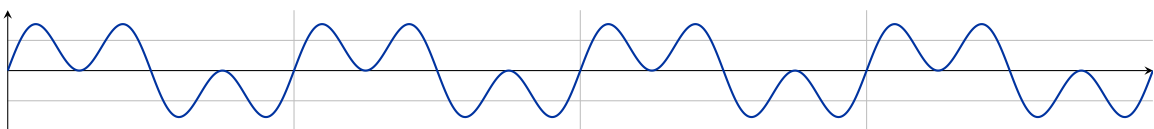Lets say we have a wave that we want to find the component frequencies of:

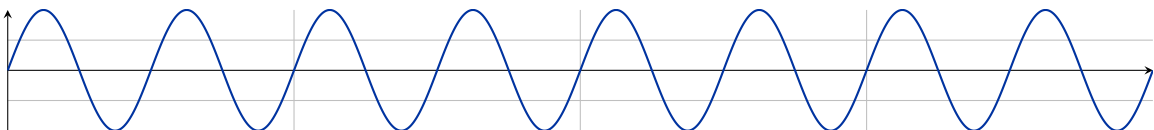We could try a brute force approach, look at the graphs of sums of sine waves until we get our original:
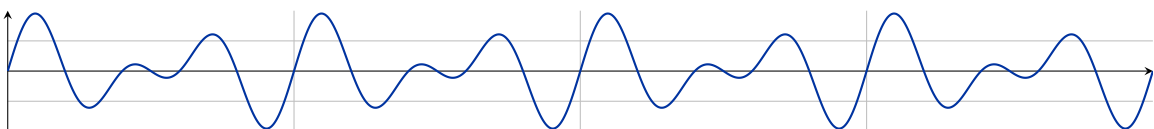
1Hz + 1Hz

1Hz + 2Hz

1Hz + 3Hz

2Hz + 2Hz

2Hz + 3Hz

We made quite a few assumptions doing this though. We assumed that our wave's amplitude was centered on the horizontal axis and not shifted further up or down, we assumed the phase of our waves is perfect and not shifted left or right, we assumed all of the component frequencies are integers, and we would need to define a range of acceptable error or expect the input wave to be perfect. The first two might also be brute forcible but we are already dealing with a solution too complex to be practical.

Lets try visualizing it in a different way by wrapping our waveform around the origin of a graph:
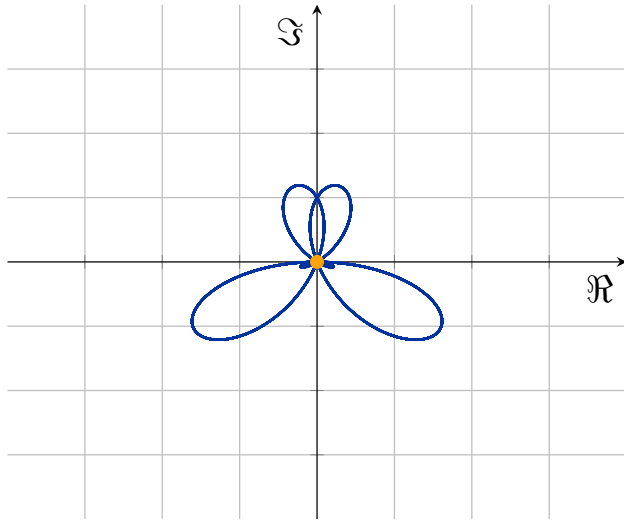
Rotation Period = 1
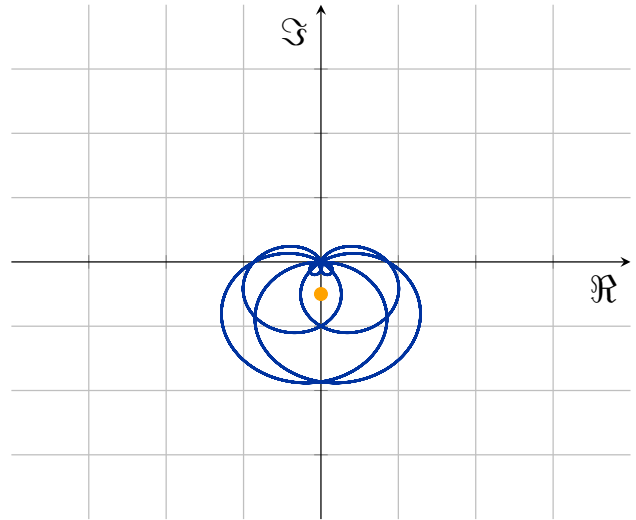
Rotation Period = 3

Rotation Period = 2

Rotation Period = 4

The dot is the center or average of this wound up version of our waveform, and we can see that it is almost always very close to zero except when the rotation period is 3 or 2, which is also the component frequencies of our wave!

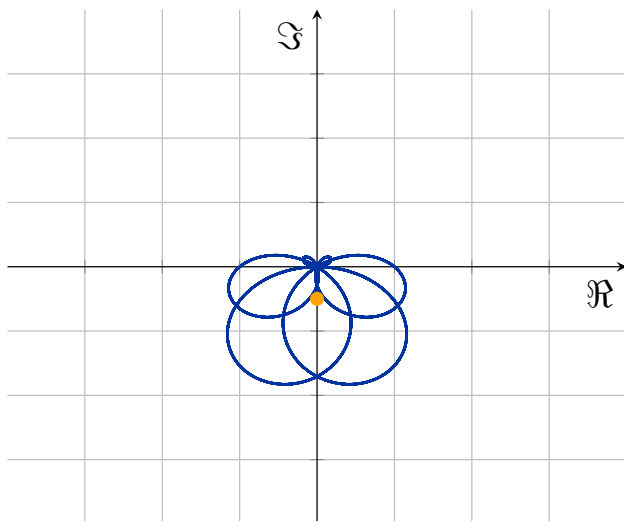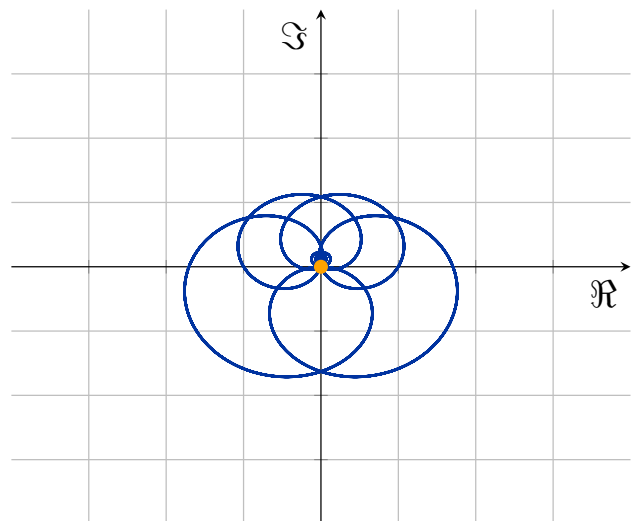## 2   Euler's Formula

Okay thats great but what is the actual math behind this? Wrapping the function around the origin is not any simpler visually than brute forcing, and even though we only had to check each frequency once getting that average is an extra step. Also, what is rotation period and how do we change it? The axis labels have already given away part of the secret, these wrappings are being graphed on the complex plane!

