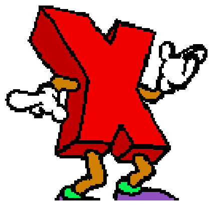
Fast Fourier Transform (FFT) with applications

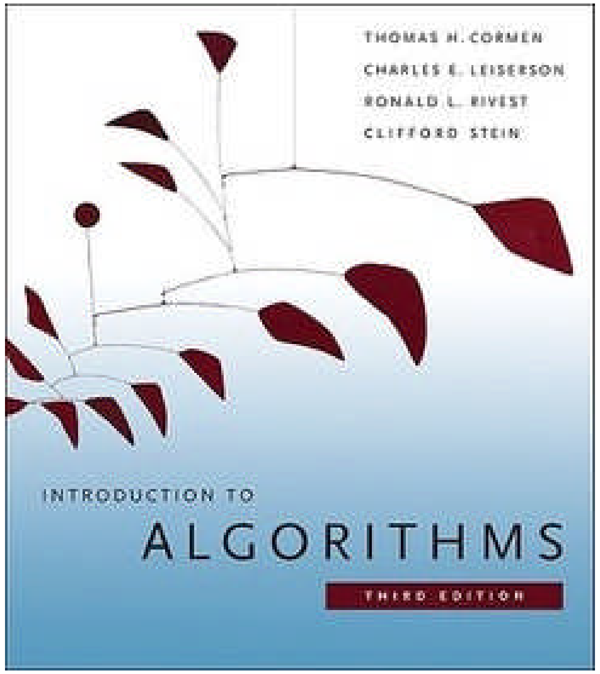


Slides compiled from the internet (google, wikipedia, books)
MoshAhmed at gmail
29/8/2013

Textbook and references

- Cormen, 3rd edition.

  http://www.flipkart.com/introduction-algorithms-8120340078



References

- Design and Analysis of Computer Algorithms,

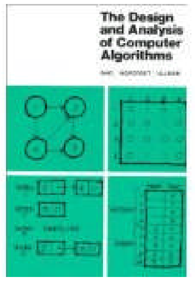
  by Ullman and Hopcroft

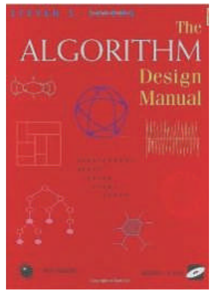
  http://www.flipkart.com/design-analysis-computer-algorithms-8131702057
- Algorithm Design Manual, 2nd ed, by Skiena

  http://www.flipkart.com/algorithm-design-manual-8184898657

Google, Wikipedia

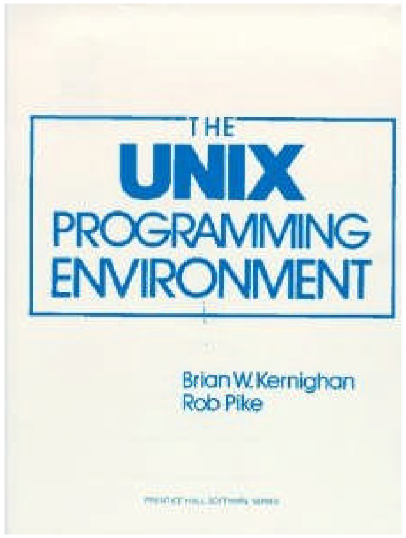
- Mathematics for CS by Meyer, from MIT.
- MIT courseware video lectures

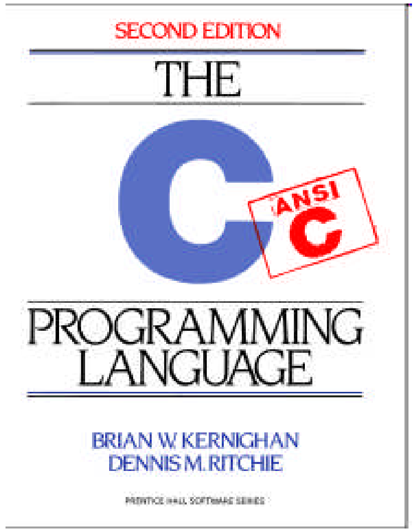


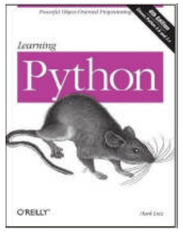


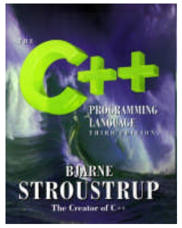
Programming textbooks

- K&R: C Programming Language, by Kernighan and Ritchie, 2nd ed (not older ed).
- Unix Programing Environment, by Kernighan and Pike.
- C++ Language, by Stroustrup, 3rd ed (not older ed).
- Learning Python (or docs.python.org)
- Core Java, Vol 1 & 2, by Horstmann


THE UNIX PROGRAMMING ENVIRONMENT
Brian W. Kernighan
Rob Pike


SECOND EDITION
THE C PROGRAMMING LANGUAGE
ANSI C
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE
PRENTICE HALL SOFTWARE SERIES


Learning Python
O'REILLY


THE C++ PROGRAMMING LANGUAGE
THIRD EDITION
BJARNE STROUSTRUP
The Creator of C++

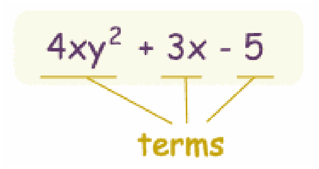
core JAVA 2
Volume I–Fundamentals
SEVENTH EDITION
Sun
SUN MICROSYSTEMS PRESS
CAY S. HORSTMANN · GARY CORNELL

Lab

- C programs using gcc/g++ IDE from http://www.codeblocks.org or MS-VC, not dev-cpp. recent gcc 4.
- On Linux or Windows (install cygwin for bash, make, configure, vim, ctags).
- Sometimes we will use Python, Java, C++, js
- Demonstration packages: Maple, Mathematica

