# Lecture 17: Spectral Analysis

**Steven Skiena**

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.sunysb.edu/∼skiena

## Spectral Analysis

Certain phenomena of financial (and other) time series data is best revealed in the *frequency domain*, or equivalently represented by their *spectra*.

A *duality transform* is a one-to-one mathematical function that takes a mathematical object of type-1 and maps it to an equivalent type-2 mathematical object.

Sample duality relations are point-line duality in computational geometry, and Laplace transforms used solving differential equations.

Such transforms are useful if there are interesting algorithms and tools for manipulating data of type 2.

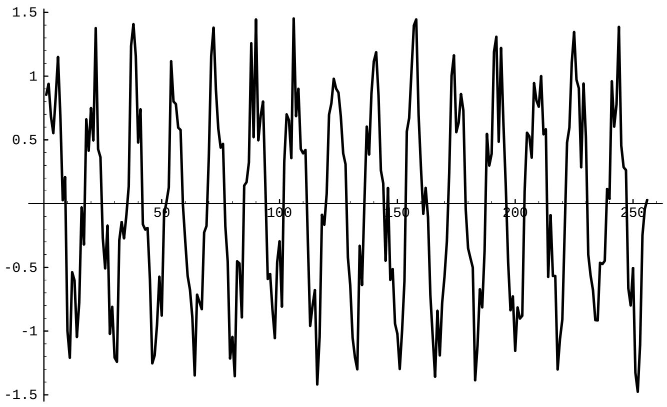
## Fourier Transforms

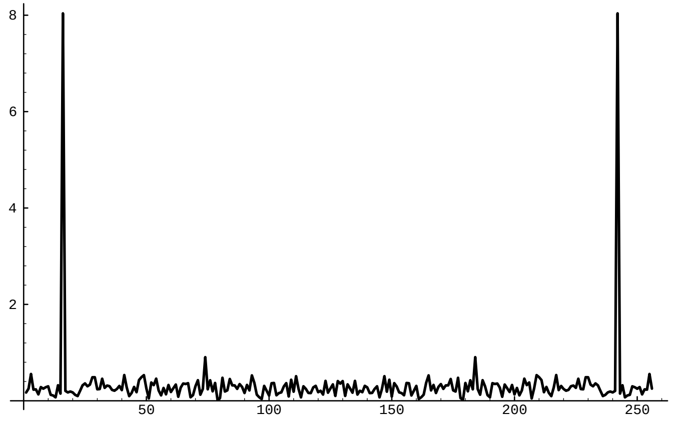
Perhaps the most useful duality transform known is the *Fourier transform* for representing time-series data as the sums of sine and cosine functions.

Its wide applicability is due to the existence of *Fast Fourier Transform* algorithm or *FFT* which computes what seems like an inherently quadratic function in $O(n \lg n)$ time.

## Filtering via the FFT

On the left, we construct a time series of points sampled from a sine function with added random noise.
On the right we take the Fourier transform of this series, plotting the coefficients of the resulting sine functions:

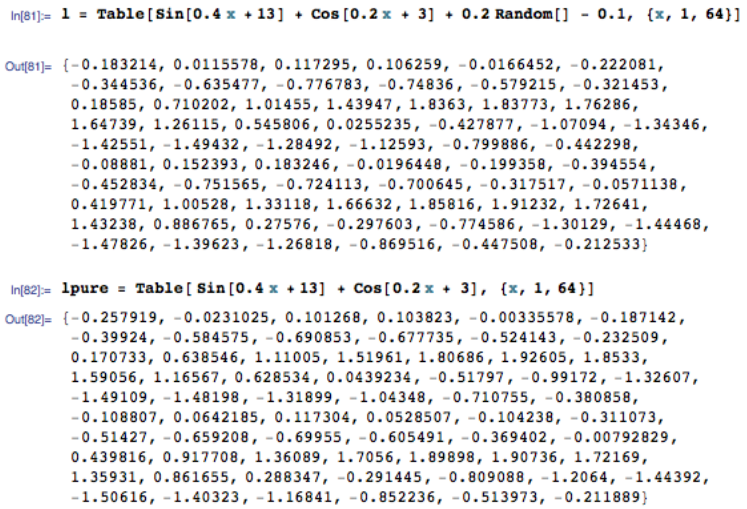




- *Filtering* – By eliminating undesirable high- and/or low-frequency components (i.e. dropping some of the sine functions) and taking an inverse Fourier transform to get us back into the time domain, we can filter a function to remove noise and other artifacts.
- *Compression* – A smoothed function less information than a noisy function, while retaining a similar appearance. We can perform lossy compression by eliminating the coefficients of sine functions that contribute relatively little to the function.

## A Fourier Transform Tale

We add white noise to points sampled from the sum of two trigonometric series:



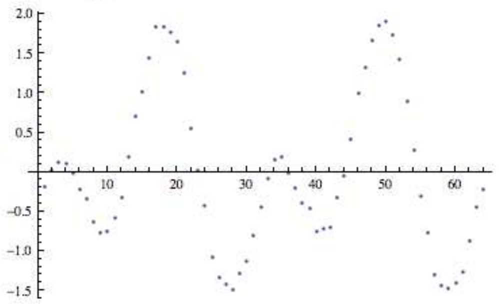
```
In[81]:= l = Table[Sin[0.4 x + 13] + Cos[0.2 x + 3] + 0.2 Random[] - 0.1, {x, 1, 64}]

Out[81]= {-0.183214, 0.0115578, 0.117295, 0.106259, -0.0166452, -0.222081,
          -0.344536, -0.635477, -0.776783, -0.74836, -0.579215, -0.321453,
          0.18585, 0.710202, 1.01455, 1.43947, 1.8363, 1.83773, 1.76286,
          1.64739, 1.26115, 0.545806, 0.0255235, -0.427877, -1.07094, -1.34346,
          -1.42551, -1.49432, -1.28492, -1.12593, -0.799886, -0.442298,
          -0.08881, 0.152393, 0.183246, -0.0196448, -0.199358, -0.394554,
          -0.452834, -0.751565, -0.724113, -0.700645, -0.317517, -0.0571138,
          0.419771, 1.00528, 1.33118, 1.66632, 1.85816, 1.91232, 1.72641,
          1.43238, 0.886765, 0.27576, -0.297603, -0.774586, -1.30129, -1.44468,
          -1.47826, -1.39623, -1.26818, -0.869516, -0.447508, -0.212533}

In[82]:= lpure = Table[ Sin[0.4 x + 13] + Cos[0.2 x + 3], {x, 1, 64}]

Out[82]= {-0.257919, -0.0231025, 0.101268, 0.103823, -0.00335578, -0.187142,
          -0.39924, -0.584575, -0.690853, -0.677735, -0.524143, -0.232509,
          0.170733, 0.638546, 1.11005, 1.51961, 1.80686, 1.92605, 1.8533,
          1.59056, 1.16567, 0.628534, 0.0439234, -0.51797, -0.99172, -1.32607,
          -1.49109, -1.48198, -1.31899, -1.04348, -0.710755, -0.380858,
          -0.108807, 0.0642185, 0.117304, 0.0528507, -0.104238, -0.311073,
          -0.51427, -0.659208, -0.69955, -0.605491, -0.369402, -0.00792829,
          0.439816, 0.917708, 1.36089, 1.7056, 1.89898, 1.90736, 1.72169,
          1.35931, 0.861655, 0.288347, -0.291445, -0.809088, -1.2064, -1.44392,
          -1.50616, -1.40323, -1.16841, -0.852236, -0.513973, -0.211889}
```

## A Periodic Function, with White Noise

The behavior of the sampled function remains apparent even in the presence of noise:

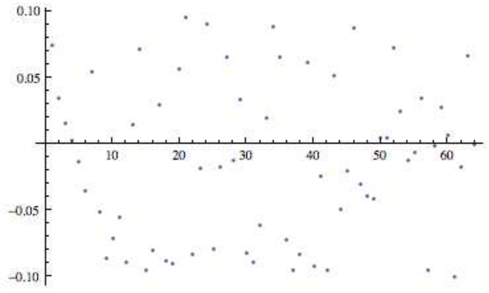
In[86]:= `ListPlot[l]`

Out[86]=

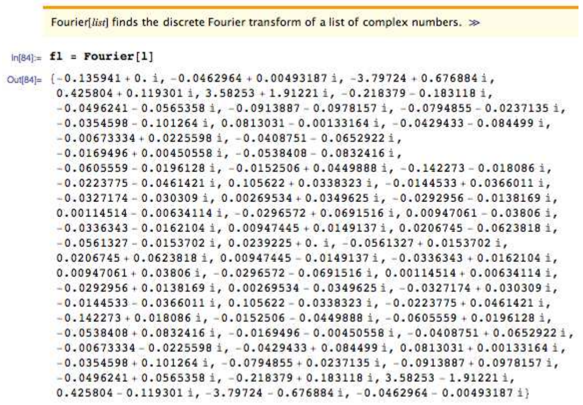


In[83]:= `ListPlot[l - lpure]`

Out[83]=



## Transformed Data

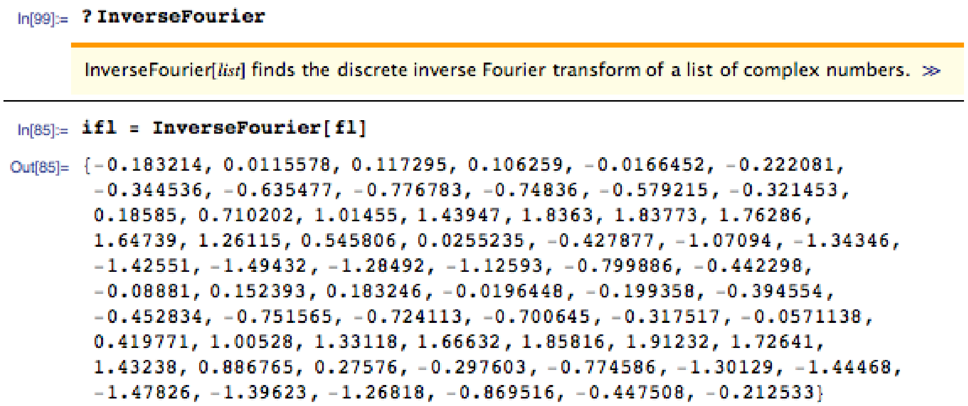


Fourier[*list*] finds the discrete Fourier transform of a list of complex numbers. ≫

```
In[84]:= fl = Fourier[l]

Out[84]= {-0.135941 + 0. i, -0.0462964 + 0.00493187 i, -3.79724 + 0.676884 i,
  0.425804 + 0.119301 i, 3.58253 + 1.91221 i, -0.218379 - 0.183118 i,
  -0.0496241 - 0.0565358 i, -0.0913887 - 0.0978157 i, -0.0794855 - 0.0237135 i,
  -0.0354598 - 0.101264 i, 0.0813031 - 0.00133164 i, -0.0429433 - 0.084499 i,
  -0.00673334 + 0.0225598 i, -0.0408751 - 0.0652922 i,
  -0.0169496 + 0.00450558 i, -0.0538408 - 0.0832416 i,
  -0.0605559 - 0.0196128 i, -0.0152506 + 0.0449888 i, -0.142273 - 0.018086 i,
  -0.0223775 - 0.0461421 i, 0.105622 + 0.0338323 i, -0.0144533 + 0.0366011 i,
  -0.0327174 - 0.030309 i, 0.00269534 + 0.0349625 i, -0.0292956 - 0.0138169 i,
  0.00114514 - 0.00634114 i, -0.0296572 + 0.0691516 i, 0.00947061 - 0.03806 i,
  -0.0336343 - 0.0162104 i, 0.00947445 + 0.0149137 i, 0.0206745 - 0.0623818 i,
  -0.0561327 - 0.0153702 i, 0.0239225 + 0. i, -0.0561327 + 0.0153702 i,
  0.0206745 + 0.0623818 i, 0.00947445 - 0.0149137 i, -0.0336343 + 0.0162104 i,
  0.00947061 + 0.03806 i, -0.0296572 - 0.0691516 i, 0.00114514 + 0.00634114 i,
  -0.0292956 + 0.0138169 i, 0.00269534 - 0.0349625 i, -0.0327174 + 0.030309 i,
  -0.0144533 - 0.0366011 i, 0.105622 - 0.0338323 i, -0.0223775 + 0.0461421 i,
  -0.142273 + 0.018086 i, -0.0152506 - 0.0449888 i, -0.0605559 + 0.0196128 i,
  -0.0538408 + 0.0832416 i, -0.0169496 - 0.00450558 i, -0.0408751 + 0.0652922 i,
  -0.00673334 - 0.0225598 i, -0.0429433 + 0.084499 i, 0.0813031 + 0.00133164 i,
  -0.0354598 + 0.101264 i, -0.0794855 + 0.0237135 i, -0.0913887 + 0.0978157 i,
  -0.0496241 + 0.0565358 i, -0.218379 + 0.183118 i, 3.58253 - 1.91221 i,
  0.425804 - 0.119301 i, -3.79724 - 0.676884 i, -0.0462964 - 0.00493187 i}
```