If you have spent any amount of time studying math you have likely ran into the so-called most beautiful equation in mathematics: $e^{i\pi}+1=0$. This is Euler's identity and its big cousin Euler's formula is the secret to this winding. How does it work though? Why does $e^{i\pi}+1$ equal zero?
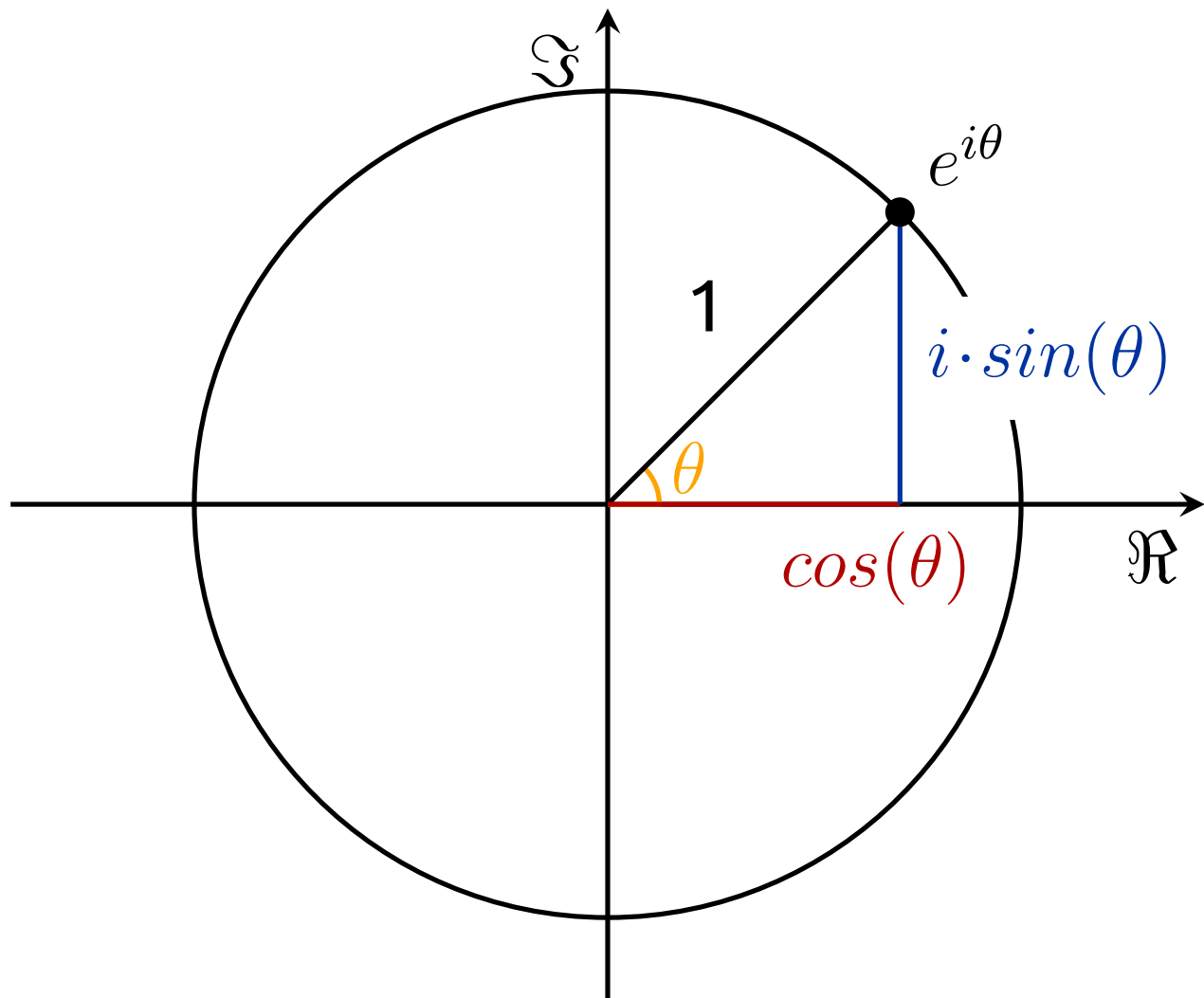
The Complex Unit Circle

The very short version is that it is a simplification of the full formula which is $e^{i\theta}=cos(\theta)+i\cdot sin(\theta)$. It is also worth noting that this is continuous and works for angles larger than 2π or less than 0.

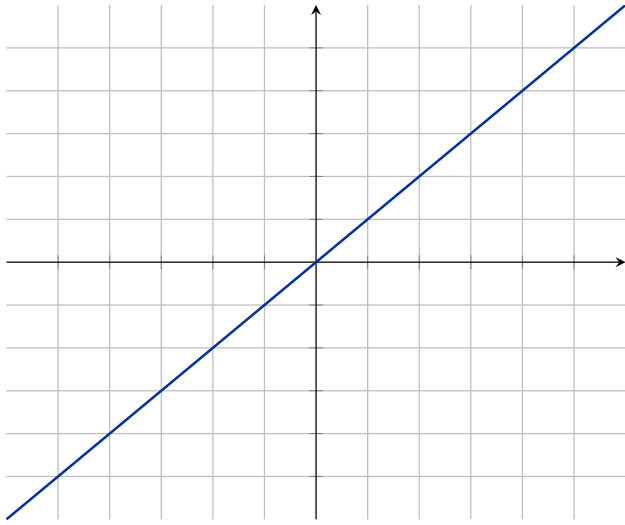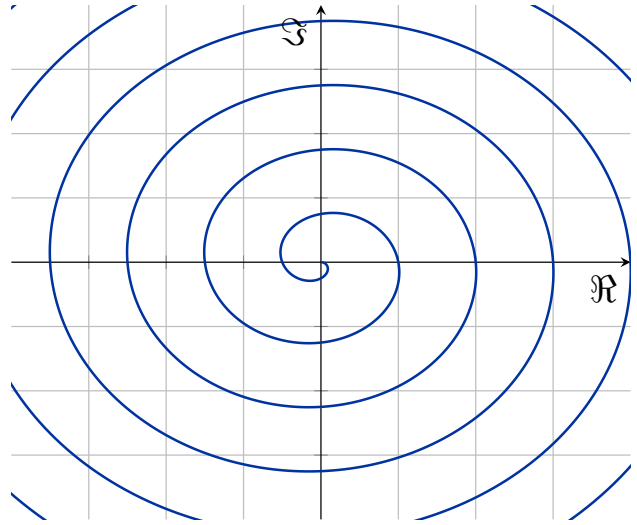## 3 Spiralling a Function

If Euler's formula always has a distance from the origin of 1 rotated by θ radians and multiplying anything by 1 is itself, could we just plug in our variable times 2π for θ and then multiply our function with that to wrap it around the origin? This is math, there is no way it could be that easy, could it?

$$F(x)=x$$

$$F(x)\cdot e^{ix2\pi} \text{ where } x\geq 0$$

It really is that easy! Above is a spiraled version of the equation $F(x)=x$, for positive x so it is more readable, we can see by looking at the positive real axis that for every one full rotation it increases by one unit of distance from the origin. This can be seen in all of the axes, with the other ones having quarter, half, and three-quarter offsets from the last integer.