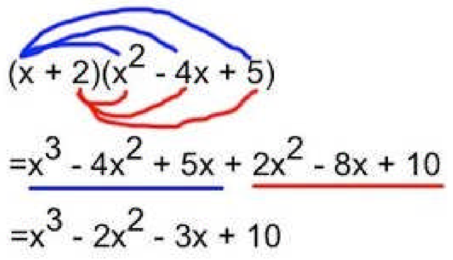
Examples

E.g. Polynomial in 2 variables with 3 terms



E.g. Polynomial multiplication

$$(2x+3)(5x+4)=?$$
$$=(2x)(5x)+(2x)(4)+(3)(5x)+(3)(4)$$
$$=10x^2+8x+15x+12$$
$$=10x^2+23x+12$$



$$(x+2)(x^2-4x+5)$$
$$=\underline{x^3-4x^2+5x}+\underline{2x^2-8x+10}$$
$$=x^3-2x^2-3x+10$$

Polynomials

Polynomial: $A(x) = \sum_{j=0}^{n-1} a_j x^j$

coefficient: complex, e.g.,

$3.25 + 6.999i$

real part

imaginary part

$\sqrt{-1}$

Degree-bound of A(x) is n.

Degree is k if a_k is the highest non-zero coefficient.

Polynomial Addition

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

$$B(x) = \sum_{j=0}^{n-1} b_j x^j$$

$$C(x) = A(x) + B(x) = \sum_{j=0}^{n-1} c_j x^j, \ c_j = a_j + b_j$$

$\Theta(n)$ time to compute.

Example of Polynomial Multiplication

$$\begin{array}{rrrrrrrrrrrrr}
 & & & & & & 6x^3 & + & 7x^2 & - & 10x & + & 9 \\
 & & & & & & -2x^3 & & & + & 4x & - & 5 \\
\cline{7-13}
 & & & & & & -30x^3 & - & 35x^2 & + & 50x & - & 45 \\
 & & & & 24x^4 & + & 28x^3 & - & 40x^2 & + & 36x & & \\
-12x^6 & - & 14x^5 & + & 20x^4 & - & 18x^3 & & & & & & \\
\hline
-12x^6 & - & 14x^5 & + & 44x^4 & - & 20x^3 & - & 75x^2 & + & 86x & - & 45
\end{array}$$

Q. What is the complexity of this?

Usual multiplication

- Normal multiplication would take $(2n)^2$ cross multiplies and some data manipulation, $O(n^2)$.

Polynomial Multiplication

$$C(x) = A(x) \cdot B(x)$$

$$= \sum_{j=0}^{2n-2} c_j x^j$$

$$\text{where } c_j = \sum_{k=0}^{j} a_k b_{j-k}$$

$$\uparrow$$

called **convolution**

Note: Degree bound of C(x) is 2n – 1.

Straightforward computation takes $\Theta(n^2)$ time.

Maple

```
C:\MAPLEV4\BIN.WIN\WMAPLE32.EXE

>  P:= x^3 + 5*x^2 + 11*x + 15;
>  Q:= 2*x + 3;
> degree(P,x);               =  3
> coeff(P,x,1);              = 11
> coeffs(P,x);               = {15, 11, 1, 5}
> subs(x=3,P);               = 120
```

Polynomials in Maple

```
> P:= x^3+5*x^2+11*x+15;
> Q := 2*x + 3;
> m := expand(P * Q);
    m = 2 x^4  + 13 x^3  + 37 x^2  + 63 x + 45.

> factor(m);
(2*x+3)*(x+3)*(x^2+2*x+5)

> roots(m);   … {-3,-3/2}      … Two Real roots.
> roots(m, I) … {-1±2i,-3,-3/2}… Complex roots
```

Pari

- Pari is a free *Computer Algebra Software* for Linux and windows.
- C:\tools\pari> gp.exe
- gp > m = 2*x^4 + 13*x^3 + 37*x^2 + 63*x + 45;
- gp > factor(m)

  [x + 3, 1], [2*x + 3, 1], [x^2 + 2*x + 5, 1]
- gp > log(x+1)   .... Get series expansion

  x - 1/2*x^2 + 1/3*x^3 - 1/4*x^4 + 1/5*x^5 - 1/6*x^6...

Multiplication of polynomials

- We assume P and Q are two polynomials in x of equal size (terms) and of odd degree.
- If they are not, pad them with 0 coefficients.

Divide and Conquer multiplication

Divide P and Q of size 2n each into smaller polynomials of size n each:

$P = a^{2n-1}x^{2n-1} + \dots + a_0$ … is a poly of size 2n, (2n terms).
factor common x^n from first n terms:
$P = x^n (a_{2n-1}x^{n-1} + \dots \quad + a_{n+1}x \quad + a_n \quad) + (a_{n-1}x^{n-1} + \dots + a_0)$

$P = A\,x^n + B$

Similarly factor Q:
$Q = C\,x^n + D$

Usual multiplication of P*Q

Divide P and Q of size 2n each into smaller polynomials of size n each:

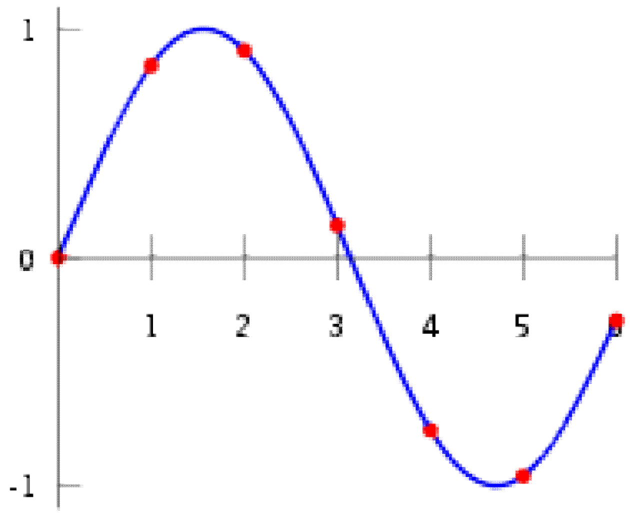
- $P = A\,x^n + B$
- $Q = C\,x^n + D$
- $P*Q = (A*C)(x^n)^2 + (A*D+B*C)(x^n) + (B*D)$
- There are 4 multiplies of size n polynomials, plus a size 2n polynomial addition:
- $O(4*(n^2))$. .. no better!