The Fourier transform converts $n$ real numbers into $n$ complex numbers.
Note that many of the magnitudes are small, i.e. near zero.

## Transforming Back

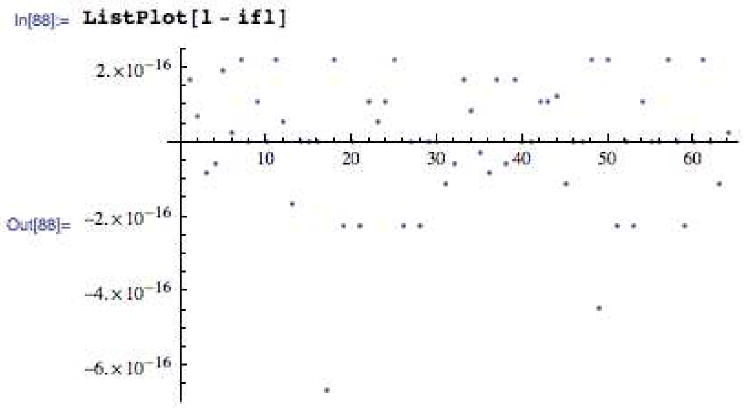


```
In[99]:= ?InverseFourier

InverseFourier[list] finds the discrete inverse Fourier transform of a list of complex numbers. >>

In[85]:= ifl = InverseFourier[fl]

Out[85]= {-0.183214, 0.0115578, 0.117295, 0.106259, -0.0166452, -0.222081,
  -0.344536, -0.635477, -0.776783, -0.74836, -0.579215, -0.321453,
  0.18585, 0.710202, 1.01455, 1.43947, 1.8363, 1.83773, 1.76286,
  1.64739, 1.26115, 0.545806, 0.0255235, -0.427877, -1.07094, -1.34346,
  -1.42551, -1.49432, -1.28492, -1.12593, -0.799886, -0.442298,
  -0.08881, 0.152393, 0.183246, -0.0196448, -0.199358, -0.394554,
  -0.452834, -0.751565, -0.724113, -0.700645, -0.317517, -0.0571138,
  0.419771, 1.00528, 1.33118, 1.66632, 1.85816, 1.91232, 1.72641,
  1.43238, 0.886765, 0.27576, -0.297603, -0.774586, -1.30129, -1.44468,
  -1.47826, -1.39623, -1.26818, -0.869516, -0.447508, -0.212533}
```

The inverse Fourier transform converts $n$ complex numbers into $n$ real numbers.

The resulting series should look familiar. . .