What about rotational period, what does that even mean? That is how many full rotations we do for one unit increase to the input of our original function, and how we can do this is already sneakily built in to the original formula, for a period of one we multiply 2π by one, for a period of two we multiply it by two, etc. If we omit 2π we get a rotation period of $\frac{1}{2\pi}$, we can think of it as dividing 2π by itself.
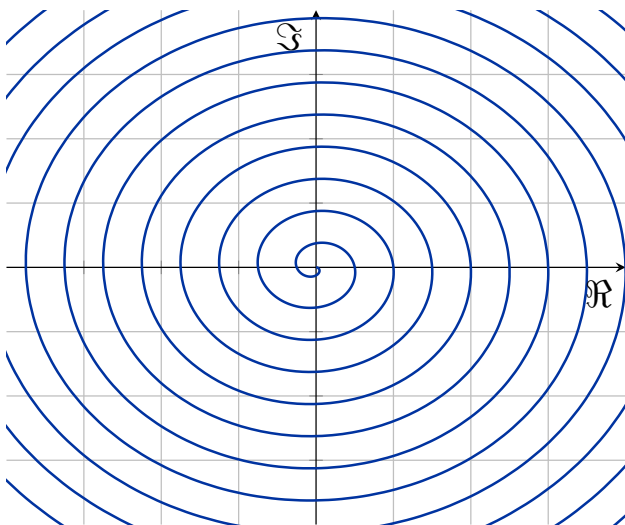
$$F(x)\cdot e^{ix2\cdot 2\pi} \text{ where } x\geq 0$$

$$F(x)\cdot e^{ix3\cdot 2\pi} \text{ where } x\geq 0$$

4
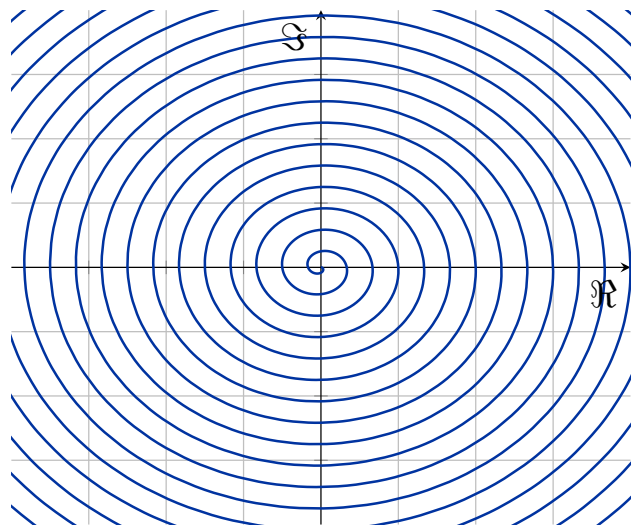
4   The Continuous Fourier Transform

There is one last piece needed to get our winding solution to work: the average. Conveniently for us we have a quick way of getting the total area under a curve, we can take its integral and divide by the total time the wave occupies:

$$\hat{G}(f) = \frac{1}{N}\int_0^N G(x) \cdot e^{2\pi i f x} dx$$

This works! When graphed we will get something like this:
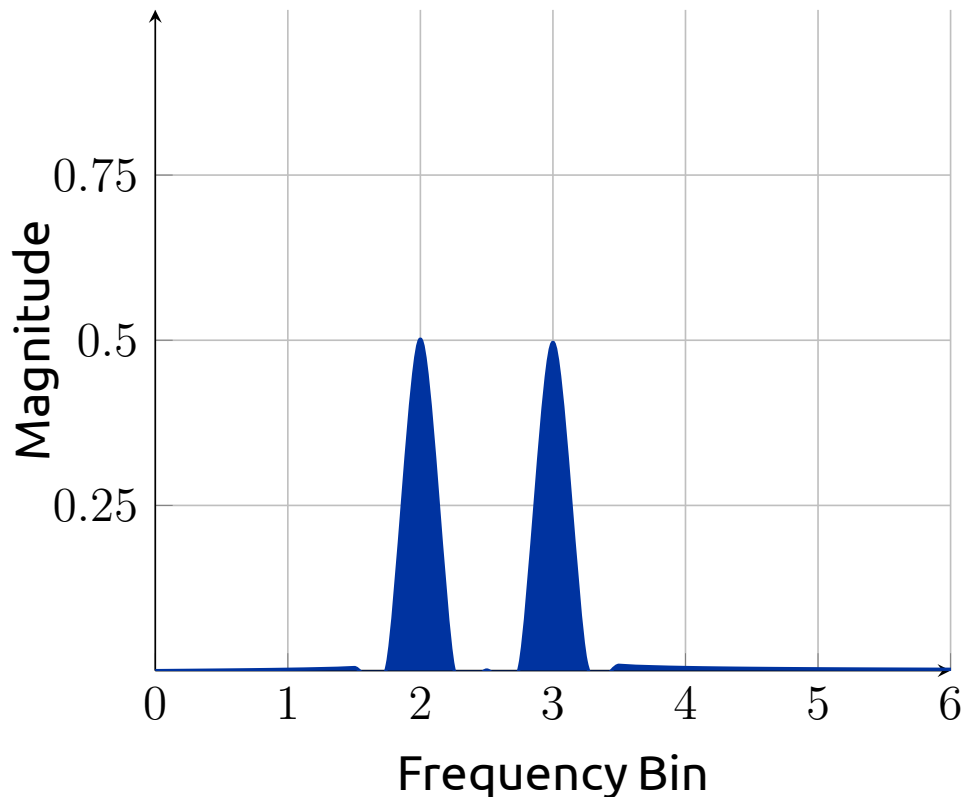
$\hat{G}(f)$ for $G(x) = sin(2 \cdot 2\pi x) + sin(3 \cdot 2\pi x)$

This is not quite the true Fourier transform though, there are a few adjustments to get there. The true Fourier transform does not take the average, just the total. With the average it is sometimes called the "normalized Fourier transform." The Fourier transform also does not typically specify a time frame, and instead goes from $-\infty$ to $+\infty$, assuming that the wave itself needs to start and stop somewhere. The winding is also usually done in the clockwise direction, so the exponent is negated. Finally, the variable being integrated is usually t. With all this:

$$\hat{G}(f) = \int_{-\infty}^{\infty} G(t) \cdot e^{-2\pi i f t} dt$$

There is just one problem: computers tend to not like continuity or infinity.

5   The Discrete Fourier Transform

Well if computers do not handle continuous functions very well then how will we get the input waveform in the first place, let alone process it? The answer might seem obvious, just take discrete points, also called samples, on the waveform at regular intervals and perform the Fourier transform on them. Lets try it, we will take a high number of samples from our original wave, wrap them around the origin, and sum up their results when multiplied with $e^{i\theta}$ at regular intervals.
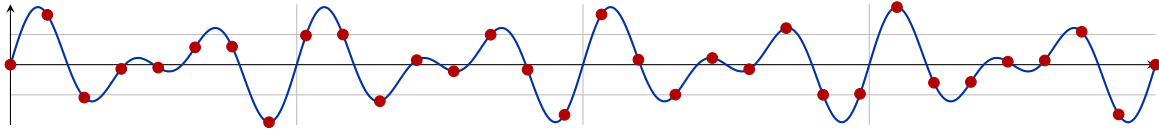
32 samples across 4 seconds, for a sample rate of 8 per second.
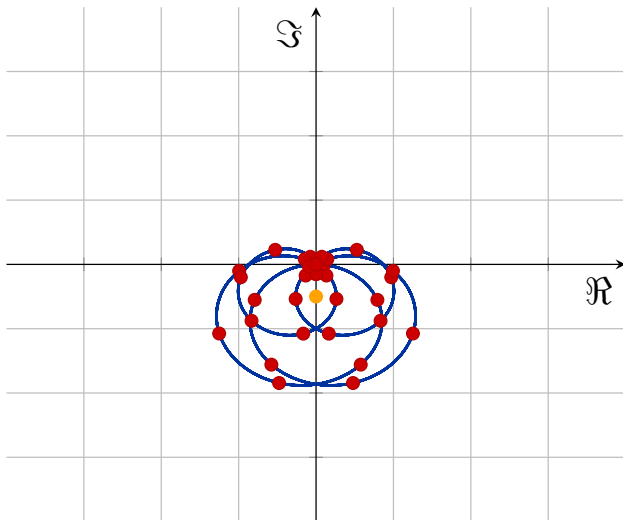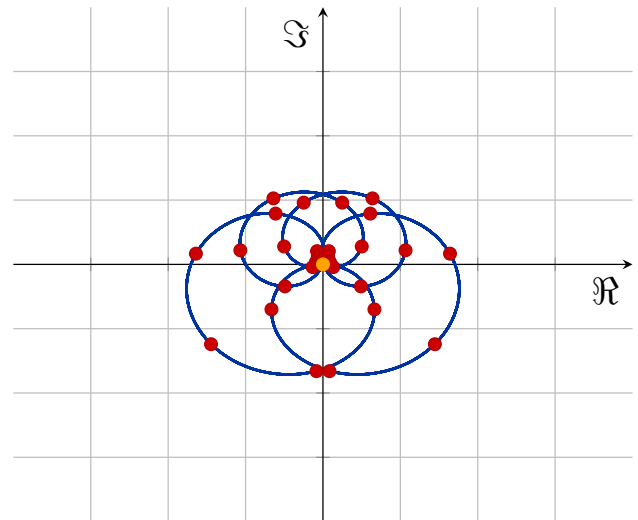
Rotation Period = 3   Rotation Period = 4