Faster Multiplication,

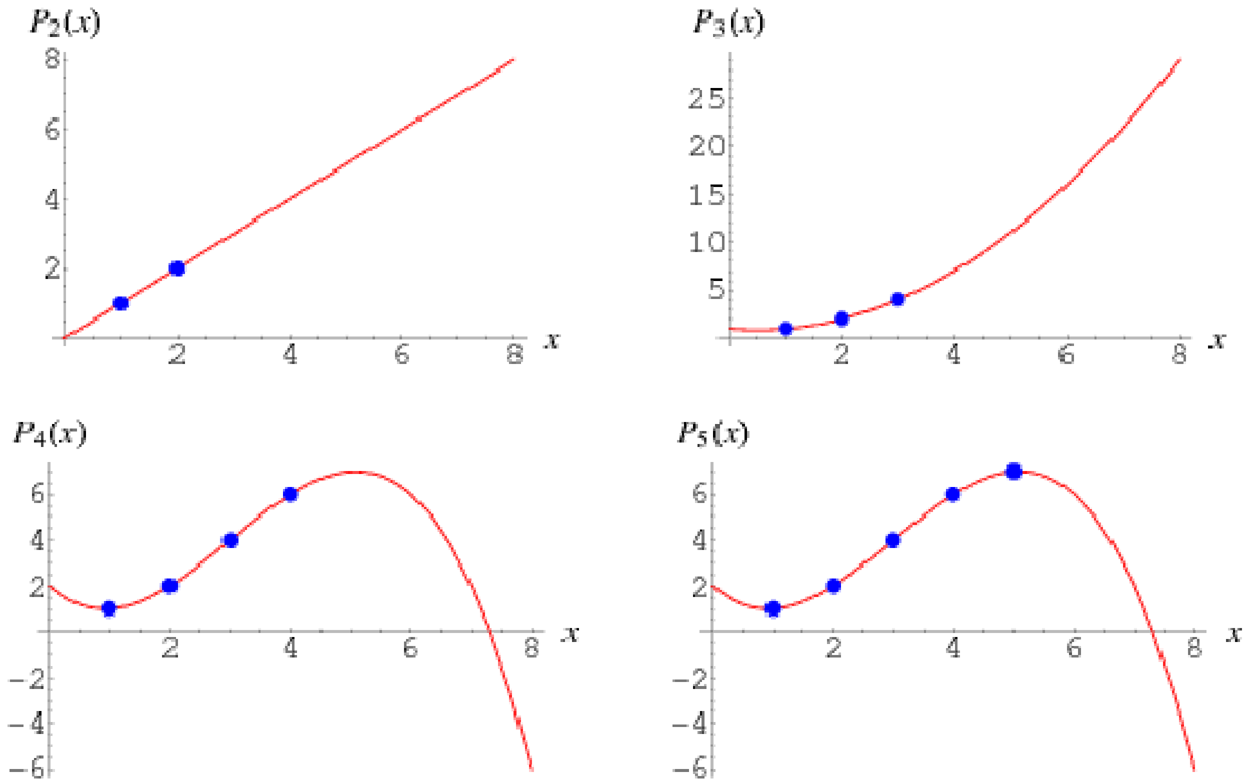
using Polynomials

Polynomial interpolation

The red dots denote the data points (x_i,y_i),
while the blue curve shows the interpolation polynomial.



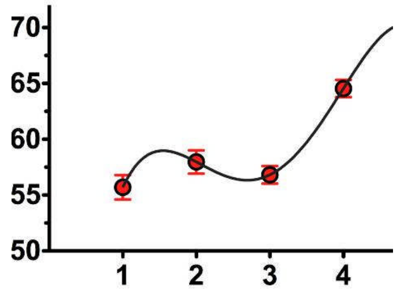
Polynomial interpolation



Polynomial interpolation

- Suppose we have 4 data points:

  (x1, y1), (x2, y2), (x3, y3), and (x4, y4).
- There is exactly one polynomial of deg 3, $p(x) = ax^3 + bx^2 + cx + d$
- which passes through the four points.