## Near Perfect Inversion



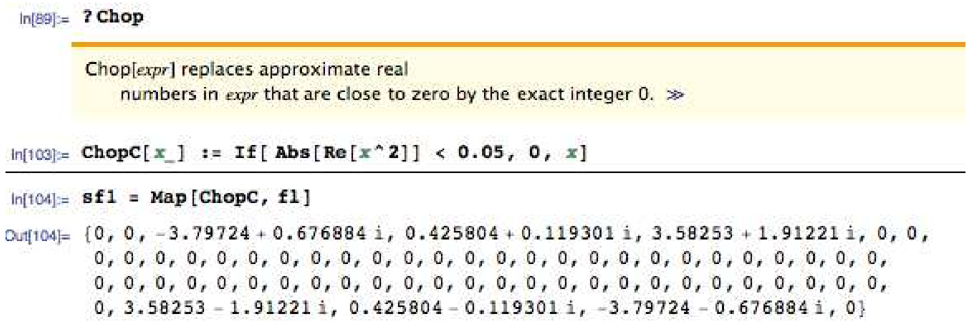
```
In[88]:= ListPlot[l - ifl]
```

```
Out[88]=
```

The inverse transform recovers the original sequence modulo tiny floating point errors.

## Filtering the Transformed Data

We can filter/compress the transformed data by replacing all small magnitude numbers by zero:

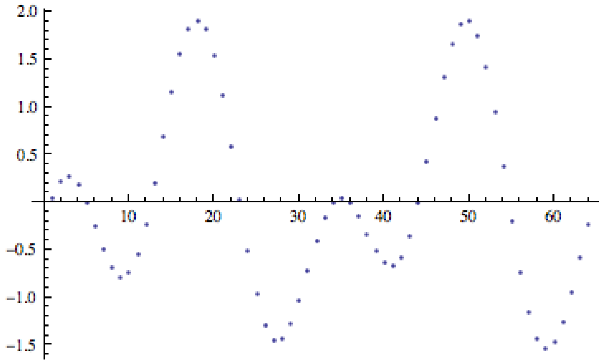


```
In[89]:= ? Chop

Chop[expr] replaces approximate real
    numbers in expr that are close to zero by the exact integer 0.  >>

In[103]:= ChopC[x_] := If[ Abs[Re[x^2]] < 0.05, 0, x]

In[104]:= sf1 = Map[ChopC, f1]

Out[104]= {0, 0, -3.79724 + 0.676884 i, 0.425804 + 0.119301 i, 3.58253 + 1.91221 i, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 3.58253 - 1.91221 i, 0.425804 - 0.119301 i, -3.79724 - 0.676884 i, 0}
```

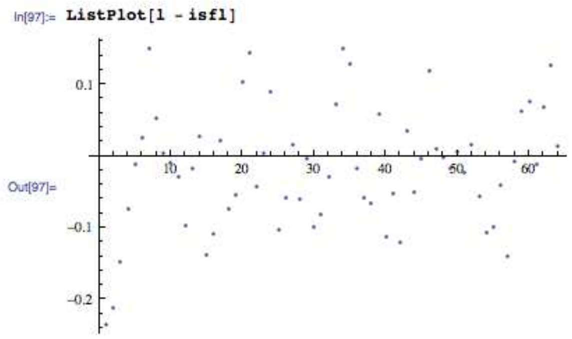
Only three complex numbers remain, defining a very simple model. Note the symmetry.