The discrete version of the Fourier transform should be relatively easy to reverse engineer:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn}$$

It is convention to use x_n for the input set and X_n for the output set, both of size N. We might also want to normalize the output:

$$X_k = \frac{1}{N}\sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn}$$

With all that we get the following for our graph:
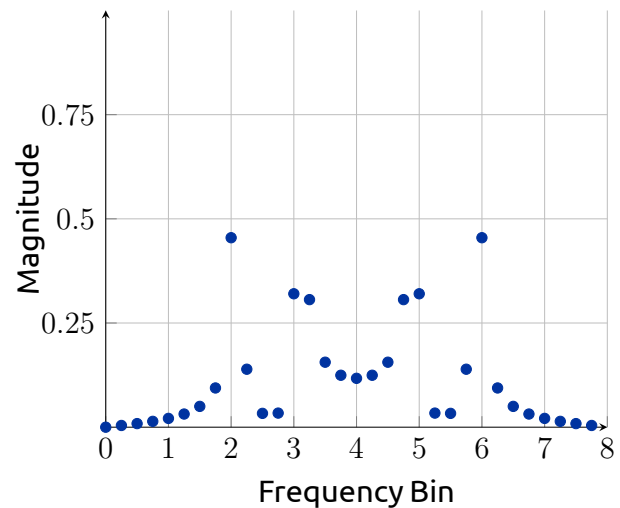
This doesn't look right, there are peaks at 2hz and 3hz but also 5hz and 6hz, and it looks symmetrical. What is going on?

6

6 Nyquist-Shannon

Our graph of the discrete Fourier transform above is actually correct, but the oddness in its results hints at a deeper problem we are bumping up against: how do we know how many points are enough?
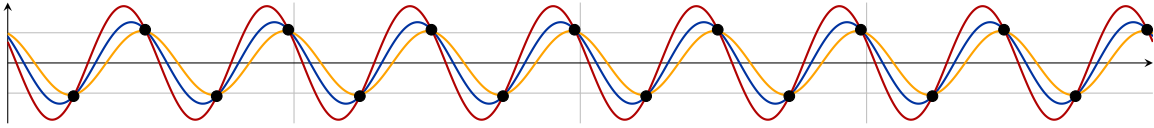
All of these waveforms have the same samples at the same sample rate. The problem is not just that these different waves produce the same samples in this one case, its that these samples have many different waves that could fit them. It gets worse, there are infinitely many waveforms that could fit any set of discrete points, we need to know some constraints on the waveform coming in in order to do anything useful with the samples.

The symmetry of our spectrum graphs above actually give us a hint about it, here is one with double the sample rate:
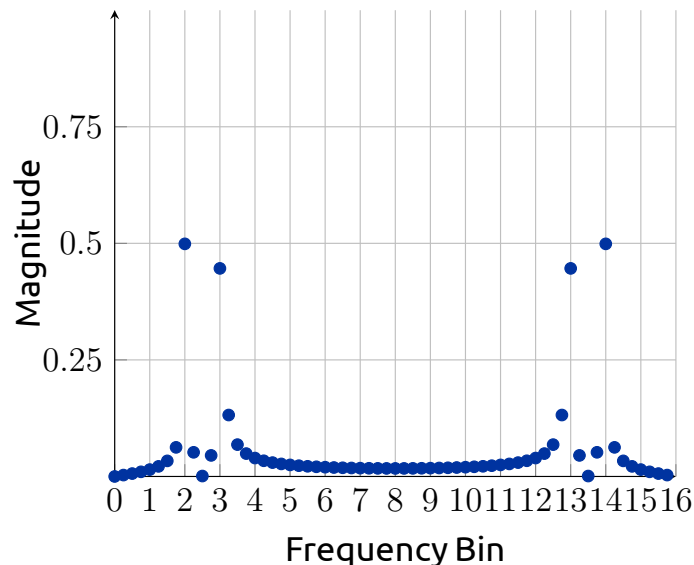
The results got more accurate, as expected, but the center of symmetry doubled as well. This "center of symmetry" is called the Nyquist limit, and is the highest frequency that discrete samples can represent without a loss in fidelity. Conveniently the Nyquist limit is just half of our sample rate, non-inclusive. The symmetrical second half is the complex-conjugate of the first half, and is usually ignored when working with real-numbered inputs.

Which frequency a particular index corresponds to, its frequency bin, can be gotten with $k\frac{f_s}{N}$ where k is the index of the result, f_s is the sample rate per-second, and N is the total number of samples. The magnitude, absolute value, of each entry is usually what is graphed in charts like the ones above. The imaginary part of each entry tells us that frequency's phase, how much it is shifted left or right.

7 Complex Programming

Before we can start translating this math into code there is one last hurdle: how do we write programs with complex numbers? The good news is that most programming languages include complex numbers as part of their standard library, or as part of a common third party library.

C does some macro magic to "overload" the operators for complex types, but cannot overload the functions so there is a separate set of math functions for them, as well as a few macros for assignment:

```c
#include <complex.h>
void dft(double complex *x, double complex *X, int N){
  for(int i = 0; i < N; i++)
    X[i] = CMPLX(0.0, 0.0);

  for(int k = 0; k < N; k++)
    for(int n = 0; n < N; n++)
      X[k] += x[n] * cexp(-(((I * 2 * M_PI) / N) * k * n));
}
```

C++ is similar but uses templates:

```cpp
#include <vector>
#include <complex>
void dft(std::vector<std::complex<double>> &x,
         std::vector<std::complex<double>> &X, int N){
  X.clear();
  X.resize(N, 0);

  for(int k = 0; k < N; k++)
    for(int n = 0; n < N; n++)
      X[k] += x[n] * std::exp(-(((1.0i * 2 * M_PI) / N) * k * n));
}
```

Perl:

```perl
use Math::Complex;
sub DFT{
  my ($x, $X) = @_;
  my $N = scalar(@$x);
  @$X = ();

  for(my $k = 0; $k < $N; $k++){
    for(my $n = 0; $n < $N; $n++){
      $$X[$k] += $$x[$n] * exp(-(((i * 2 * pi) / $N) * $k * $n));
    }
  }
}
```

The complexity of this algorithm is $O(N^2)$, however, and we are going to want to be able to process frequencies in the gigahertz range. To make this practical we need to somehow speed this up.

8 A Simpler Problem

Lets tackle a similar but simpler problem, how can we get a set of points on a polynomial?

Say we have a class for our points that looks like:

```
1   class point{
2     double x, y;
3     point(_x, _y) : x(_x), y(_y){}
4   };
```

Next our function prototype looks like:

```
1   vector<point> polynomial(int degree, double coefficients[], int n);
```

We might try to use the brute force method:

```
1   vector<double> points(n);
2   for(int x = 0; x < n; x++){
3     double y = coefficients[0];
4     for(int j = 1; j <= degree; j++){
5       y += coefficients[j] * pow(x,j);
6     }
7     points[x] = point(x,y);
8   }
9   return samples;
```

However, this takes $O(nk)$ time where n is the degree of the polynomial and k is the number of samples needed, this is also not counting the not-insignificant time pow() can take. Is there a faster way?