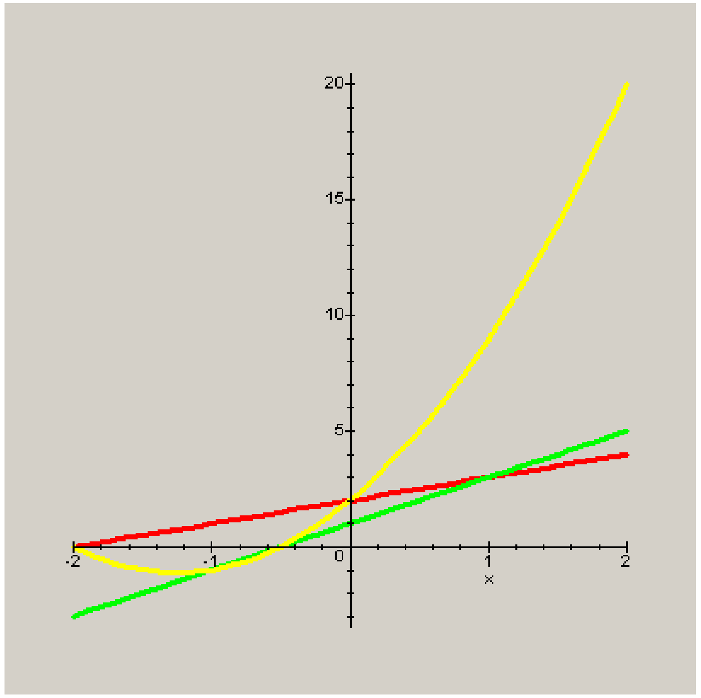


Interpolate in Maple

```
# Evaluate P(x) and Q(x) at 3 points x=[-1,0,1,-1]
# And Interpolate at 3 points ([x1,x2,x3], [y1,y2,y3])
> p := x -> 1 + 2 * x;
                          p := x -> 1 + 2 x
> q := x -> 2 + x;
                           q := x -> 2 + x
> seq( p(x)*q(x), x=-1..1);
                              -1, 2, 9
> interp([-1,0,1],[-1,2,9], x);
                            2 x^2  + 5 x + 2
```

Plot the graphs

```
> plot({p(x), q(x), p(x)*q(x)},x=-2..2}
```

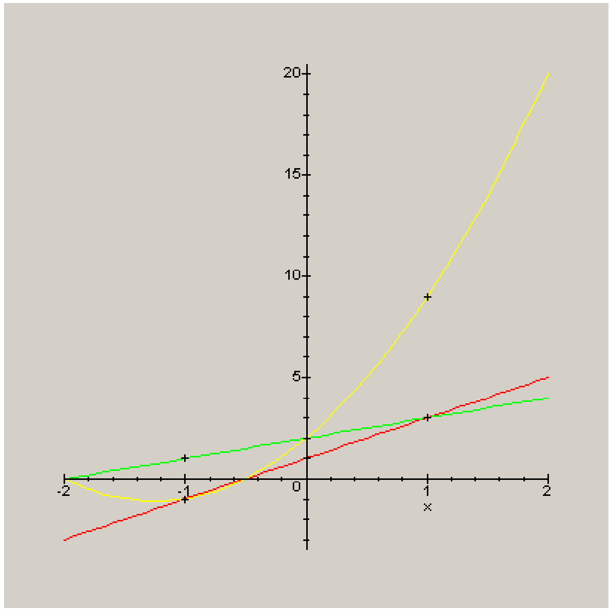


In the plot:

p(x) :	green
q(x):	red
p(x)*q(x):	yellow

All the maple commands together

```
1.  p := x -> 1 + 2 * x;
2.  q := x -> 2 + x;
3.  seq(p(x)*q(x), x=-1..1);
4.  interp([-1,0,1],[-1,2,9], x);
5.  py:= {seq([x,p(x)], x=-1..1)};
6.  qy:= {seq([x,q(x)], x=-1..1)};
7.  pqy:= {seq([x,p(x)*q(x)], x=-1..1)};
8.  pl1:=plot({p(x), q(x), p(x)*q(x)},x=-2..2):
9.  pl2:=pointplot(py):
10. pl3:=pointplot(qy):
11. pl4:=pointplot(pqy);
12. with(plots):
13. display({pl1,pl2,pl3,pl4});
```



Vandermonde matrix

- Given the points $\{p(x_i) : i=0..n\}$
- we can compute the polynomial

  $p(x)= a_n x^n + .. + a_0$
- using the matrix inverse:

$$\begin{bmatrix} x_0^n & x_0^{n-1} & x_0^{n-2} & \dots & x_0 & 1 \\ x_1^n & x_1^{n-1} & x_1^{n-2} & \dots & x_1 & 1 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ x_n^n & x_n^{n-1} & x_n^{n-2} & \dots & x_n & 1 \end{bmatrix} \begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_0 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

Multiply polynomials h(x)= f(x)*g(x) by Interpolation at n points

A. for i=0…n;

   evaluate f(i) and g(i);

   save the value of h(i) = f(i) * g(i).

B. Interpolate { (h(i),i) : i=0..n } to the unique polynomial h(x) that passes through these points: (0,h(0)),(1,h(1))...(n,h(n)).

Horner’s rule to evaluate P(X)

- Evaluating $P(x)$ at a point x_0.
- Brute force takes O(n^2)

  $P(x) = a_0 + x\, a_1 + x^2 a_2 + x^3 a_3 \ldots$
- ***Horner’s rule*** does it in O(n) by factoring on x:

  $P(x) = a_0 + x\, (a_1 + x\, (a_2 + x\, (\ldots)))$
- Doing it at n points takes O(n^2).

Horner's Polynomial Evaluation

- Given the coefficients $(a_0,a_1,a_2,\ldots,a_{n-1})$ of x^i in p(x).
- Evaluate p(x) at z in O(n) time using:

- **Function Horner**($A=(a_0,a_1,a_2,\ldots,a_{n-1})$,z):

```
If n==1 then return $a_0$
else return ($a_0$+ z * **Horner**(A'=($a_1,a_2,\ldots,a_{n-1}$),z) )
```

Using Maple

```
> convert( 5 * x^5 + 3 * x^3 + 22 * x^2 + 55,
  horner);

55+(22+(3+5*x^2)*x)*x^2
```

Homework

Write C program to compute p(x) using horner rule, given polynomial coeffs as p[ ] and points x[ ].

e.g. p(x) = 5 * x^5 + 3 * x^3 + 22 * x^2 + 55

Horner of p(x) is 55+(22+(3+5*x^2)*x)*x^2

```
double p [ ] = { 55,0,22,3,0,5};
double x [ ] = { 1, 10, 20};
for k in 0..2
   print horner(p, x[k])
```

Homework solution: horner in C

```
double horner( double *coeffs, int n, double x ) {
    double y = 0.0;
    while ( n-- > 0)
        y = y * x + coeffs[n];
    return y;
}
#define LEN(A) (sizeof(A)/sizeof(A[0]))
int main(  ) {
    double k, p[] = { 1, 0, 3 };    // p(x):=1+0x+3x^2 = 1+x(0+3*x));
    for(k=-2;k<=2;k++)
      printf( "p(%g)= %g\n", k, horner( p, LEN(p), k) );
    return 0;
}
```

A peculiar way to multiply polynomials f(x) by g(x)

How much does this cost?