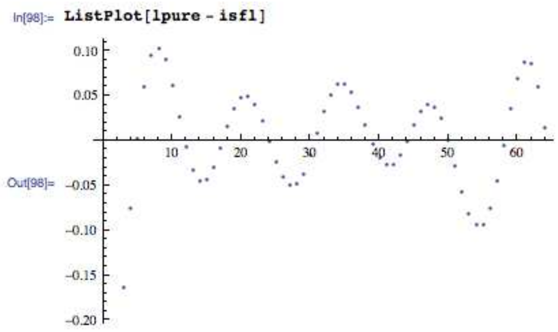
## Recovering the Series From Filtered Data

```
In[92]:= isfl = InverseFourier[sfl]

Out[92]= {0.052773, 0.222868, 0.264133, 0.179686, -0.00526127, -0.247671,
          -0.493734, -0.687622, -0.780594, -0.73905, -0.550304, -0.225158,
          0.203106, 0.683208, 1.15314, 1.54915, 1.81512, 1.91082, 1.81797,
          1.54308, 1.11659, 0.588641, 0.0218966, -0.517341, -0.968491,
          -1.28509, -1.44095, -1.43327, -1.28216, -1.02673, -0.718474,
          -0.412779, -0.160129, 0.00181806, 0.0539711, -0.00148848, -0.141845,
          -0.327902, -0.510703, -0.639868, -0.672229, -0.579407, -0.35313,
          -0.00743453, 0.42263, 0.885626, 1.32102, 1.66804, 1.87477, 1.9061,
          1.74929, 1.41635, 0.942725, 0.382611, -0.198552, -0.733223, -1.16122,
          -1.43805, -1.54099, -1.47177, -1.25579, -0.93778, -0.574592, -0.22636}

In[93]:= ListPlot[isfl]

Out[93]=
```



## Garbage In, Data Out?

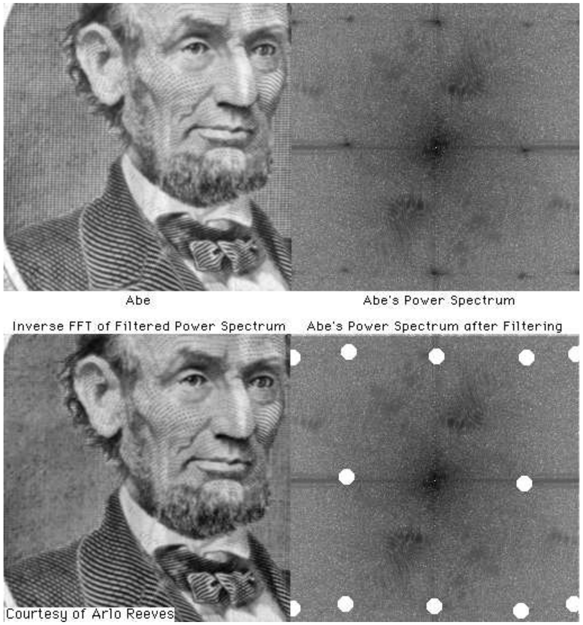




The random noise has been filtered away, leaving smaller residuals with the "correct" sequence than the input sequence.

## FFT Image Filtering Example

Note how very particular cross-hatching was removed by eliminating the appropriate transform coefficients.


Abe
Abe's Power Spectrum
Inverse FFT of Filtered Power Spectrum
Abe's Power Spectrum after Filtering
Courtesy of Arlo Reeves

## Convolutions and Correlation

- *Convolution and Deconvolution* – A *convolution* is the pairwise product of elements from two different sequences, such as in multiplying two $n$-variable polynomials $f$ and $g$ or multiplying two long integers. Implementing the product directly takes $O(n^2)$, while $O(n \lg n)$ suffices using the fast Fourier transform.
- *Computing the correlation of functions* – The *correlation function* of two functions $f(t)$ and $g(t)$ is defined by

  $$z(t) = \int_{-\infty}^{\infty} f(\tau) g(t + \tau)$$

  and can be easily computed using Fourier transforms.

If the two functions are similar in shape but one is shifted relative to the other (such as $f(t) = \sin(t)$ and $g(t) = \cos(t)$), the value of $z(t_0)$ will be large at this shift offset $t_0$.

As an application, suppose that we want to detect whether there are any funny periodicities (autocorrelations) in a time series or random number generator. When we take the Fourier transform of this series, any large spikes will correspond to potential periodicities.

## Polynomials: A Refresher

We can represent a given polynomial

$$A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n_1} x^{n-1}$$

by either (1) the set of coefficients $a_i$, or (2) a set of $n$ points $(x_i, y_i)$ on the polynomial, i.e. $y_i = A(x_i)$.

## Fast Operations on Polynomials

Given the coefficient representation, we can add or subtract two polynomials in $O(n)$ time by just adding or subtracting each pair of corresponding terms.

