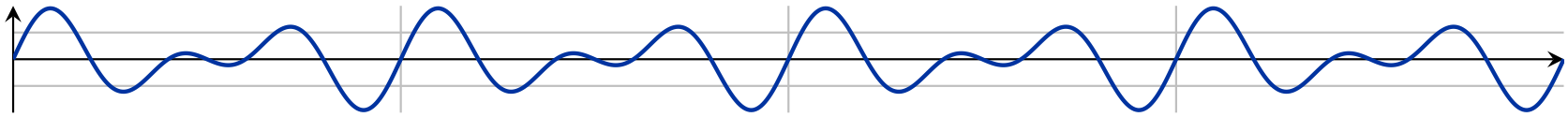
The Fast Fourier Transform

Zachary Todd Edwards

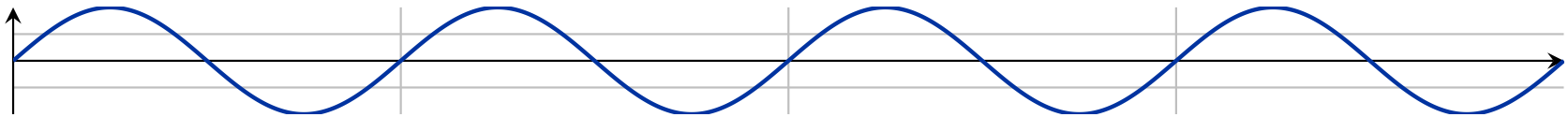
July 19, 2023

A Hard Problem

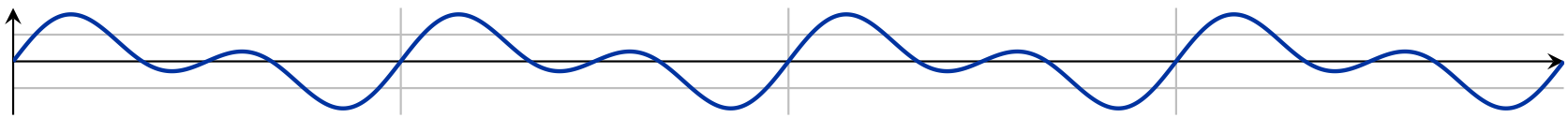
Lets say we have a wave that we want to find the component frequencies of:



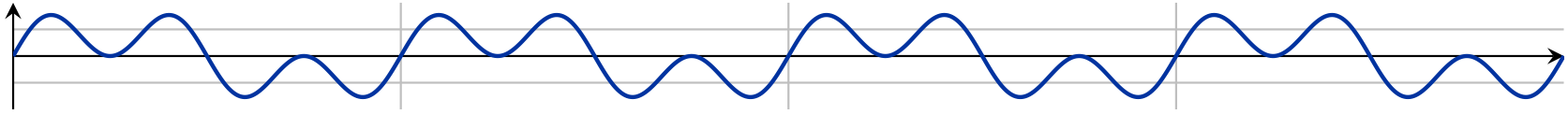
We could try a brute force approach, look at the graphs of sums of sine waves until we get our original:



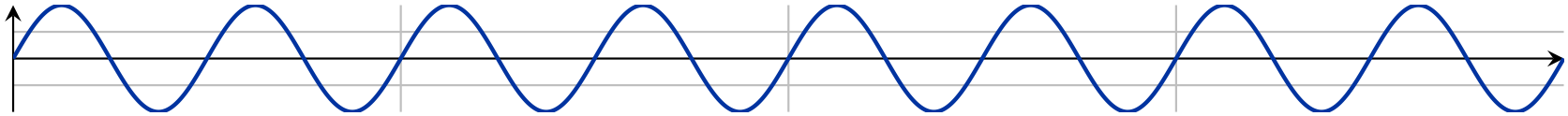
$1\text{Hz} + 1\text{Hz}$



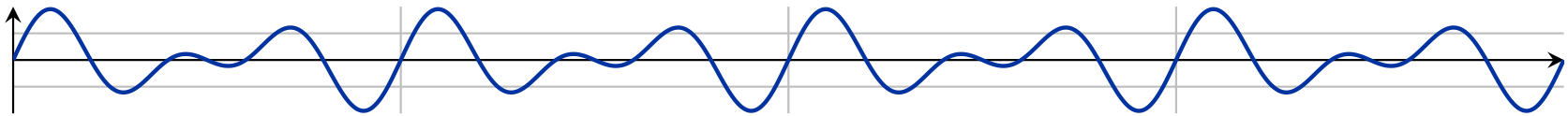
$1\text{Hz} + 2\text{Hz}$



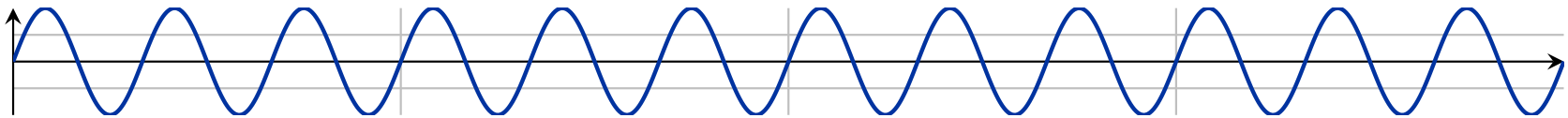
1Hz + 3Hz



2Hz + 2Hz



2Hz + 3Hz



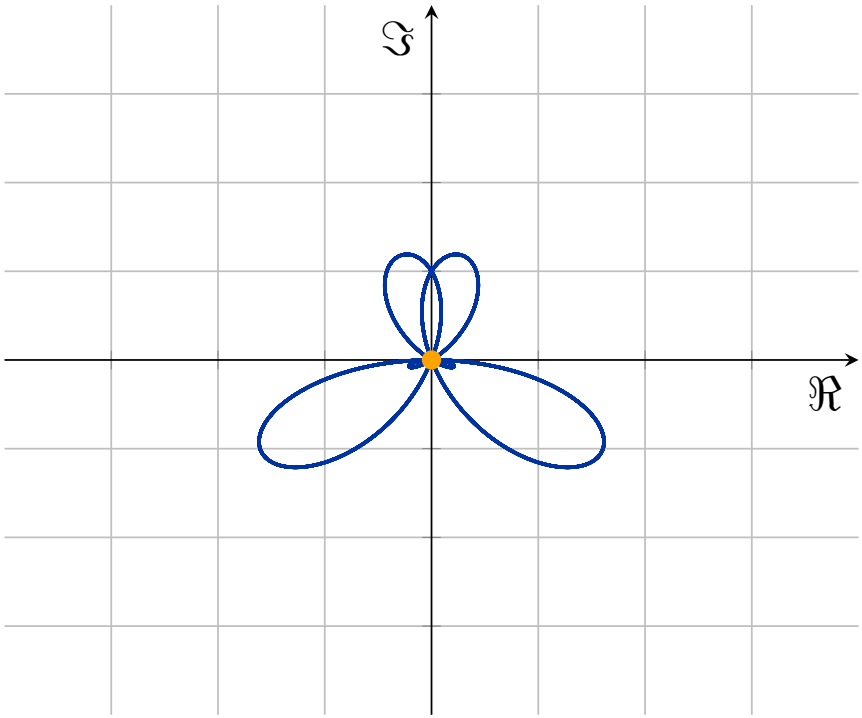
3Hz + 3Hz

Assumptions made:

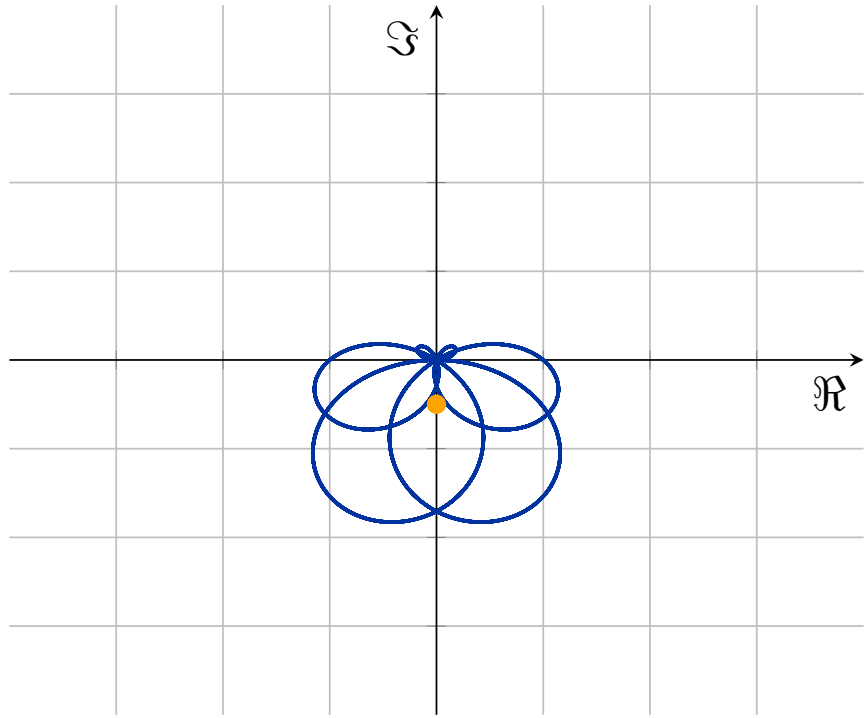
- Amplitude centered at zero.
- Phase centered, not shifted left or right.
- All component frequencies were integer.
- Input was perfect, no noise.