9 Symmetry

Lets look at a few graphs of polynomials:
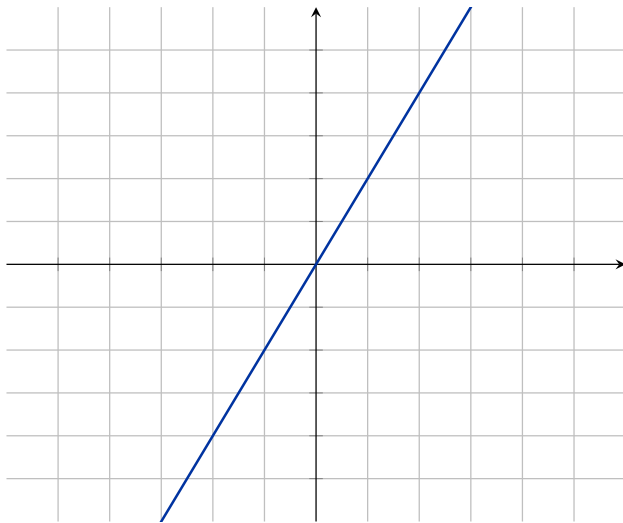
$$F(x) = 2x$$
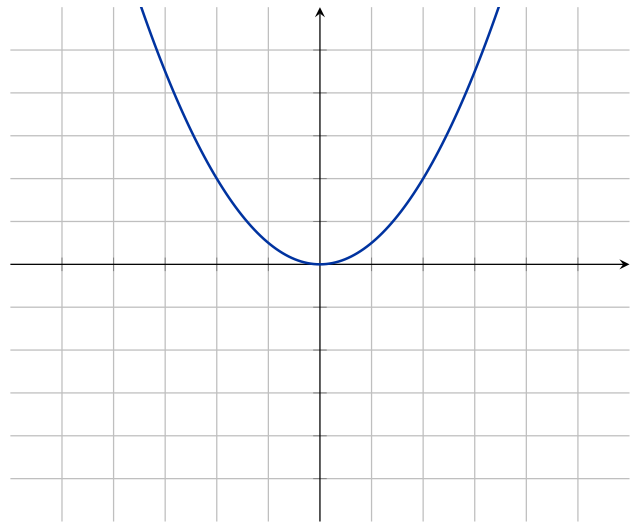
$$F(x) = \frac{1}{2}x^2$$
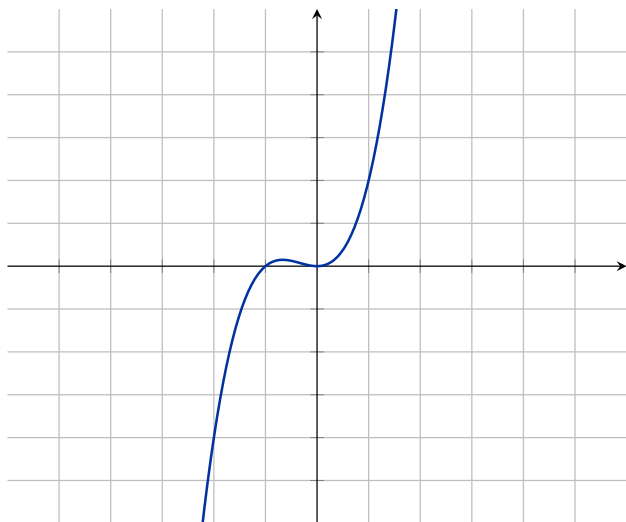
$$F(x)=x^2+x^3$$
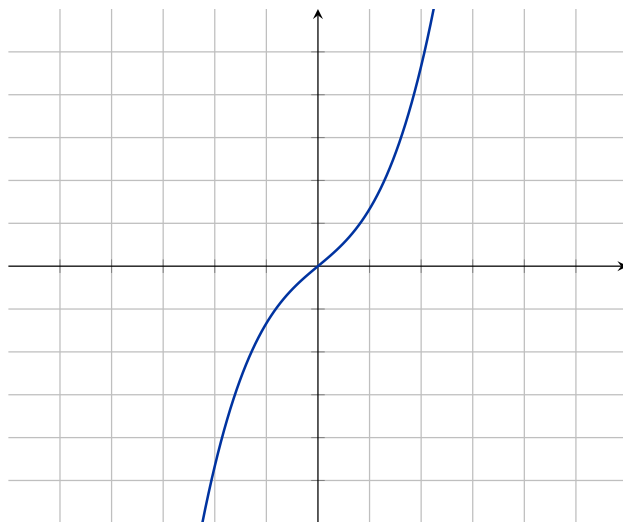
$$F(x)=x+\tfrac{1}{3}x^3$$

$$F(x)=3x^3-x^4$$
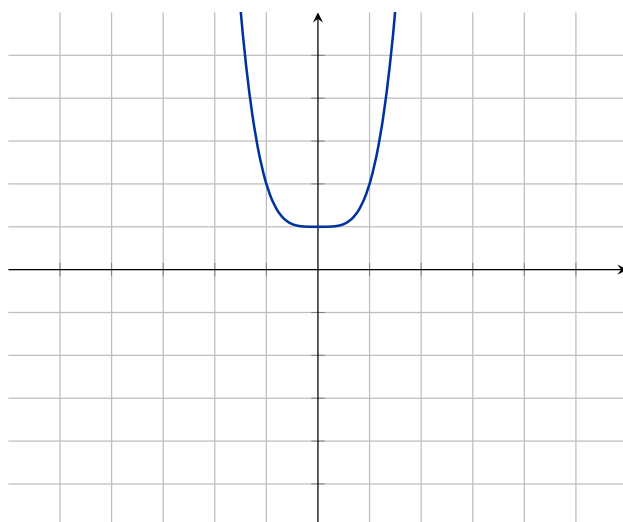
$$F(x)=1+x^4$$

Some of these are symmetrical across the vertical axis, some of them are symmetrical across both axes, and some of them look almost symmetrical but aren't quite. What is the difference in the polynomials that are symmetrical and those that aren't?

The polynomials with only even-powered terms or odd-powered terms are symmetrical, with the former being across the vertical axis: $F(x) = F(-x)$, and the latter being symmetrical across both: $F(x) = -F(-x)$. Most polynomials don't only contain terms of the same parity, so how can we take advantage of this?

10 Polynomial Decomposition

Lets take the following rather wooly polynomial:
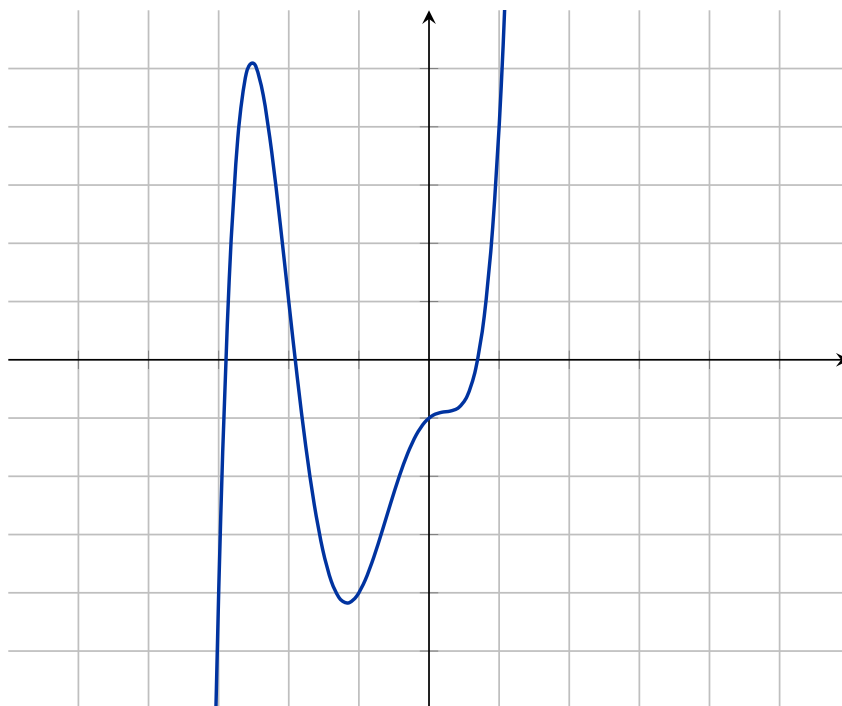
$$F(x) = -1x - 3x^2 + 2x^3 + 4x^4 + x^5$$