Given the coefficient representation, we can *evaluate* a polynomial in linear time using *Horner's rule*

$$A(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \ldots)))$$

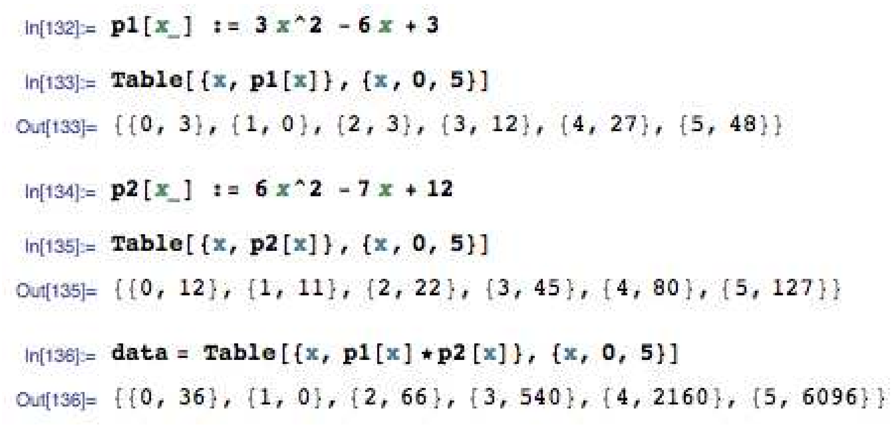
But it is not clear how to multiply two polynomials in coefficient representation in less than $O(n^2)$.

## Fast Operations on Point Sets

However, it is easy to add and subtract pairs of polynomials in point-value representation (assuming the same set of $x_i$ values is used in both polynomials) by operating on the pair of points for each $x$ value.

*We can multiply coefficient polynomials by (1) converting them to point-value representation, (2) multiply these point-wise in $O(n)$, and (3) interpolate these $2n$ points back to a polynomial.*

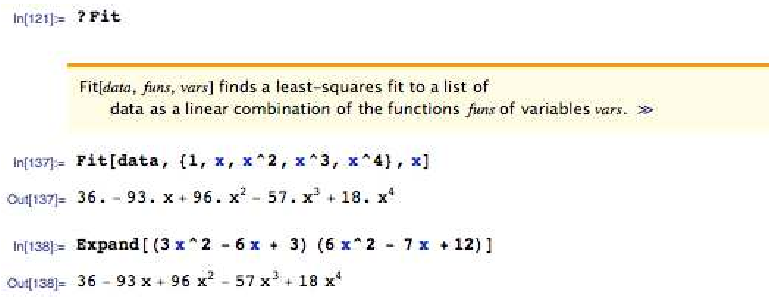
## Multiplication Example



```
In[132]:= p1[x_] := 3 x^2 - 6 x + 3

In[133]:= Table[{x, p1[x]}, {x, 0, 5}]
Out[133]= {{0, 3}, {1, 0}, {2, 3}, {3, 12}, {4, 27}, {5, 48}}

In[134]:= p2[x_] := 6 x^2 - 7 x + 12

In[135]:= Table[{x, p2[x]}, {x, 0, 5}]
Out[135]= {{0, 12}, {1, 11}, {2, 22}, {3, 45}, {4, 80}, {5, 127}}

In[136]:= data = Table[{x, p1[x] * p2[x]}, {x, 0, 5}]
Out[136]= {{0, 36}, {1, 0}, {2, 66}, {3, 540}, {4, 2160}, {5, 6096}}
```

Note that the product of two $n$-degree polynomials has degree $2n$.

## Multiplication by Interpolation



```
In[121]:= ?Fit

Fit[data, funs, vars] finds a least-squares fit to a list of
    data as a linear combination of the functions funs of variables vars. >>

In[137]:= Fit[data, {1, x, x^2, x^3, x^4}, x]

Out[137]= 36. - 93. x + 96. x^2 - 57. x^3 + 18. x^4

In[138]:= Expand[(3 x^2 - 6 x + 3) (6 x^2 - 7 x + 12)]

Out[138]= 36 - 93 x + 96 x^2 - 57 x^3 + 18 x^4
```

Steps 1 (conversion) and 3 (interpolation) look quadratic, but can in fact be done on $O(n \log n)$ by using the DFT and inverse transform.