Brute force was already not practical, these make it even worse!

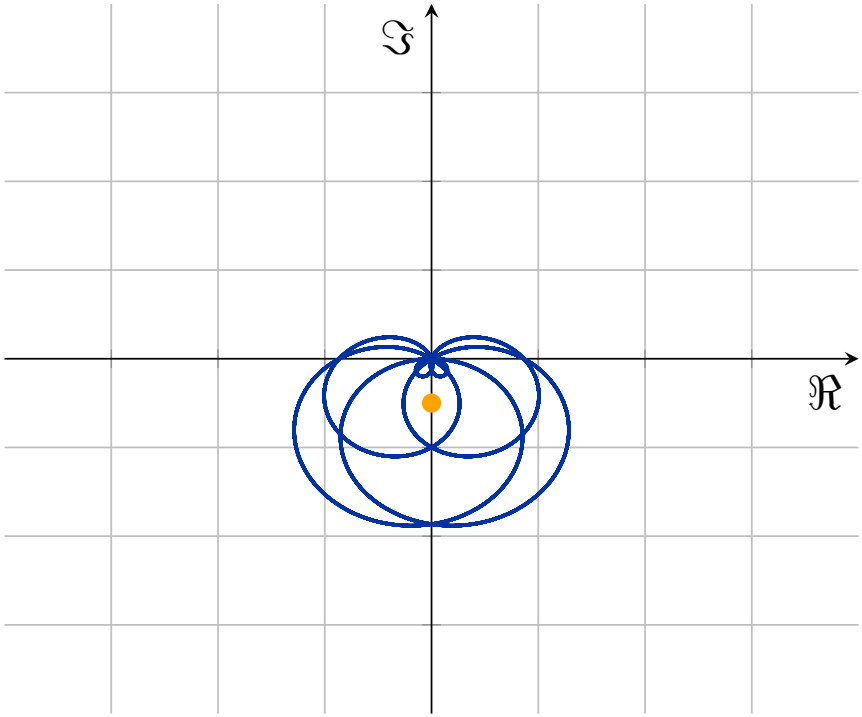
Lets try visualizing it in a different way by wrapping our waveform around the origin of a graph:



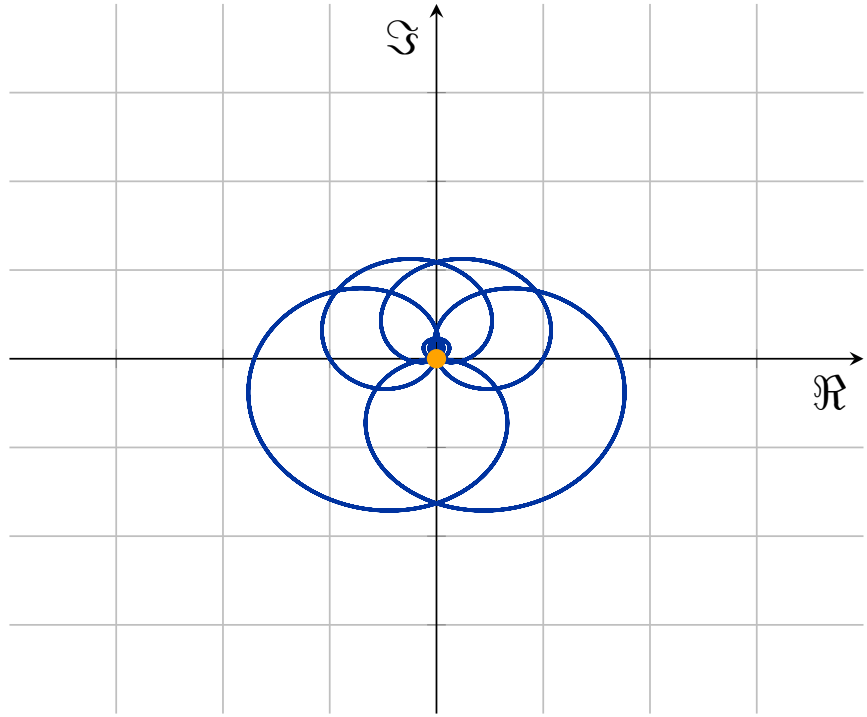
Rotation Period = 1



Rotation Period = 2



Rotation Period = 3



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!

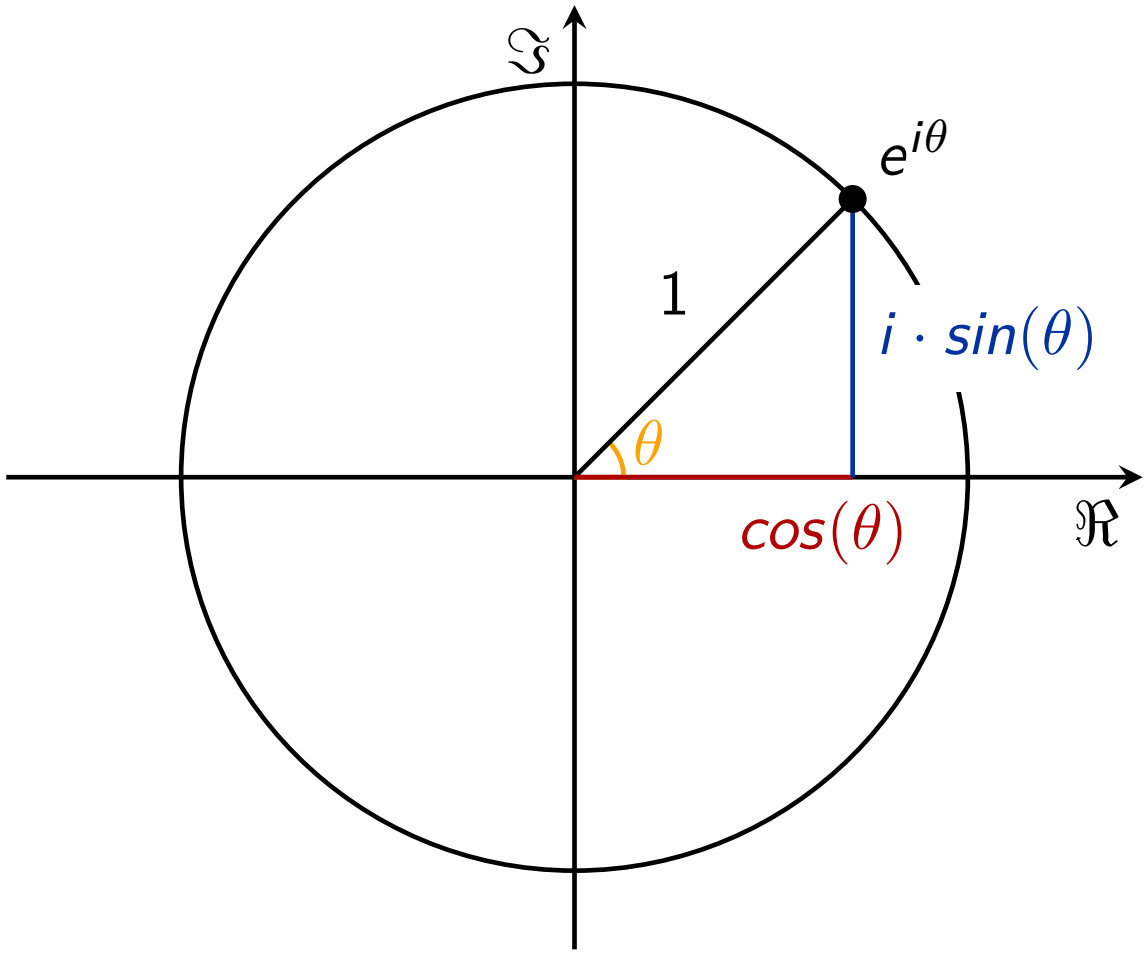
Euler's Formula

How do we mathematically express this, is it actually any easier?

- Its not any simpler visually.
- Getting the average is an extra step.
- Rotation period seems vague and complicated.

The axis labels have already given away part of the secret, these wrappings are being graphed on the complex plane!