- Evaluations: 2*n evaluations of size(n) using Horner’s rule is O(n^2).
- Interpolation: using “Lagrange” or “Newton Divided Difference” is also O(n^2).

Evaluate P(x) faster?

- Our choice of points 0,1,2,... n was arbitrary.
- What if we choose “better” points?
- Evaluating P(0) is almost free.
- Evaluating at symmetric points:

  P(c) and P(-c) can be done “faster” by sharing the calculations
- e.g. take coefficients of odd and even powers separately.

Evaluate p(-x) faster from p(x)

- p1 = P_{ood}:= $a_{2n+1}x^{2n+1}+a_{2n-1}x^{n-1}+\ldots +a_1$
- p2= P_{even}:=$a_{2n}x^{2n}$ $+ \ldots + a_0$

- $P(x) = P_{odd}(x^2) *x + P_{even}(x^2)$
- P(x) = +p1*x+p2
- P(-x) = -p1*x+p2.
- So P(c) can be computed by evaluating
  - p1 = $P_{even}(c^2)$
  - p2 = $P_{odd}(c^2)$
  - and returning c*p1+p2
- Finally, P(-c) is just –c*p1+p2, nearly free.

Are complex numbers better?

- Take a number r (real, complex, finite field,..)
- Compute the 2^m roots of 1 = $\{1, w, w^2, w^3, \ldots w^{2^m}\}$, these roots have lot of symmetry.
- Evaluate P at these roots, exploiting symmetry to save on calculations.
- Evaluating $P(-w^k)$ and $P(w^k)$ can be done “faster”..
  - take even and odd coefficients separately,
  - by the same trick as before, with $c^2 = w^{2k}$

Enter: Numerical FFT

- Evaluate P at “complex roots of unity.”
- This reduces cost from n^2 to O(n log n);
- and same complexity for inverse (interpolation).

Fast multiplication of polynomials

$$\begin{tikzcd}[column sep=huge, row sep=huge]
\boxed{\begin{array}{c} a_0,\ a_1,\ \cdots\ a_{n-1} \\ b_0,\ b_1,\ \cdots\ b_{n-1} \end{array}} \arrow[r, "\text{Ordinary multiplication of coeffs } O(n^2)"] \arrow[d, "\begin{array}{l}\text{Evaluation at roots}\\ \text{of unity, } O(n \log n)\end{array}"] & \boxed{c_0,\ c_1,\ \cdots\ c_{2n-1}} & \textit{Coefficient representation} \arrow[l, Rightarrow] \\
\boxed{\begin{array}{c} A(\omega_{2n}^{0}),\ \ B(\omega_{2n}^{0}) \\ A(\omega_{2n}^{1}),\ \ B(\omega_{2n}^{1}) \\ \vdots \\ A(\omega_{2n}^{2n-1}),\ \ B(\omega_{2n}^{n-1}) \end{array}} \arrow[r, "\text{Pointwise multiplication } O(n)"] & \boxed{\begin{array}{c} C(\omega_{2n}^{0}) \\ C(\omega_{2n}^{1}) \\ \vdots \\ C(\omega_{2n}^{2n-1}) \end{array}} \arrow[u, "\begin{array}{l}\text{Interpolation}\\ O(n \log n)\end{array}"'] & \textit{Point-value representation} \arrow[l, Rightarrow]
\end{tikzcd}$$

Complex roots of Unity

(^{n}⌦1 for FFT)

Complex roots of unity

- *The n-th root of unity* is the complex number ω such that $\omega^n=1$.
- The *value* $w_n = exp(i\ 2\pi/n)$ is called ***the principal n-th root of unity*****.**

Where: $i=\sqrt{-1}$,

- $exp(i*u)=cos(u)+i*sin(u)$.

All the n roots of unity

The *n* roots are of unity are

$= \{\exp(k * i\ 2 \square / n), k=0..n-1\}$

$=\{1,w,w^2,w^3,..\}$

Primitive Roots of Unity

- A number ω is called a ***primitive n-th root of unity***, if $\omega^n = 1$.
- And $\omega = e^{2\pi \mathbf{i}/n}$, where i = sqrt(-1).
- The roots 1, ω, ω^2, …, ω^{n-1} are distinct

Properties of the roots of unity

Lemma-1:

For any integers $n \geq 0$ and $d > 0$,

(w_(d*n))^(d*k) = w_n^k

Proof: w_dn^dk

= exp(2pi i/dn)^(dk)

= exp(2pi i/n)^k = w_n^k

Complex roots of unity

Corollary: For any even integer *n* > 0,
w_n^(n/2) = w_2=-1

Proof:

exp(n/2 * i 2 □ / n) = exp(i □) = -1

Complex roots of unity

Lemma-2: If $n > 0$ is even, then the squares of the n-th roots of unity are the $(n/2)$ roots of unity:

$(w_n^k)^2 = w_{n/2}^k$

Proof:

$(w_n^{k+n/2})^2$

$= w_n^{2k+n}$

$= w_n^{2k} * w_n^n$

$= w_n^{2k} = (w_n^k)^2$

Complex roots of unity

Lemma-3:

For any integer $n \geq 1$ and $k >= 0$ not divisible by n,

S = sum_{j=0..n-1}{ (w_k^n)^j } = 0

Proof:

S = ((w_n^k)^n – 1) / (w_n^k – 1) Geometric series sum.