Lets try separating it into its odd and even terms:
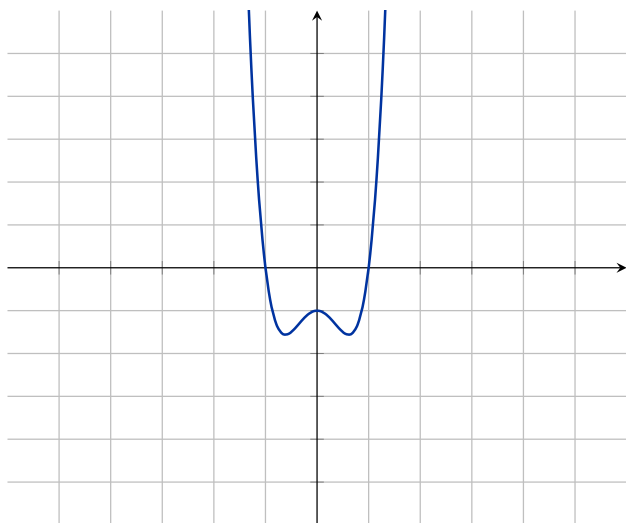
$$F_e(x) = -1 - 3x^2 + 4x^4$$
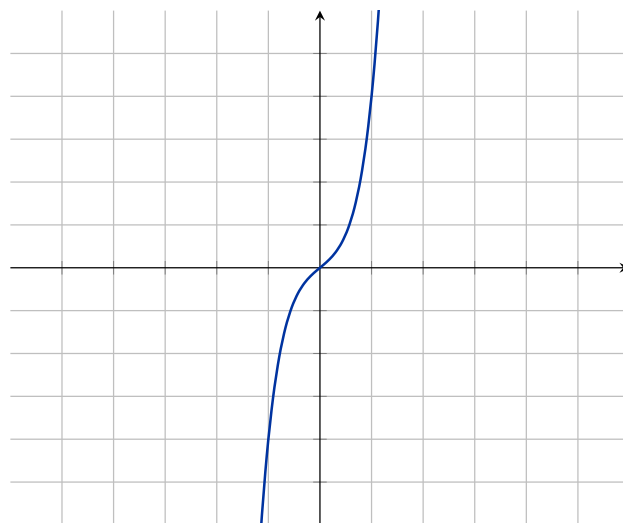
$$F_o(x) = x + 2x^3 + x^5$$

These are both symmetrical and when added together we get our original polynomial, we might be able to work with this.

Lets try writing another algorithm to take advantage of this, to try to keep it as simple as possible we will assert that n, the number of points we want, is even:

```
1    vector<point> points;
2    assert(!(n % 2));
3    for(int x = 1; x <= n / 2; x++){
4      double yp = coefficients[0];
5      double yn = coefficients[0];
6      for(int j = 1; j <= degree; j++){
7        double term = pow(x,j);
8        yp += term;
9        yn += (j % 2) ? -term : term;
10     }
11     points.push_back(point(x,yp));
12     points.push_back(point(-x,yn));
13   }
14   return points;
```

Gross! Not only is this inelegant, it does not save us that much time either; asymptotically it is the same. Lets go back to our polynomials and see what more we can squeeze out of them.

We can factor an x out of the odd terms so that we get $F_o(x) = x(1+2x^2+x^4)$ or $xF_o(x) = 1+2x^2+x^4$. Putting this together we get $F(x) = F_e(x)+xF_o(x)$ and now that $F_o(x)$ is all even terms it would be the same for positive and negative inputs. This could possibly eliminate the ternary operator above which might benefit performance a bit, but we are still stuck with the same complexity. To get any noticeable benefit we need a way to do this recursively.

We can start by trying to make $F_e(x)$ and $F_o(x)$ polynomials with their own odd and even terms. All of the terms in each of these are even, so we could divide the exponents by two and square the inputs. This gives us $F_e(x) = -1 - 3x + 4x^2$ and $F_o(x) = 1 + 2x + x^2$ resulting in our original polynomial being $F(x) = F_e(x^2)+xF_o(x^2)$.

We have a new problem now though; if we try to recursively apply this method our input is always positive or zero after the first decomposition because it was squared, so the variable we factor out of the odd term polynomial will always be positive, and the whole point of separating out the odd and even terms in the first place was that they behaved differently with negative inputs. There is also the problem of magnitude, this repeated squaring of the inputs actually breaks the equivalence if we try to apply it more than once. We have two inputs that cooperate with this method, 0 and 1, since they are already positive and their squares have the same magnitude, what we need are more ones. We used the complex exponential e^{ix} as a kind of "1" before, would it work here?