Euler's Formula



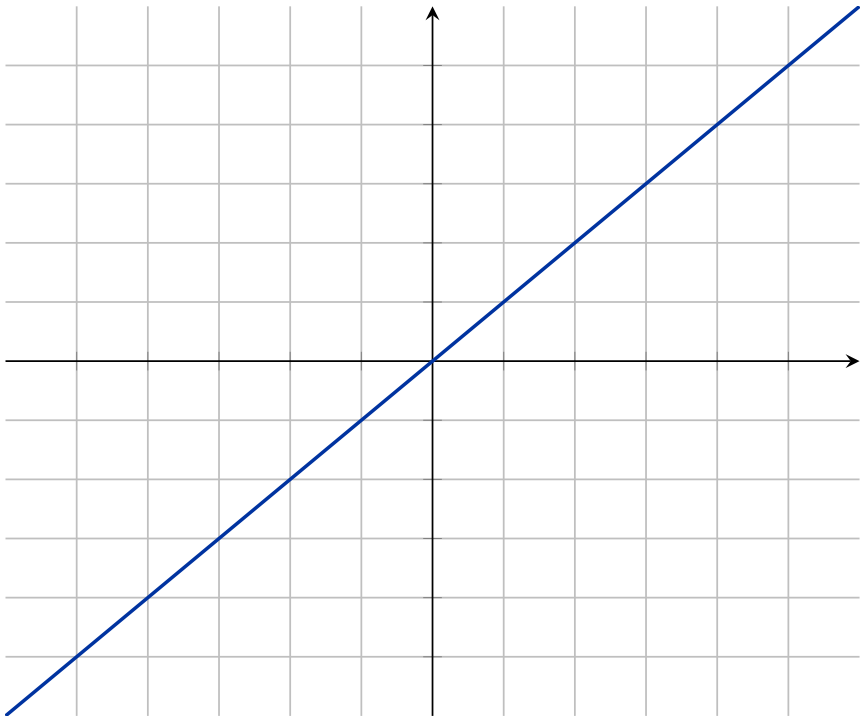
The Complex Unit Circle

$$e^{i\theta} = cos(\theta) + i \cdot sin(\theta)$$

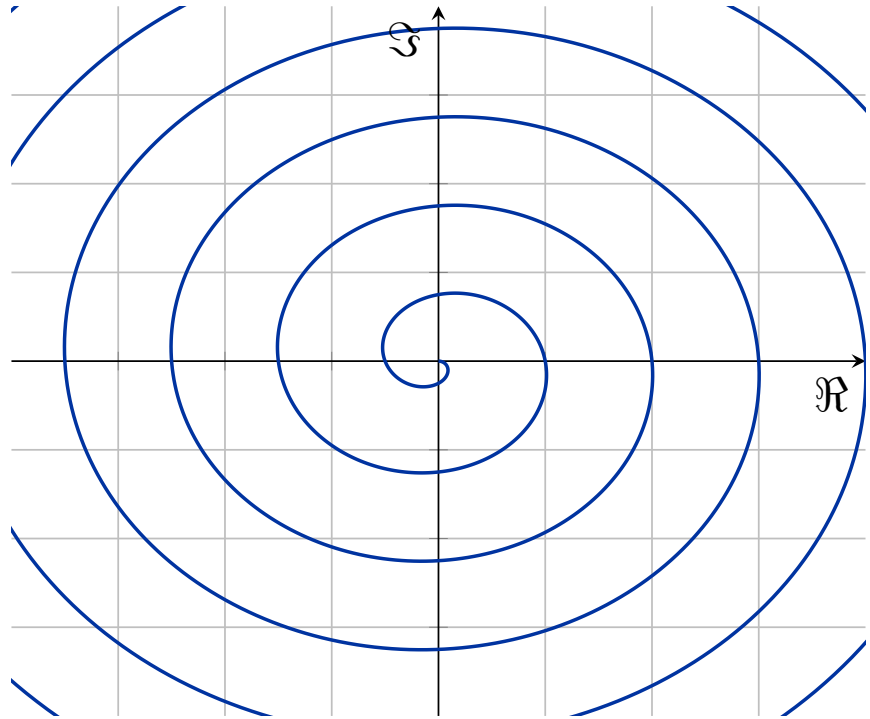
If e^x is growth that is equal to its current value, and i is a right angle on the complex plane, e^{ix} is rotation equal to its current value. This is a circle with a radius of 1. It is also worth noting that this is continuous and works for angles larger than 2π or less than 0.

Spiralling a Function

Since Euler's formula always has a distance from the origin of 1 rotated by θ radians and multiplying anything by 1 is itself we can plug in our variable times 2π for θ and then multiply our function with that to wrap it around the origin!

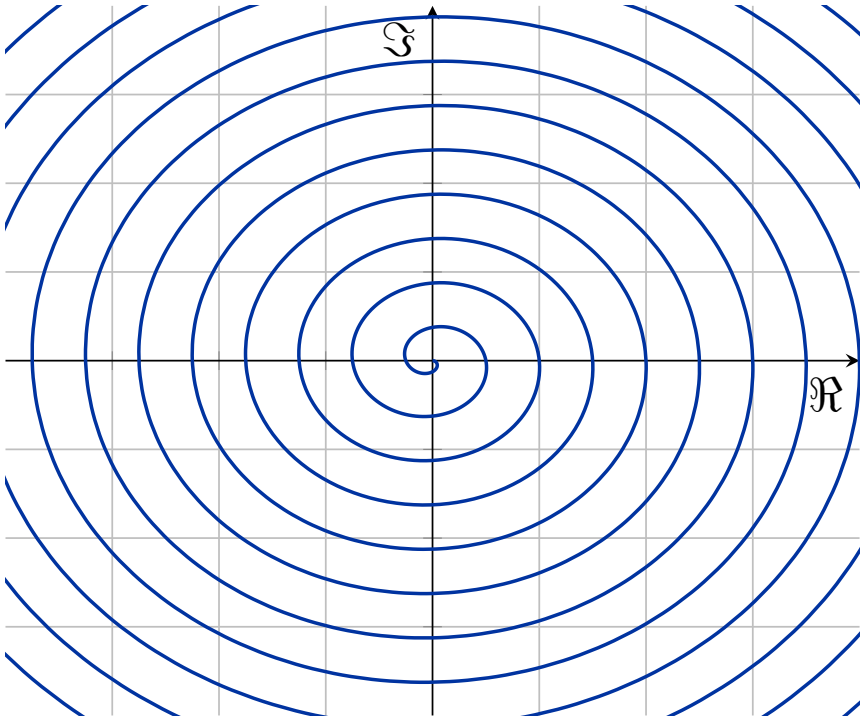


$F(x) = x$

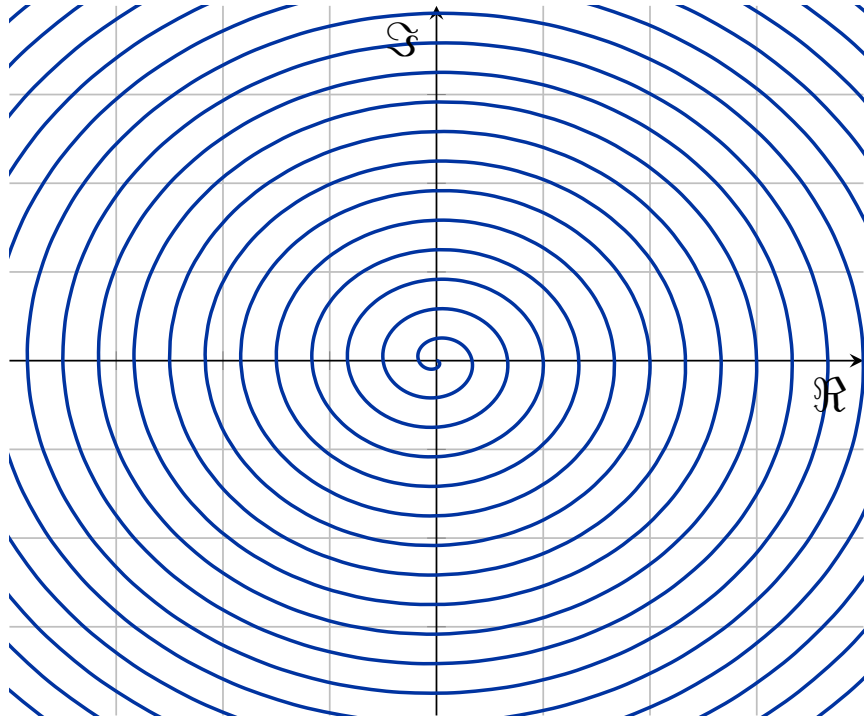


$F(x) \cdot e^{ix2\pi}$ where $x \geq 0$

The rotation period is how many full rotations we do for one unit increase to the input of our original function. We can adjust this is by multiplying 2π in the original formula. For a period of two we multiply it by two, etc. We can multiply by fractions to get periods under zero.