= ((w_n^n)^k – 1) / …

= (1 - 1)/ …

= 0

Properties of Primitive Roots of Unity

- **Inverse Property:** If ω is a primitive root of unity, then $\omega^{-1}=\omega^{n-1}$
  - Proof: $\omega\omega^{n-1}=\omega^n=1$
- **Cancellation Property:** if $-n<k<n$, $\sum_{j=0}^{n-1}\omega^{kj}=0$
  - Proof:

$$\sum_{j=0}^{n-1}\omega^{kj}=\frac{(\omega^k)^n-1}{\omega^k-1}=\frac{(\omega^n)^k-1}{\omega^k-1}=\frac{(1)^k-1}{\omega^k-1}=\frac{1-1}{\omega^k-1}=0$$

- **Reduction Property:** If w is a primitive (2n)-th root of unity, then ω^2 is a primitive n-th root of unity:
- Proof: If $1,\omega,\omega^2,\ldots,\omega^{2n-1}$ are all distinct, so is the subset $1,\omega^2,(\omega^2)^2,\ldots,(\omega^2)^{n-1}$

- **Reflective Property:** If n is even, then $\omega^{n/2} = -1$.
  - Proof: By the cancellation property, for k=n/2:

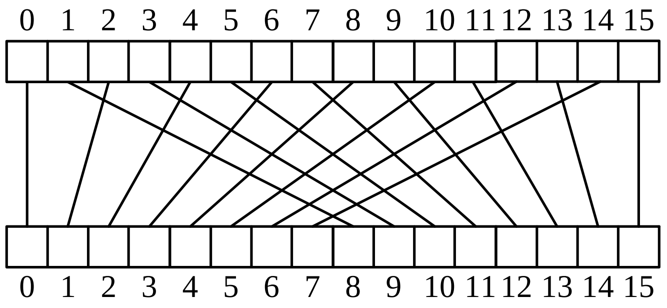
$$0=\sum_{j=0}^{n-1}\omega^{(n/2)j}=\omega^0+\omega^{n/2}+\omega^0+\omega^{n/2}+\cdots+\omega^0+\omega^{n/2}=(n/2)(1+\omega^{n/2})$$

  - Corollary: $\omega^{k+n/2}= -\omega^k$.

FFT
Fast Fourier Transform

The Fast Fourier Transform

in O(n log n).



The DFT

We wish to evaluate a polynomial
A(x) at each of the n-th roots of unity.

Assume *n* is a power of 2
(pad with 0 if required).

The DFT *Discrete Fourier Transform*

The vector y=(y_0..y_{n-1}) is called the **DFT** of the coefficient vector a=(a_0..a_{n-1}) of a polynomial A(x), and written as vector: y=DFT$_n$(a).

Each y_k is evaluated at the k-th root of unity (one):
y_k = A(w_n^k) = sum_{j=0..n-1}(a_j * w_n^{kj})
    for k=0,1, ..., n-1

Divide the Polynomial into Odd/Even

Write: $A(x) = A0(x^2) + x\ A1(x^2)$.

Where

$A0(x) = a_0 + a_2\ x + a_4\ x^2 + ..$

$A1(x) = a_1 + a_3\ x\ \ + a_5\ x^2 + ..$

Break the polynomial in odd/even

– If n is even, we can divide a polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

into two polynomials

$$\begin{aligned} p^{\text{even}}(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1} \\ p^{\text{odd}}(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1} \end{aligned}$$

$$p(x) = p^{\text{even}}(x^2) + x p^{\text{odd}}(x^2).$$

FFT divide and conquer

The problem of evaluating *A*(*x*) at {w_n^0..w_n^{n-1}} reduces to

1. Evaluating the two *n*/2 degree polynomials A1(x), A0(x) at square of {w_n^0..w_n^{n-1}}
2. Combining the results, we get y_k=A(w_n^k).

Recursive FFT

```
RECURSIVE-FFT($a$)
1   $n \leftarrow length[a]$
2   if $n = 1$
3       then return $a$
4   $\omega_n \leftarrow e^{2\pi i/n}$
5   $\omega \leftarrow 1$
6   $a^{[0]} \leftarrow (a_0, a_2, \ldots, a_{n-2})$
7   $a^{[1]} \leftarrow (a_1, a_3, \ldots, a_{n-1})$
8   $y^{[0]} \leftarrow$ RECURSIVE-FFT($a^{[0]}$)
9   $y^{[1]} \leftarrow$ RECURSIVE-FFT($a^{[1]}$)
10  for $k \leftarrow 0$ to $n/2 - 1$
11      do $y_k \leftarrow y_k^{[0]} + \omega y_k^{[1]}$
12          $y_{k+n/2} \leftarrow y_k^{[0]} - \omega y_k^{[1]}$
13          $\omega \leftarrow \omega\omega_n$
14  return $y$                              ▷ $y$ is assumed to be a vector.
```

Running time of RECURSIVE-FFT

- Each invocation takes time O(*n*), beside the recursive calls.
- T(n)=2 T(n/2) + O(n) = O(n log n).

Interpolation

We can write the DFT as the matrix product ***y* = *Va*** that is

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Interpolation, inverse matrix

Theorem: for j and k *in* {0,1, ..., *n*-1},

the (j,k) entry of the inverse of matrix is w_n^{-kj}/n.

And $V^* V^{-1} = I$

Proof: See CLR3 book, pg 913.

iDFT_n(y) = $(a_0..a_{n-1})$ can be computed using V^{-1}

a_j = 1/n * sum_{k=0..n-1}{y_k * w_n^{-kj}}

How to find inverse: F_n^{-1}?