Lagrange's formula solves the interpolation problem in $O(n^2)$, but it is too slow.

## Complex Numbers: A Refresher

Although any polynomial of degree $d$ should have $d$ roots/solutions, they sometimes require *complex numbers*:

$$x^2 + 1 = 0$$

If $i = \sqrt{-1}$, the two solutions are $i$ and $-i$.
The $n$ *roots of unity* are the $n$ solutions to the equation

$$x^n = 1$$

These roots are defined by the $n$ powers $\omega_n^i$ for $0 \leq i \leq n-1$, where

$$\omega_n = \cos(2\pi/n) + i\sin(2\pi/n) = e^{2\pi i/n}$$

The identity linking the trigonometric functions to $e$ is

$$e^{iu} = \cos u + i \sin u$$

## The Discrete Fourier Transform

The discrete Fourier transform takes as input $n$ a time series of $n$ equally-spaced complex numbers $h_k$, $0 \leq k \leq n-1$.
It outputs $n$ complex numbers $H_k$, $0 \leq k \leq n-1$, each describing a trigonometric function of the given frequency.
The discrete Fourier transform is defined by

$$H_m = \sum_{k=0}^{n-1} h_k e^{-2\pi ikm/n}$$

and the inverse Fourier transform is defined by

$$h_m = \frac{1}{n} \sum_{k=0}^{n-1} H_k e^{2\pi ikm/n}$$

which enables us move easily between $h$ and $H$.
But the complexity of a naive implementation is $O(n^2)$.

## The FFT Algorithm

The critical step in efficiently computing the DFT takes as input a set of $n$ complex numbers $a_0, a_1, \ldots, a_{n-1}$ and outputs the sequence of $n$ complex numbers

$$A(1), A({\omega_n}^1), A({\omega_n}^2), \ldots, A({\omega_n}^{n-1})$$

resulting from evaluating the polynomial

$$A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n_1} x^{n-1}$$

In particular, to evaluate $A(x)$ when $n$ is even, let

$$A_{even}(y) = a_0 + a_2 y + \ldots + a_{n-2} y^{n/2-1}$$

$$A_{odd}(y) = a_1 + a_3 y + \ldots + a_{n-1} y^{n/2-1}$$

It should be clear that

$$A(x) = A_{even}(x^2) + xA_{odd}(x^2)$$

FFT algorithms are based on divide-and-conquer. Essentially, the problem of computing the discrete Fourier transform on $n$ points is reduced to computing two transforms on $n/2$ points each and is then applied recursively.
By itself, this recurrence does not help us, since we have to spend linear time to evaluate it at each of $n$ points.
But the fact that the $n$ points are all the complex roots of unity provide the magic to speed things up.

## Implementation Details

The FFT algorithm assumes that $n$ is a power of two.
If this is not the case for your data, you are better off padding your data with zeros to create $n = 2^k$ elements rather than hunting for a more general code.
Some care is needed to determine where to best pad the zeros.
Historically, the FFT has been implemented in assembler or even hardware for performance optimization.
Highly optimized FFT implementation exist which tune themselves to your specific hardware configurations (e.g. cache size). Check out the FFTW (Fastest Fourier Transform in the West).

## Wavelets

In recent years, wavelets have been proposed as a generalization of Fourier transforms in filtering.
Wavelets are ‘small waves’, sinusoids are ‘big waves’.
A real-valued function $\Phi(.)$ defined over the real axis is a wavelet if

- its integral is zero, $\int \Phi(u)du = 0$, and
- the integral of $\Phi^2(.)$ is unity, $\int \Phi^2(u)du = 1$,

Such properties hold for $\Phi(u) = \sin(x)/\sqrt{\pi}$.
The wavelet transforms work for more general basis functions than just sine waves.