$F(x) \cdot e^{ix2 \cdot 2\pi}$ where $x \geq 0$



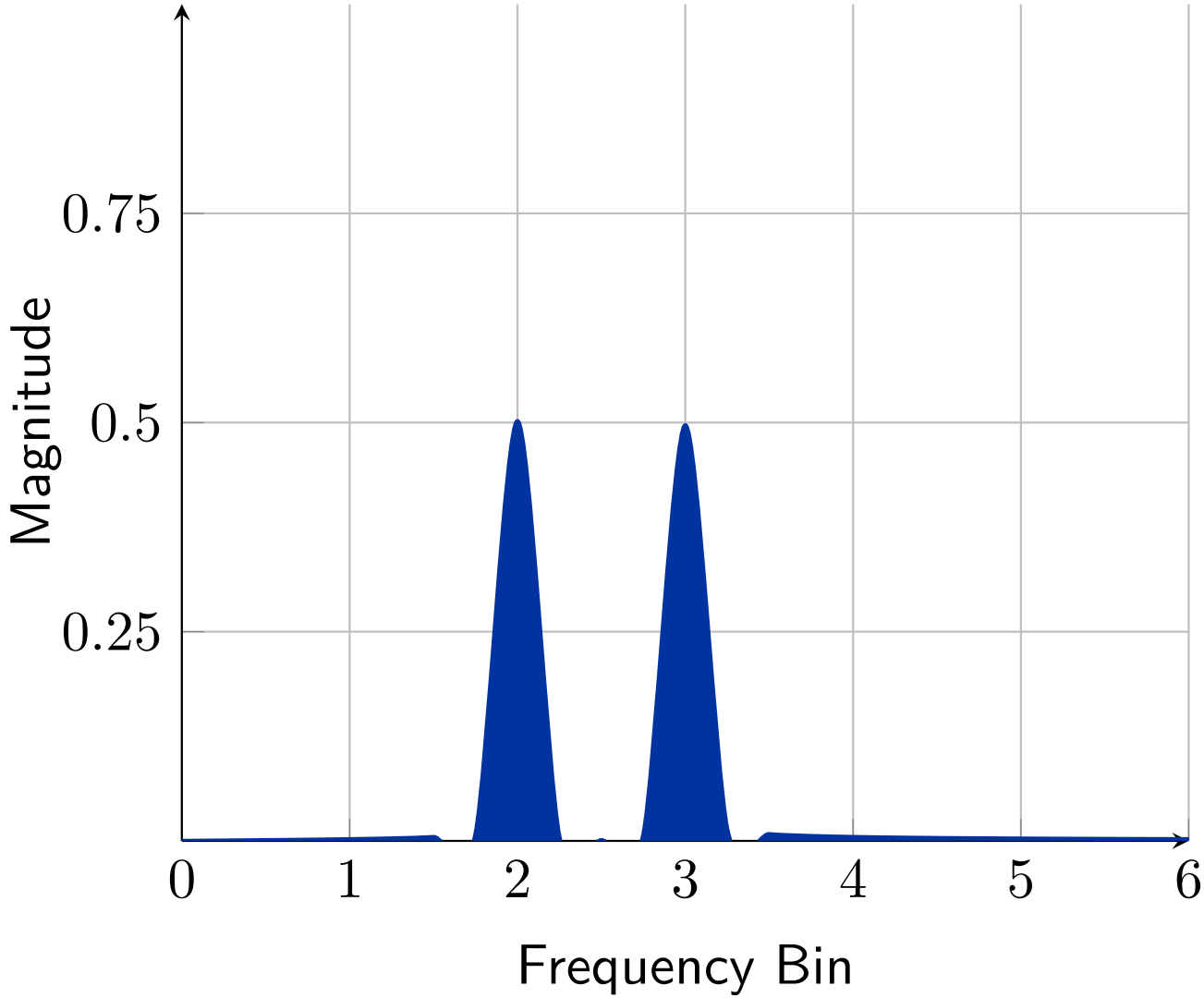
$F(x) \cdot e^{ix3 \cdot 2\pi}$ where $x \geq 0$

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:

- No average, with the average it is sometimes called the "normalized Fourier transform."
- No time frame, runs from $-\infty$ to $+\infty$.
- Winding is done in clockwise direction, negative exponent.
- Variable being integrated is usually t.

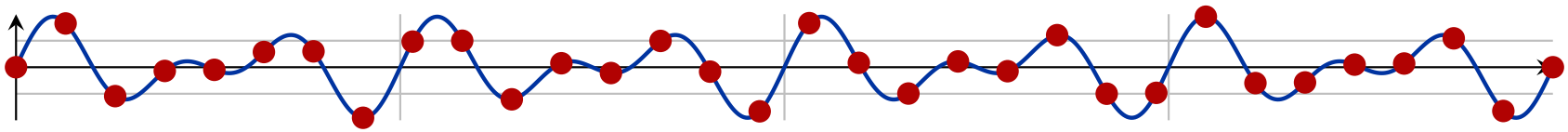
With all this:

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

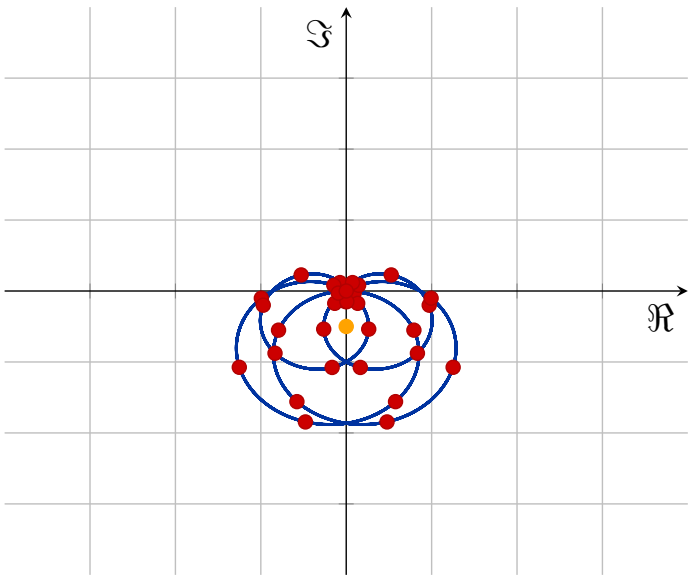
There is just one problem: computers tend to not like continuity or infinity.

The Discrete Fourier Transform

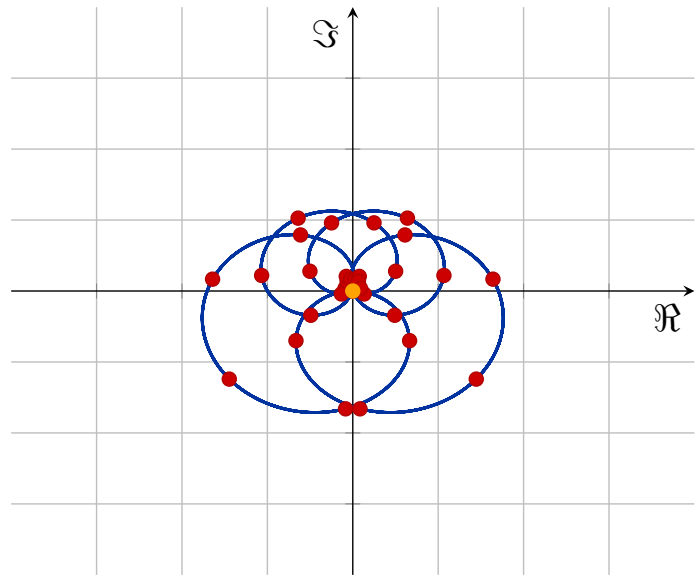
Lets try taking discrete points on the waveform and sum up their results when multiplied with $e^{i\theta}$ at regular intervals.



32 samples across 4 seconds, for a sample rate of 8 per second.



Rotation Period = 3



Rotation Period = 4

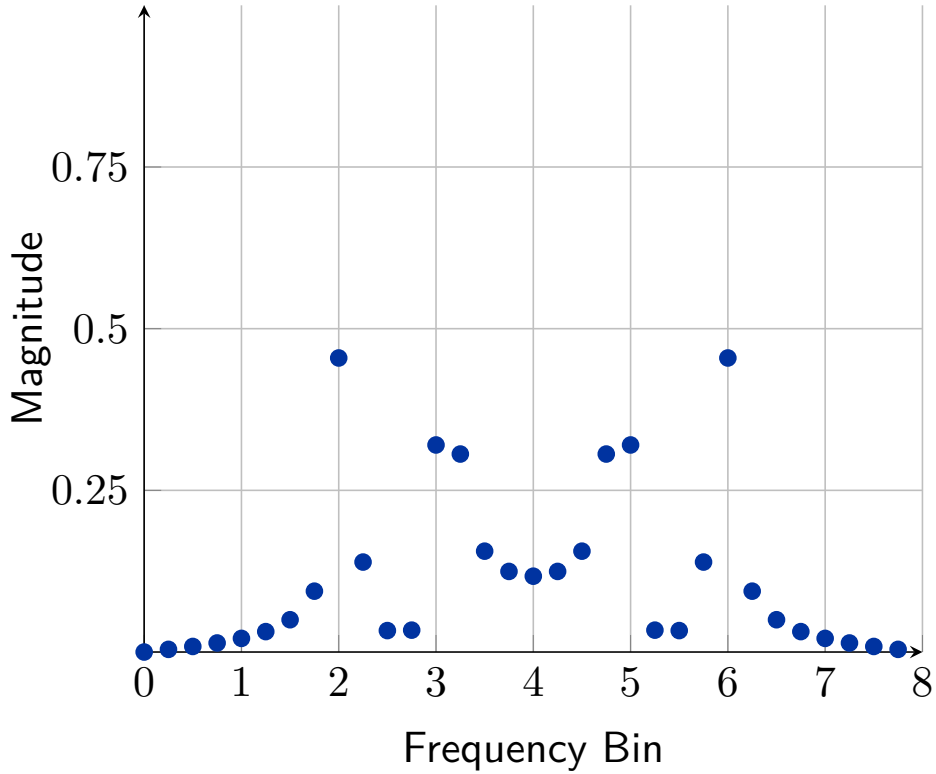
The discrete version of the Fourier transform should be relatively easy to reverse engineer:

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

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:

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

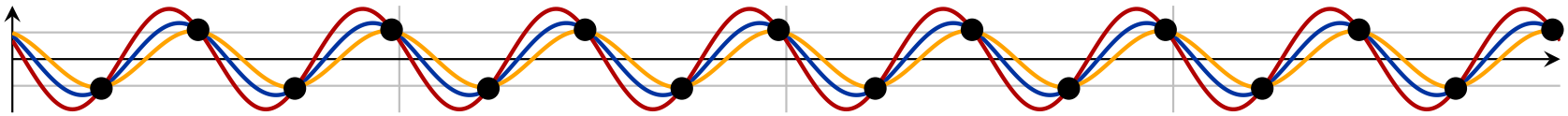
With all that we get the following for our graph:



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?