Proposition. Let ω be a primitive l-th root of unity over a field L.Then

$$\sum_{k=0}^{l-1} \omega^k = \begin{cases} 0 & if\ l > 1 \\ 1 & otherwise \end{cases}$$

Proof. The l =1 case is immediate since ω=1.

Since ω is a primitive l-th root, each ω^k ,$k \neq 0$ is a distinct l-th root of unity.

$$Z^l - 1 = (Z - \omega_l^0)(Z - \omega_l)(Z - \omega_l^2)...(Z - \omega_l^{l-1}) =$$

$$= Z^l - (\sum_{k=0}^{l-1} \omega_l^k) Z^{l-1} + ... + (-1)^l \prod_{k=0}^{l-1} \omega_l^k$$

Comparing the coefficients of Z^{l-1} on the left and right hand sides of this equation proves the proposition.

Inverse matrix to F_n

Proposition. Let ω be an n-th root of unity.Then,

$$F_n(\omega)\cdot F_n(\omega^{-1}) = nE_n$$

Proof. The ij^{th} element of $F_n(\omega)F_n(\omega^{-1})$ is

$$\sum_{k=0}^{n-1}\omega^{ik}\omega^{-ik} = \sum_{k=0}^{n-1}\omega^{k(i-j)} = \begin{cases} 0, & \text{if } i \neq j \\ n, & \text{otherwise} \end{cases}$$

The *i*=*j* case is obvious.If *i*≠*j* then ω^{i-j} will be a primitive root of unity of order *l*, where *l*|*n*.Applying the previous proposition completes the proof.

So,

$$F_n^{-1}(\omega) = \frac{1}{n}F_n(\omega^{-1})$$

Evaluating	**y**= $F_n(\omega)$ **a**
Interpolation	**a**= $\frac{1}{n}F_n(\omega^{-1})$ **y**

FFT and iFFT are O(n log n)

By using the FFT and the iFFT,
we can transform a polynomial of degree *n*
back and forth between its coefficient
representation and a point-value representation
in time O(*n* ***log*** *n*).

The convolution

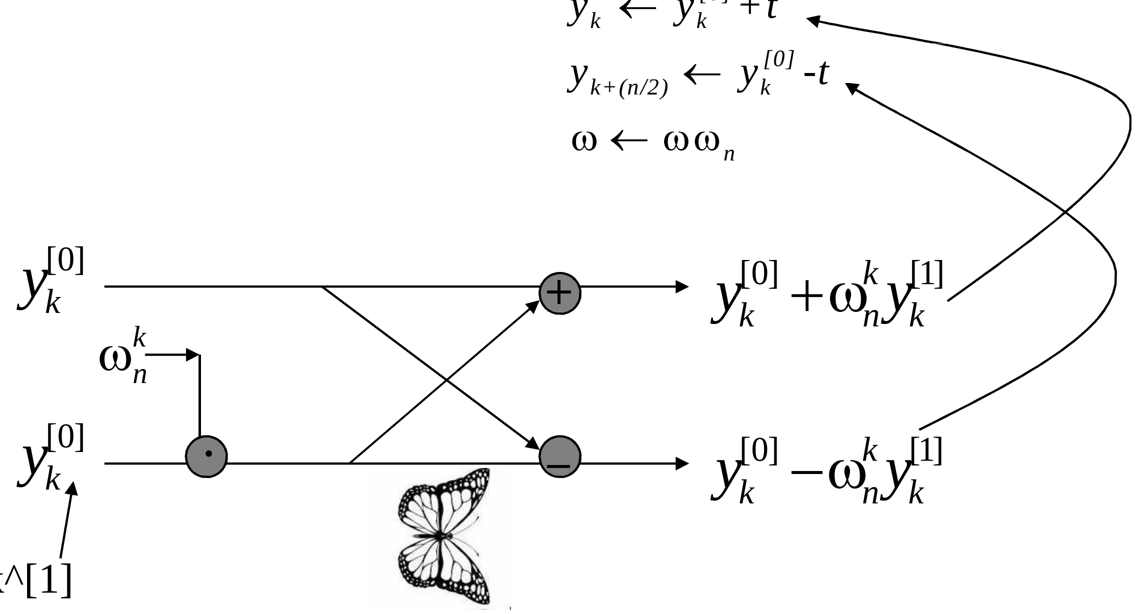
For any two vectors ***a*** and ***b*** of length *n* is a power of 2, we can do:

$$\mathbf{a} \otimes \mathbf{b} = DFT_{2n}^{-1}(DFT_{2n}(\mathbf{a}) \cdot DFT_{2n}(\mathbf{b}))$$

More effective implementations

The **for** loop involves computing the value $\omega_n^k y_k^{[1]}$ twice.
We can change the loop (the butterfly operation):

```
for  $k \leftarrow 0$ to $n/2-1$
    do $t \leftarrow \omega y_k^{[1]}$
        $y_k \leftarrow y_k^{[0]} + t$
        $y_{k+(n/2)} \leftarrow y_k^{[0]} - t$
        $\omega \leftarrow \omega\omega_n$
```