11 The Roots of Unity
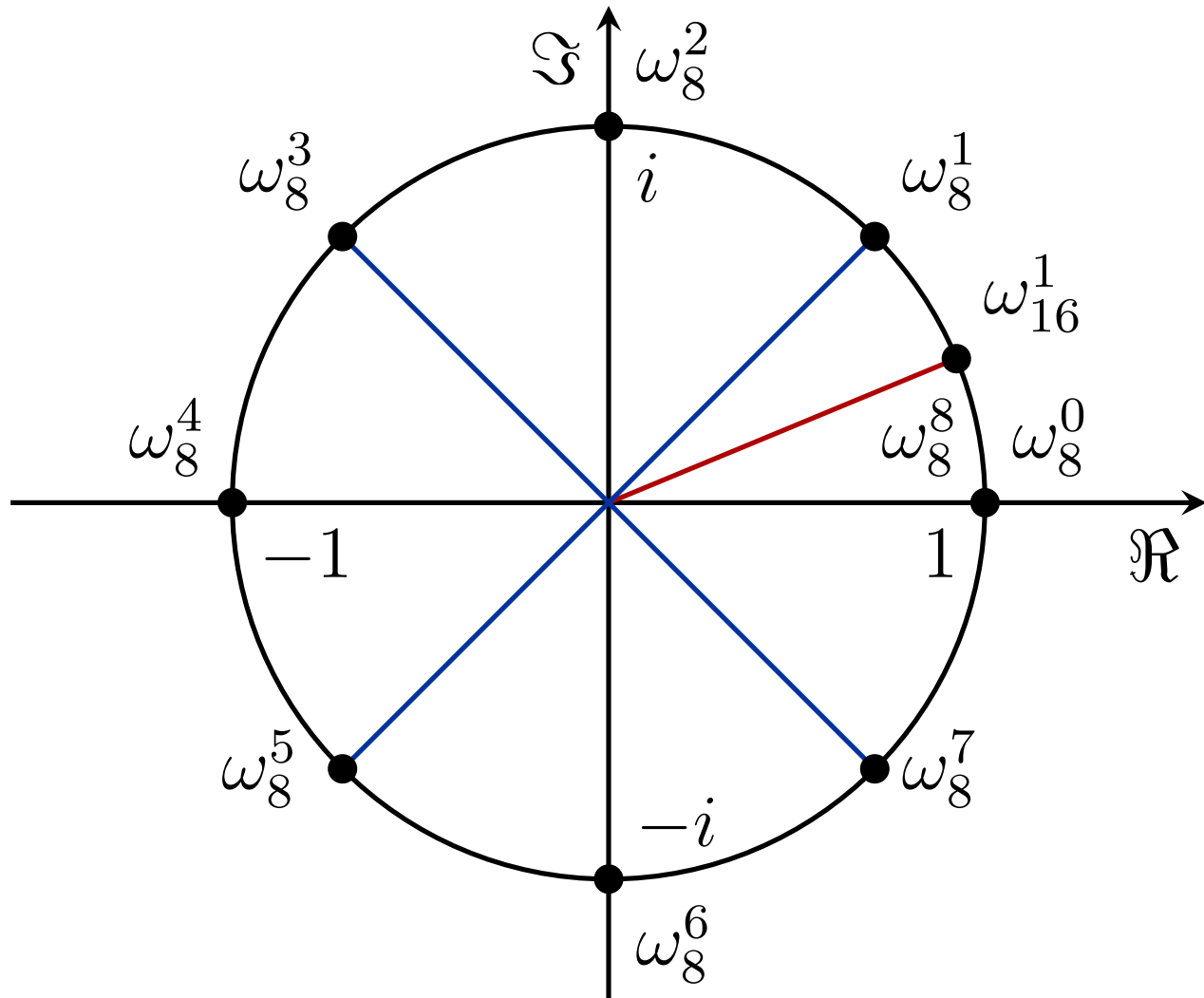
The 8th Roots of Unity and First 16th Root of Unity

For real numbers, $\sqrt{1}$ has two answers, 1 and -1, and this is true for complex numbers as well. Higher roots get more interesting, for real numbers $\sqrt[3]{1}$ has only one answer which is 1, however for complex numbers $\sqrt[3]{1}$ has three answers: $1, e^{\frac{i2\pi}{3}}, e^{\frac{i4\pi}{3}}$. Higher roots have more complex results; the n-th root of 1 has n complex values and these are the roots of unity. The first root of unity after 1 is generally denoted with ω (omega), and the j-th root of unity for the n-th complex root of 1 would be ω_n^j.

The roots of unity have a few useful properties: they always add up to zero, their absolute value is always one, all equivalent ratios of j and n for ω_n^j are equivalent eg $\omega_8^2 = \omega_4^1$, negation "flips" their position eg $-\omega_8^1 = \omega_8^5$, and they are cyclical when raised to a power or multiplied eg $\omega_8^{3^{10}} = \omega_8^3 \cdot \omega_8^{10} = \omega_8^{30} = \omega_8^6$. Since getting the square root of a complex number is relatively trivial these properties lets us quickly get power-of-two-first-roots of unity by repeatedly getting the square root (raising to the power of $\frac{1}{2}$) of ω.

The roots of unity can also be calculated using $e^{\frac{i2\pi j}{n}}$, this is equivalent to the exponential used in the discrete transform.

Complex numbers can be written as $a+bi$, with this the formula for getting the square root of a complex number is:

$$\sqrt{a+ib}=\pm\left(\sqrt{\frac{|z|+a}{2}}+i\sqrt{\frac{|z|-a}{2}}\right)\text{ when }b\geq0$$

$$\sqrt{a+ib}=\pm\left(\sqrt{\frac{|z|+a}{2}}-i\sqrt{\frac{|z|-a}{2}}\right)\text{ when }b<0$$

The absolute value of a complex number is $\sqrt{a^2+b^2}$ and squaring a complex number is the same as real algebra, keeping an eye our for anytime an i gets squared to replace it with -1. If we are only dealing with $a+bi$ we can treat it like a binomial and the squaring formula is a^2-b^2+2abi. Lets run through a few examples to see all of this in action:

$$\sqrt{i}=\sqrt{0+1i}=\sqrt{\frac{|0+1i|+0}{2}}+i\sqrt{\frac{|0+1i|-0}{2}}$$
$$=\sqrt{\frac{1}{2}}+i\sqrt{\frac{1}{2}}$$
$$=\frac{1}{\sqrt{2}}+\frac{i}{\sqrt{2}}$$
$$=\frac{1+i}{\sqrt{2}}$$

$$\left(\frac{1+i}{\sqrt{2}}\right)^2=\frac{1+i}{\sqrt{2}}\cdot\frac{1+i}{\sqrt{2}}$$
$$=\frac{(1+i)(1+i)}{\sqrt{2}\cdot\sqrt{2}}$$
$$=\frac{1+2i+i^2}{2}$$
$$=\frac{1+2i-1}{2}$$
$$=\frac{2i}{2}$$
$$=i$$

C and C++ complex square roots handle multiple possible results by making a "branch cut along the negative real axis." If we visualize the square root rotating a number around the complex plane, if it has multiple results it will favor the one that does not cross the negative real axis. Putting all of this together we can generate a list of first roots-of-unity like this:

```
std::vector<std::complex<double>> gen_omegas(long N, bool inverse) {
  int Nth = (bit_length(N) - 1); // log_2(N)
  std::vector<std::complex<double>> omegas;
  omegas.push_back(-1 + 0i);
  omegas.push_back(0  - 1i);
  for (int j = 2; j < Nth; j++) omegas.push_back(std::sqrt(omegas[j - 1]));
  if (inverse)
    for (int j = 1; j < Nth; j++) omegas[j] = std::conj(omegas[j]);
  return omegas;
}
```

bit_length(x) is the number of bits required to represent a number and that is the integer $log_2(x)+1$. This is much faster than using the floating-point log functions, most modern CPUs can do this in a single instruction. We can implement this in C++20, otherwise compiler intrinsics need to be used:

```
#include <bits>
#include <climits>
int bit_length(unsigned long){
  return (CHAR_BIT * sizeof N) - std::countl_zero(x);
}
```

12 The Fast Fourier Transform

Putting all of this together:

$$
\begin{aligned}
F(x) &= -1+x-3x^2+2x^3+4x^4+x^5 &&= F_e(x^2)+xF_o(x^2) \\
F_e(x) &= -1-3x+4x^2 &&= F_{ee}(x^2)+xF_{eo}(x^2) \\
F_o(x) &= 1+2x+x^2 &&= F_{oe}(x^2)+xF_{oo}(x^2) \\
F_{ee}(x) &= -1+4x & F_{eo}(x)=-3 \\
F_{oe}(x) &= 1+x & F_{oo}(x)=2
\end{aligned}
$$

Plugging in our roots of unity:

$$
\begin{array}{lll|lll|lll}
F_{ee}(1) &=3 & & F_e(1) &=0 & & F(1) &=4 \\
F_{ee}(-1) &=-5 & & F_e(i) &=5-3i & & F(\omega) &=-5-3i+2\omega i \\
F_{eo}(1) &=-3 & & F_e(-1) &=6 & & F(i) &=6 \\
F_{eo}(-1) &=-3 & & F_e(-i) &=-5+3i & & F(\omega i) &=-5+3i+2\omega \\
F_{oe}(1) &=1 & & F_o(1) &=2+2 & & F(-1) &=-4 \\
F_{oe}(-1) &=0 & & F_o(i) &=2i & & F(-\omega) &=-5-3i-2\omega i \\
F_{oo}(1) &=2 & & F_o(-1) &=0 & & F(-i) &=6 \\
F_{oo}(-1) &=2 & & F_o(-i) &=-2i & & F(-\omega i) &=-5+3i-2\omega
\end{array}
$$

Lets compare these answers to the results of the discrete Fourier transform for the same input set, we can pad in input set with 0s to get the discrete Fourier transform to output the same number of values, these are equivalent to coefficients of 0 for higher degree terms so it does not effect the value of the polynomial. We effectively were already doing this above.