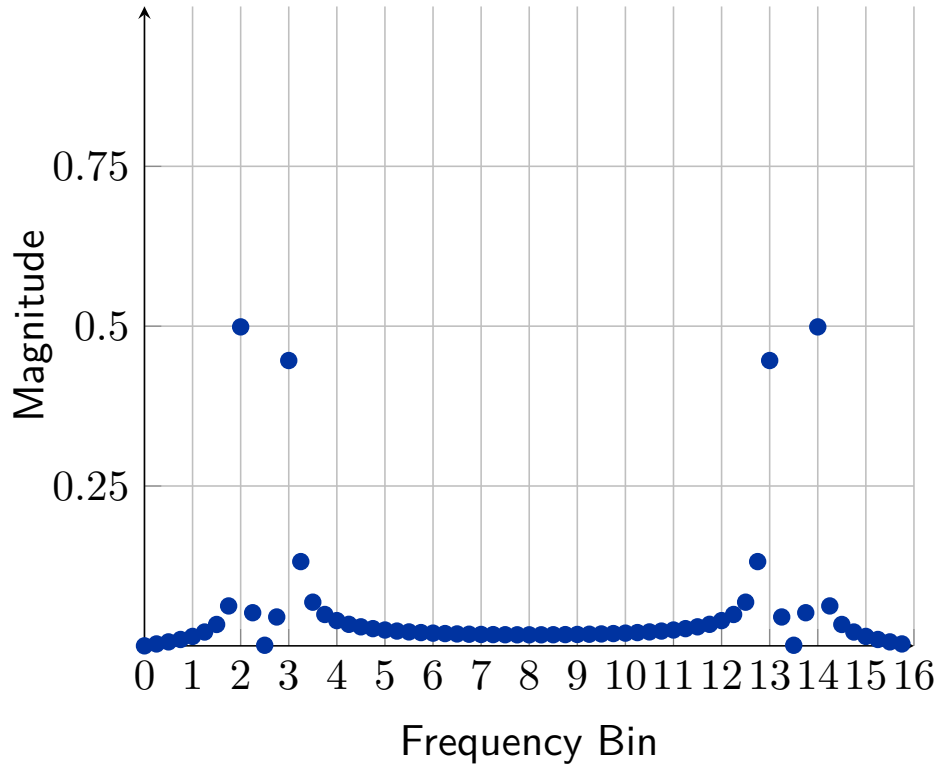
Nyquist-Shannon

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, these samples also have many different waves that could fit them, there are actually infinitely many waveforms that could fit any set of discrete points! We need 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, half of our sample rate, non-inclusive.

Interpreting the results:

- $k\frac{f_s}{N}$ is the frequency of each index 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.
- The imaginary part of each entry tells us that frequency's phase, how much it is shifted left or right.

Complex Programming

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:

```
1  #include <complex.h>
2  void dft(double complex *x, double complex *X, int N){
3    for(int i = 0; i < N; i++)
4      X[i] = CMPLX(0.0, 0.0);
5
6    for(int k = 0; k < N; k++)
7      for(int n = 0; n < N; n++)
8        X[k] += x[n] * cexp(-(((I * 2 * M_PI) / N) * k * n));
9  }
```

C++ is similar but uses templates:

```
1     #include <vector>
2     #include <complex>
3     void dft(std::vector<std::complex<double>> &x,
4              std::vector<std::complex<double>> &X, int N){
5       X.clear();
6       X.resize(N, 0);
7
8       for(int k = 0; k < N; k++)
9         for(int n = 0; n < N; n++)
10          X[k] += x[n] * std::exp(-(((1.0i * 2 * M_PI) / N) * k *
11    }
```

Perl:

```
1   use Math::Complex;
2   sub DFT{
3     my ($x, $X) = @_;
4     my $N = scalar(@$x);
5     @$X = ();
6
7     for(my $k = 0; $k < $N; $k++){
8       for(my $n = 0; $n < $N; $n++){
9         $$X[$k] += $$x[$n] * exp(-(((i * 2 * pi) / $N)
10                                  * $k * $n));
11      }
12    }
13  }
```

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.

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:

```
class point{
  double x, y;
  point(_x, _y) : x(_x), y(_y){}
};
```

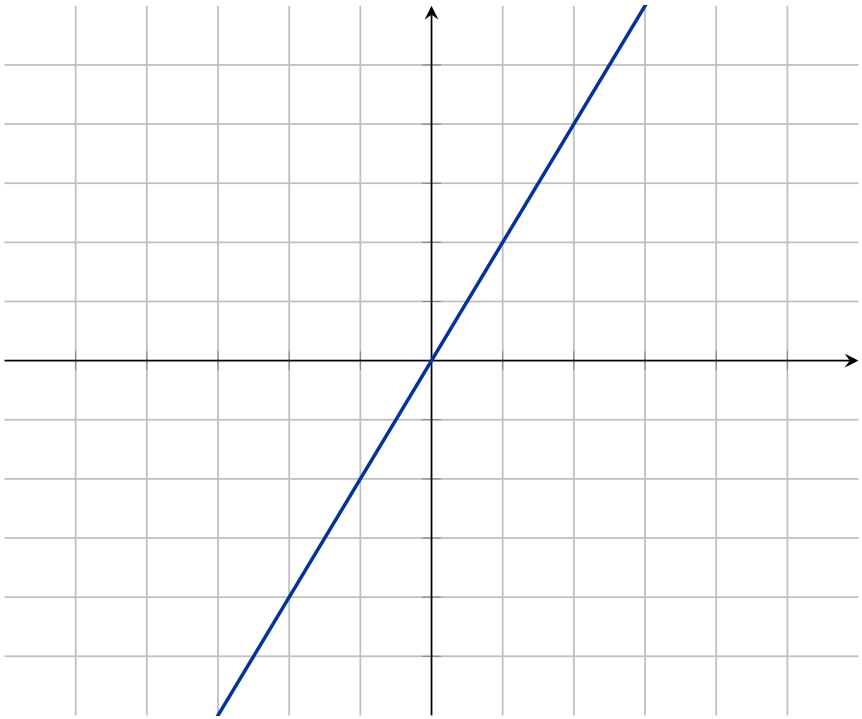
We might try to use the brute force method:

```
vector<point> polynomial(int degree,
      double coefficients[], int n){
  vector<double> points(n);
  for(int x = 0; x < n; x++){
    double y = coefficients[0];
    for(int j = 1; j <= degree; j++)
      y += coefficients[j] * pow(x,j);
    points[x] = point(x,y);
  }
  return samples;
}
```

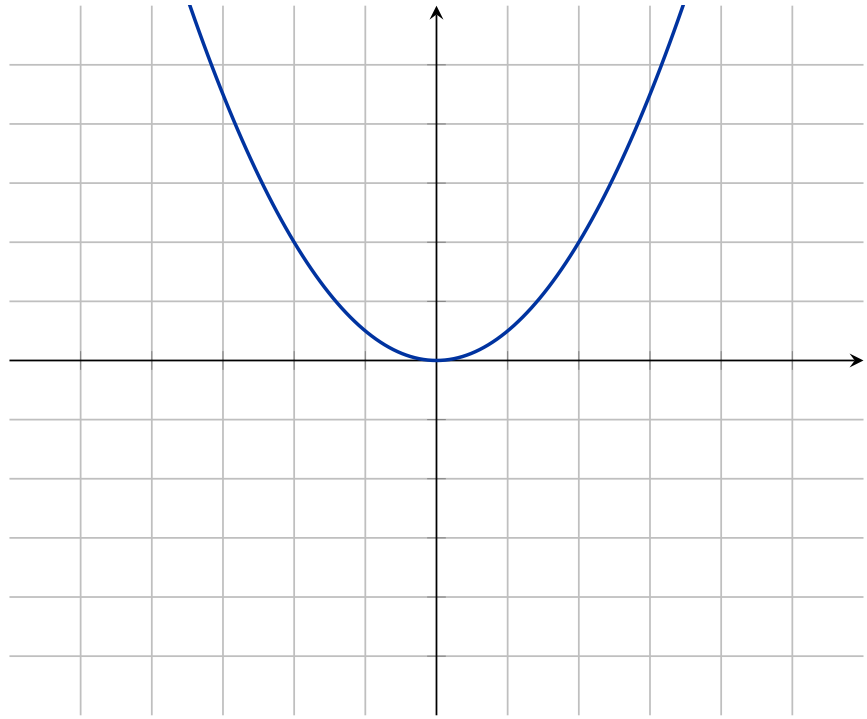
This takes $O(nk)$ time where n is the degree of the polynomial and k is the number of samples needed. Is there a faster way?