Input	-1	1	-3	2	4	1	0	0
Output	4	-6.41-1.59i	6	-3.59+4.41i	-4	-3.59-4.41i	6	-6.41+1.59i

It is a perfect match, and with $O(NlogN)$ complexity! Lets write some code to calculate this, since the FFT can be calculated in-place X will be used as the input and overwritten:

```cpp
void fft(std::vector<std::complex<double>> omegas,
        std::vector<std::complex<double>> &X, long N) {
  assert((N & (N - 1)) == 0);
  for (long n = 2; n <= N; n *= 2) {
    std::complex<double> omega = omegas[bit_length(n) - 2];
    std::complex<double> root_of_unity = 1;
    for (long j = 0; j < n / 2; j++) {
      for (long k = 0; k < N; k += n) {
        std::complex<double> product = root_of_unity * X[k + j + n / 2];
        X[k + j + n / 2] = X[k + j] - product;
        X[k + j] = X[k + j] + product;
      }
      root_of_unity *= omega;
    }
  }
}
```

Lets run it and see what it does:

Input	-1	1	-3	2	4	1	0	0
Output	4	0.1-2.9i	1.0+5.0i	-4.1+7.1i	-6	-4.1-7.1i	1.0-5.0i	0.1+2.9i

This is not correct! This code is actually good, there is just one more thing we are missing.

$$x_0+\omega_2^0\cdot x_4 \qquad x_0+\omega_4^0\cdot x_2 \qquad x_0+\omega_8^0\cdot x_1$$

$x_0 \bullet \longrightarrow \bullet X_0$

$$x_0-\omega_2^0\cdot x_4 \qquad x_4+\omega_4^{-1}\cdot x_6 \qquad x_4+\omega_8^{-1}\cdot x_5$$

$x_4 \bullet \longrightarrow \bullet X_1$

$$x_2+\omega_2^0\cdot x_6 \qquad x_0-\omega_4^0\cdot x_2 \qquad x_2+\omega_8^{-2}\cdot x_3$$

$x_2 \bullet \longrightarrow \bullet X_2$

$$x_2-\omega_2^0\cdot x_6 \qquad x_4-\omega_4^{-1}\cdot x_6 \qquad x_6+\omega_8^{-3}\cdot x_7$$

$x_6 \bullet \longrightarrow \bullet X_3$

$$x_1+\omega_2^0\cdot x_5 \qquad x_1+\omega_4^0\cdot x_3 \qquad x_0-\omega_8^0\cdot x_1$$

$x_1 \bullet \longrightarrow \bullet X_4$

$$x_1-\omega_2^0\cdot x_5 \qquad x_5+\omega_4^{-1}\cdot x_7 \qquad x_4-\omega_8^{-1}\cdot x_5$$

$x_5 \bullet \longrightarrow \bullet X_5$

$$x_3+\omega_2^0\cdot x_7 \qquad x_1-\omega_4^0\cdot x_3 \qquad x_2-\omega_8^{-2}\cdot x_3$$

$x_3 \bullet \longrightarrow \bullet X_6$

$$x_3-\omega_2^0\cdot x_7 \qquad x_5-\omega_4^{-1}\cdot x_7 \qquad x_6-\omega_8^{-3}\cdot x_7$$

$x_7 \bullet \longrightarrow \bullet X_7$
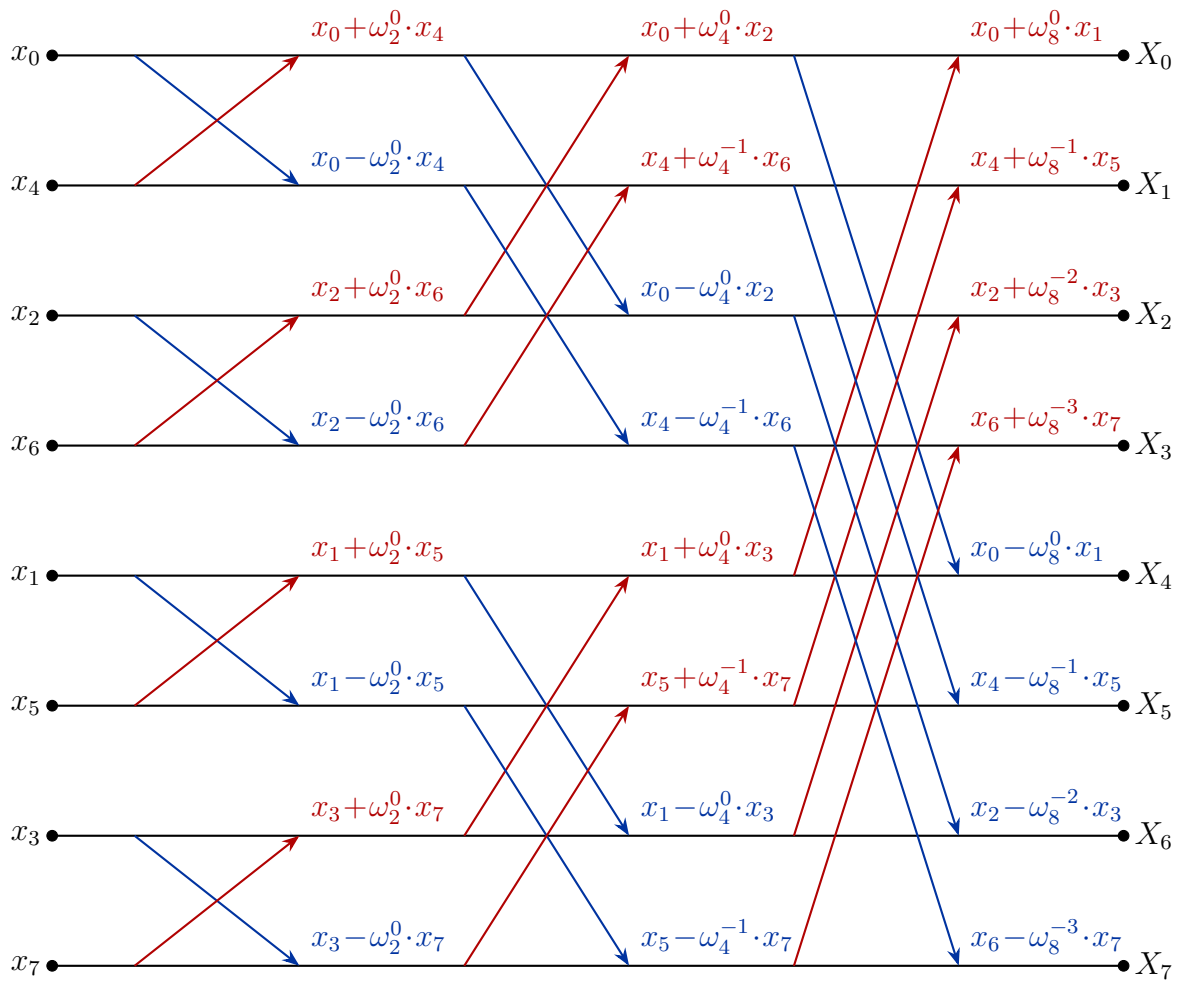
The Cooley-Tukey Radix-2 Decimation-In-Frequency Fast Fourier Transform

If we actually draw out how values are reused we get something like the above, usually called a butterfly diagram. Each of the sets of arrows are sometimes called a "butterfly operation." The part we are missing is the input ordering, we need to get the inputs in their order after the recursive even/odd separating. Alternatively we could do the operations backwards, start with the inputs in-order and do the size N butterfly operation then the size $\frac{N}{2}$ butterfly operation and so on, but then the output would be in this peculiar ordering anyway and we would have to get the first 8th root of unity from the start instead of being able to take the square root each pass. This ordering is actually a little insidious, take a second to see if you can figure out a non-recursive way to describe the ordering.

The ordering is called bit-reversal-permutation, the index of each element in this ordering is the bits of the original index reversed. With the input in this order we, at last, get:

Input	-1	1	-3	2	4	1	0	0
Output	4	-6.41-1.59i	6	-3.59+4.41i	-4	-3.59-4.41i	6	-6.41+1.59i