Symmetry

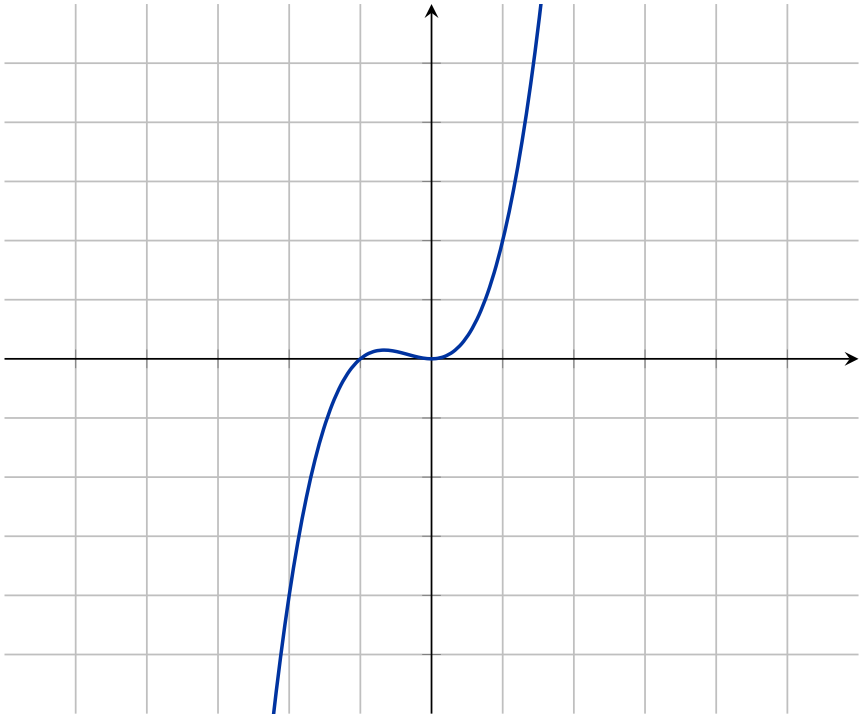
Lets look at a few graphs of polynomials:



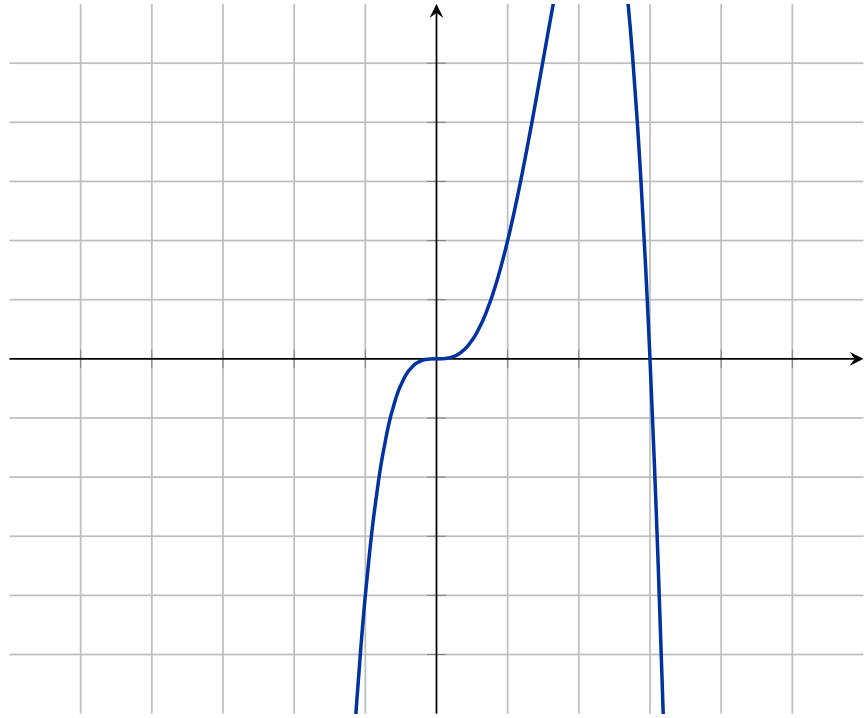
$F(x) = 2x$



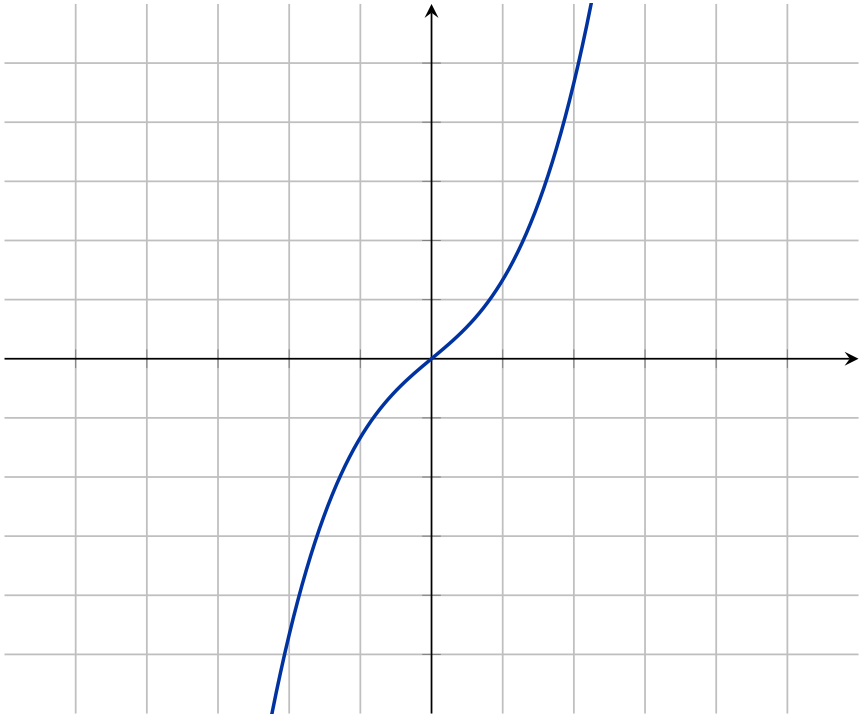
$F(x) = \frac{1}{2}x^2$



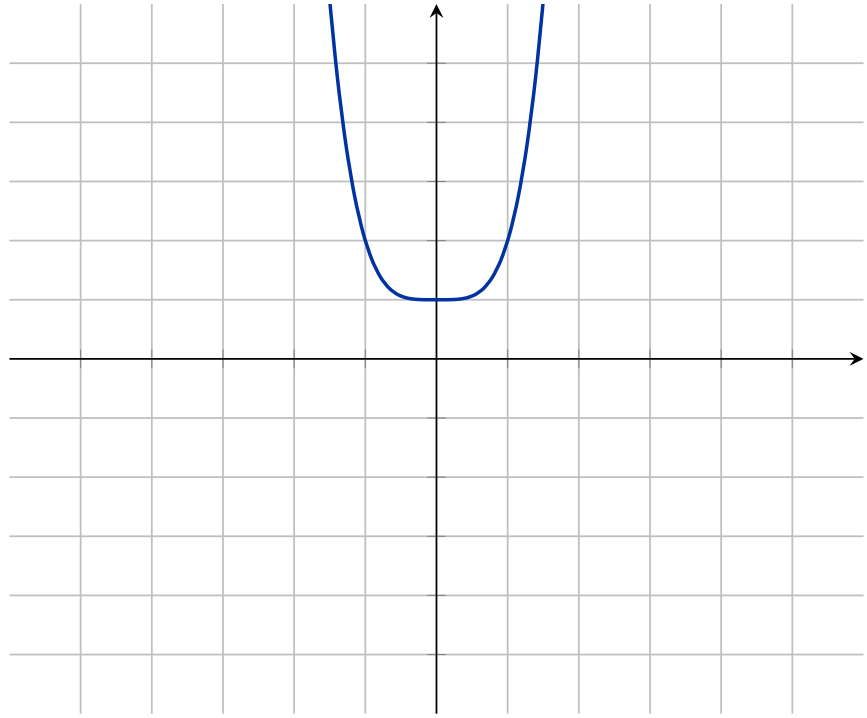
$F(x) = x^2 + x^3$



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



$F(x) = x + \frac{1}{3}x^3$



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

Observations:

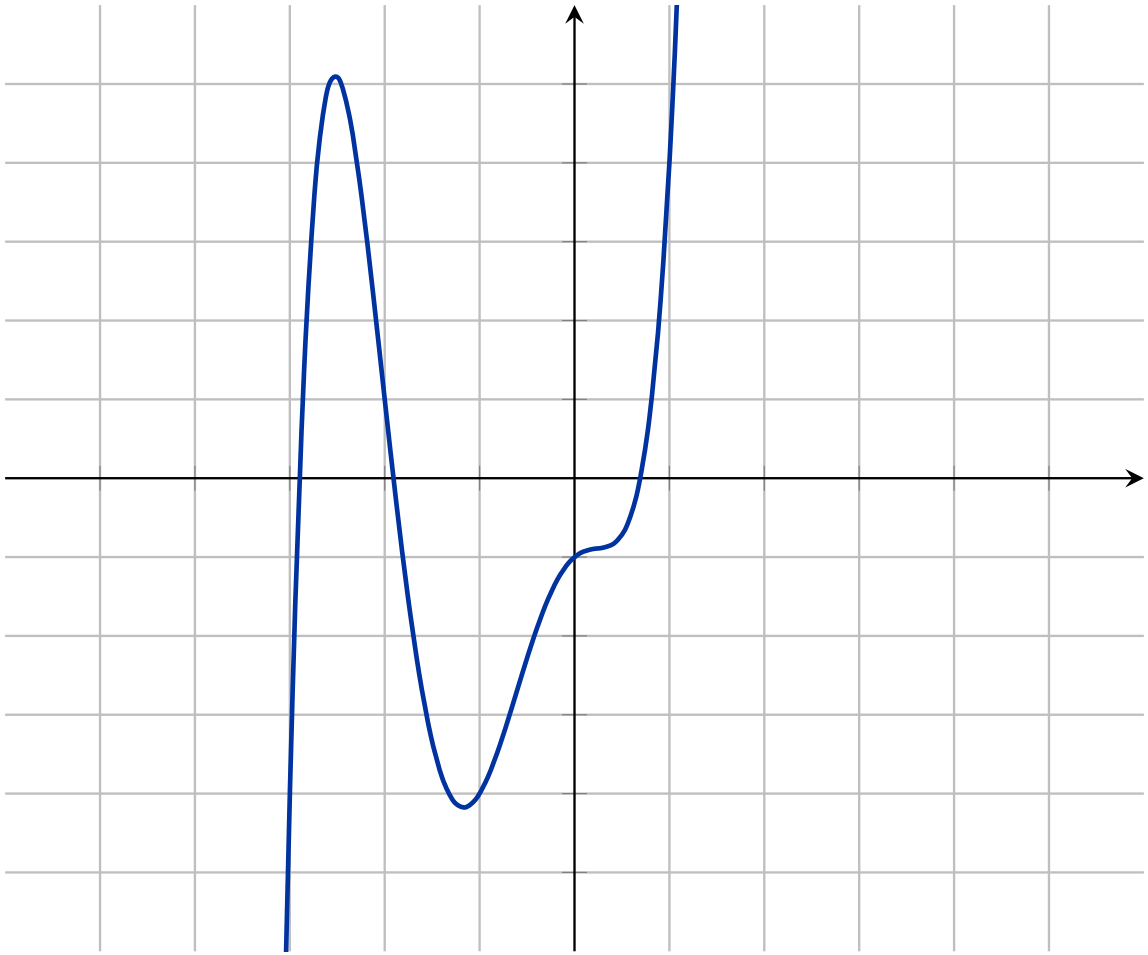
- Some are symmetrical across the vertical axis.
- Some are symmetrical across both axes.
- Some are not symmetrical across either axis.

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: $F(x) = -F(-x)$.

How can we take advantage of this?

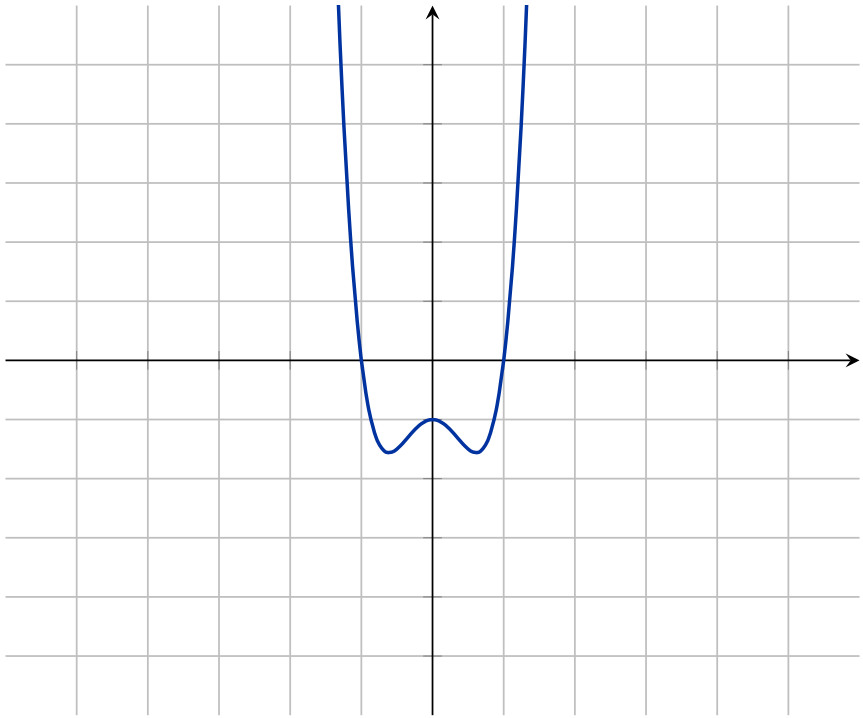
Polynomial Decomposition

Lets take the following rather wooly polynomial:

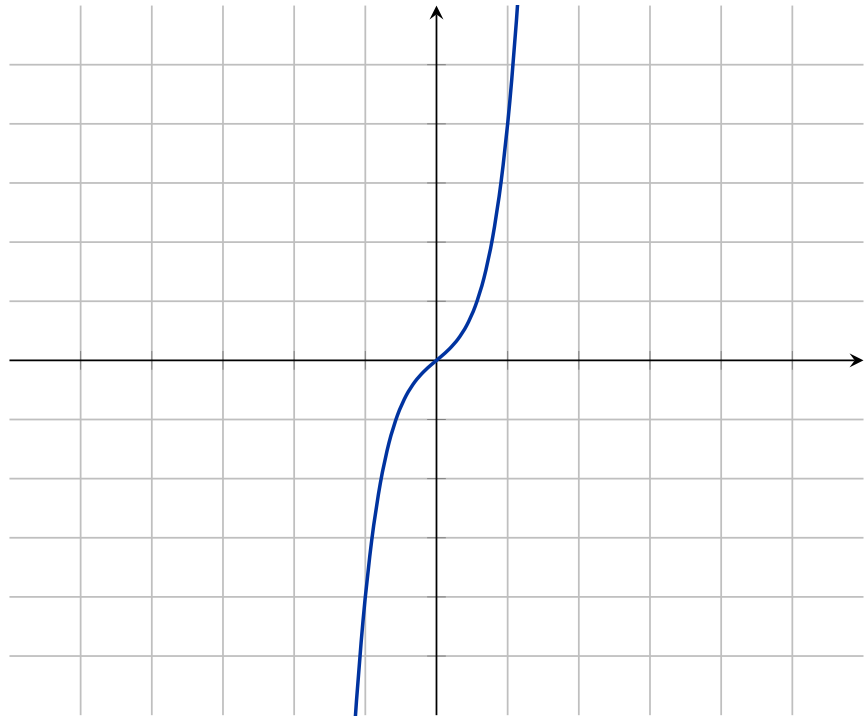


$$F(x) = -1 + x - 3x^2 + 2x^3 + 4x^4 + x^5$$

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:

```
vector<point> points;
assert(!(n % 2));
for(int x = 1; x <= n / 2; x++){
  double yp = coefficients[0];
  double yn = coefficients[0];
  for(int j = 1; j <= degree; j++){
    double term = pow(x,j);
    yp += term;
    yn += (j % 2) ? -term : term;
  }
  points.push_back(point(x,yp));
  points.push_back(point(-x,yn));
}
return points;
```

Gross!

- Inelegant!
- Asymptotically the same.

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.

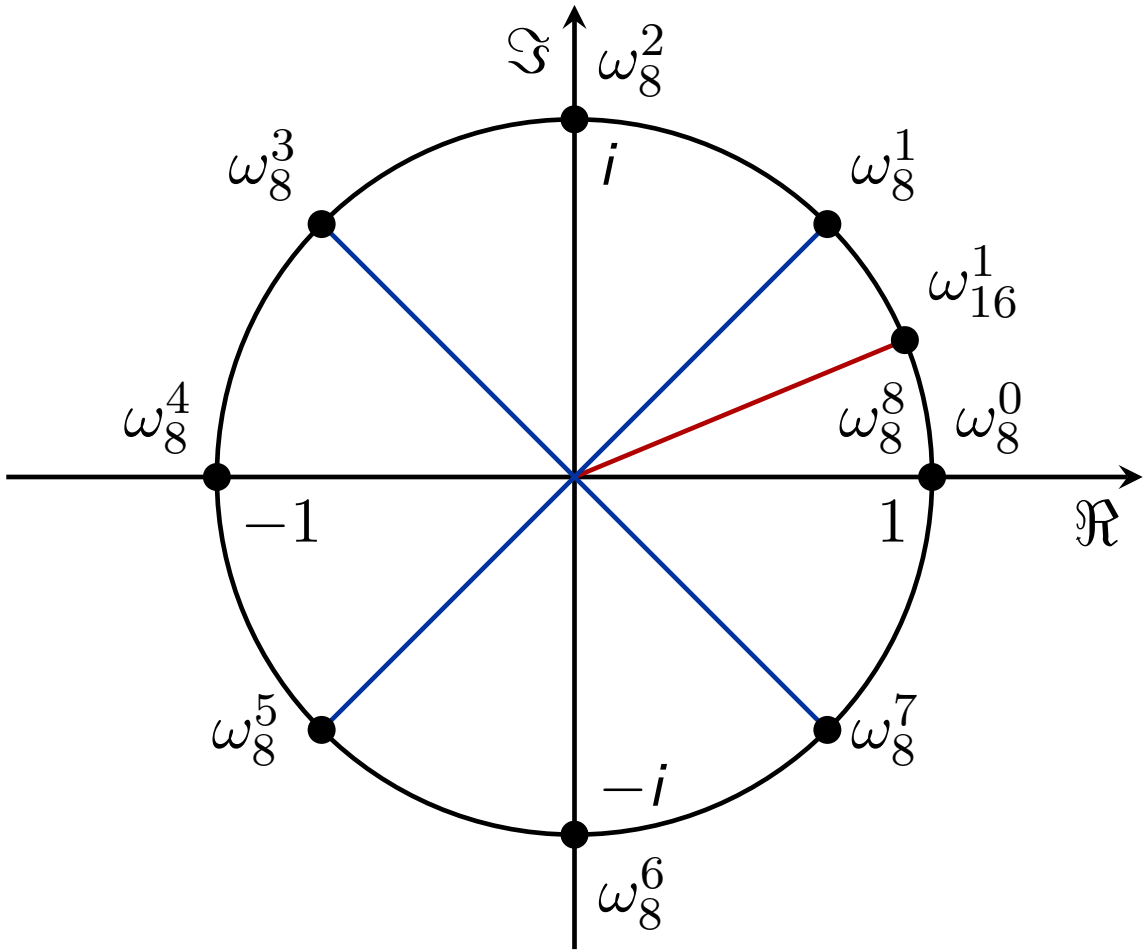
This could possibly eliminate the ternary operator above which might benefit performance a bit, but we are still stuck with the same complexity. To get any noticeable benefit we need a way to do this recursively.

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 $F(x) = F_e(x^2) + xF_o(x^2)$.

This does not work recursively! Squaring the input makes it positive and changes its magnitude, this only works with inputs of 0 and 1 for real numbers.

We used the complex exponential e^{ix} as a kind of "1" before, would it work here?

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, same for complex numbers.
- For real numbers $\sqrt[3]{1}$ has only one answer, 1, complex numbers have 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).
- 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; e.g., $\omega_8^2 = \omega_4^1$.
- Negation "flips" their position; e.g., $-\omega_8^1 = \omega_8^5$.
- They are cyclical when raised to a power or multiplied; e.g., $\omega_8^{3^{10}} = \omega_8^3 \cdot \omega_8^{10} = \omega_8^{30} = \omega_8^6$.

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:

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

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

The absolute value of a complex number is $\sqrt{a^2 + b^2}$ and squaring a complex number is the same as real algebra, replacing i^2 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:

$$\begin{aligned}\sqrt{1} &= \sqrt{\frac{|1|+1}{2}} + i\sqrt{\frac{|1|-1}{2}} \\ &= \sqrt{\frac{2}{2}} + i\sqrt{\frac{2}{2}} \\ &= \sqrt{1} + i\sqrt{0} \\ &= -1 + 0i\end{aligned}$$

$$\begin{aligned}(-1+0i)^2 &= (-1)^2 - (0)^2 \\ &\quad + 2(-1)(0)i \\ &= 1 - 0 + 0i \\ &= 1\end{aligned}$$

$$\begin{aligned}\sqrt{-1} &= \sqrt{\frac{|-1|+-1}{2}} + i\sqrt{\frac{|-1|--1}{2}} \\ &= \sqrt{\frac{0}{2}} + i\sqrt{\frac{2}{2}} \\ &= \sqrt{0} + i\sqrt{1} \\ &= 0 + 1i = i\end{aligned}$$

$$\begin{aligned}i^2 = (0+1i)^2 &= (0)^2 - (1)^2 \\ &\quad + 2(0)(1)i \\ &= 0 - 1 + 0i \\ &= -1\end{aligned}$$

$$
\begin{aligned}
\sqrt{i} = \sqrt{0+1i} &= \sqrt{\frac{|0+1i|+0}{2}} \\
&\quad + i\sqrt{\frac{|0+1i|-0}{2}} \\
&= \sqrt{\frac{1}{2}} + i\sqrt{\frac{1}{2}} \\
&= \frac{1}{\sqrt{2}} + \frac{i}{\sqrt{2}} \\
&= \frac{1+i}{\sqrt{2}}
\end{aligned}
$$

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

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:

```
1  std::vector<std::complex<double>> gen_omegas(long N,
2                                               bool inverse) {
3    int Nth = (bit_length(N) - 1); // log_2(N)
4    std::vector<std::complex<double>> omegas;
5    omegas.push_back(-1 + 0i);
6    omegas.push_back(0  - 1i);
7    for (int j = 2; j < Nth; j++)
8      omegas.push_back(std::sqrt(omegas[j - 1]));
9    if (inverse)
10     for (int j = 1; j < Nth; j++)
11       omegas[j] = std::conj(omegas[j]);
12   return omegas;
13 }
```

`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);
}
```

The Fast Fourier Transform

Putting all of this together:

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

Plugging in our roots of unity:

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

Comparing the results with that of the discrete Fourier transform on the coefficients, padding with zeros to get the same number of outputs:

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:

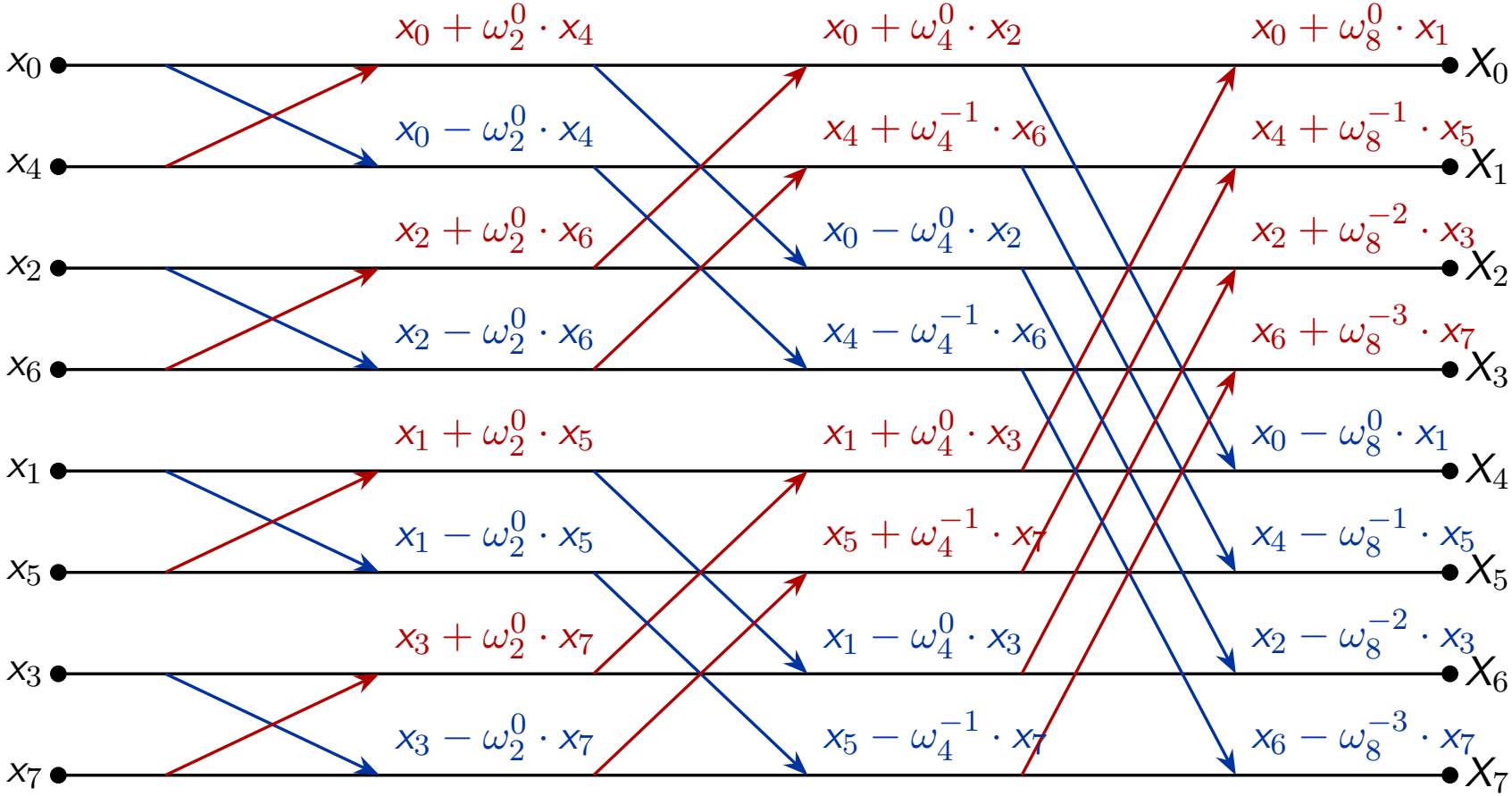
```
1   void fft(std::vector<std::complex<double>> omegas,
2            std::vector<std::complex<double>> &X, long N) {
3     assert((N & (N - 1)) == 0);
4     for (long n = 2; n <= N; n *= 2) {
5       std::complex<double> omega = omegas[bit_length(n) - 2];
6       std::complex<double> root_of_unity = 1;
7       for (long j = 0; j < n / 2; j++) {
8         for (long k = 0; k < N; k += n) {
9           std::complex<double> product =
10            root_of_unity * X[k + j + n / 2];
11          X[k + j + n / 2] = X[k + j] - product;
12          X[k + j] = X[k + j] + product;
13        }
14        root_of_unity *= omega;
15      }
16    }
17  }
```

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.

The Cooley-Tukey Radix-2 Decimation-In-Frequency Fast Fourier Transform



Drawing out how values are reused we get something like the above, usually called a butterfly diagram. The part we are missing is the input ordering, we need to get the inputs in their order after the recursive even/odd separating.

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