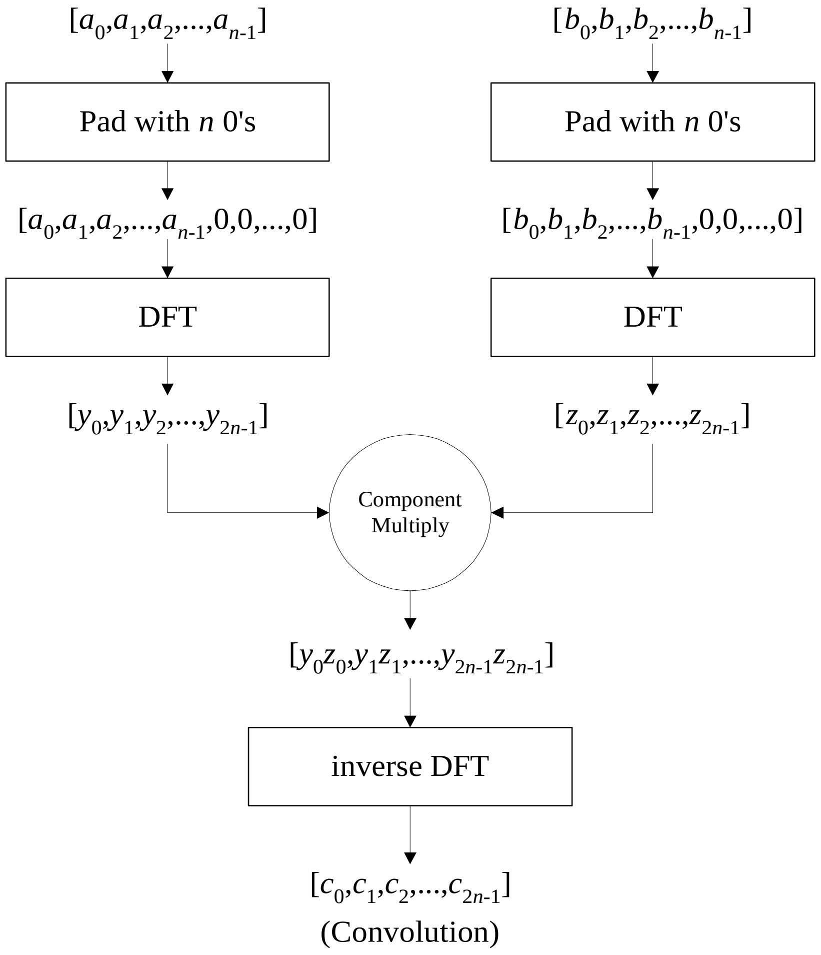


$y_k^{[0]}$
ω_n^k
$y_k^{[0]}$
$+$
$-$
$\cdot$
$y_k^{[0]} + \omega_n^k y_k^{[1]}$
$y_k^{[0]} - \omega_n^k y_k^{[1]}$

Typo: y_k^[1]

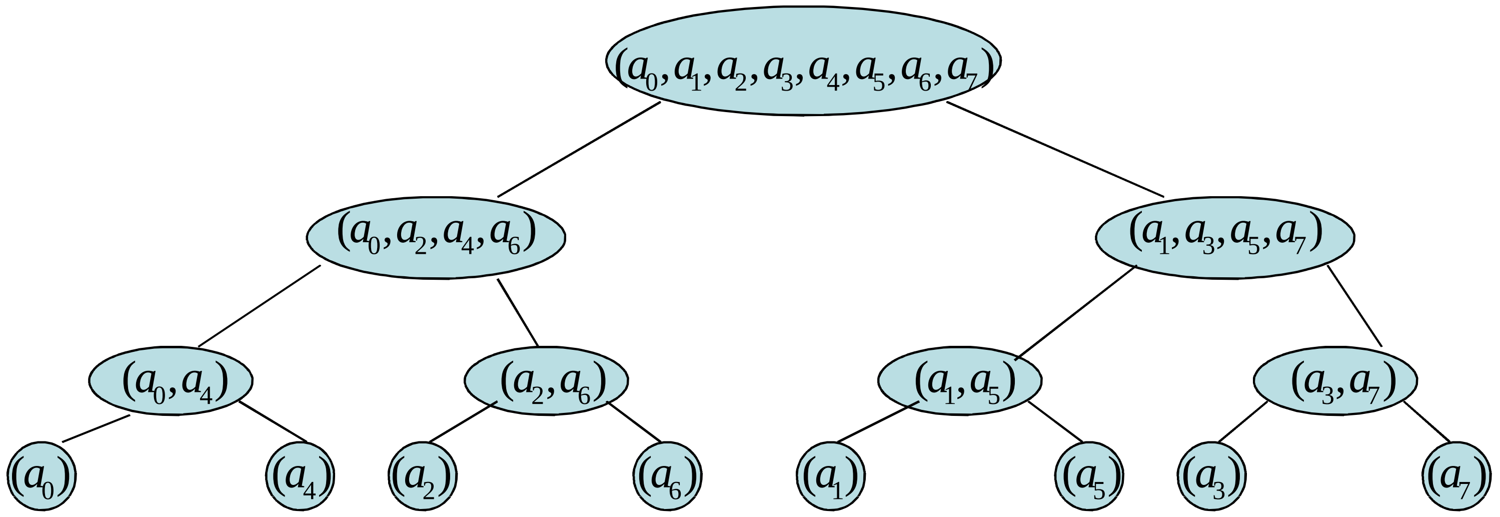
Convolution

- The DFT and the iDFT can be used to multiply two polynomials
- So we can get the coefficients of the product polynomial quickly if we can compute the DFT (and iDFT) quickly



$[a_0,a_1,a_2,...,a_{n-1}]$
$[b_0,b_1,b_2,...,b_{n-1}]$
Pad with n 0's
Pad with n 0's
$[a_0,a_1,a_2,...,a_{n-1},0,0,...,0]$
$[b_0,b_1,b_2,...,b_{n-1},0,0,...,0]$
DFT
DFT
$[y_0,y_1,y_2,...,y_{2n-1}]$
$[z_0,z_1,z_2,...,z_{2n-1}]$
Component Multiply
$[y_0z_0,y_1z_1,...,y_{2n-1}z_{2n-1}]$
inverse DFT
$[c_0,c_1,c_2,...,c_{2n-1}]$
(Convolution)

Iterative FFT



We take the elements in pairs, compute the DFT of each pair, using one butterfly operation, and replace the pair with its DFT

We take these n/2 DFT's in pairs and compute the DFT of the four vector elements.

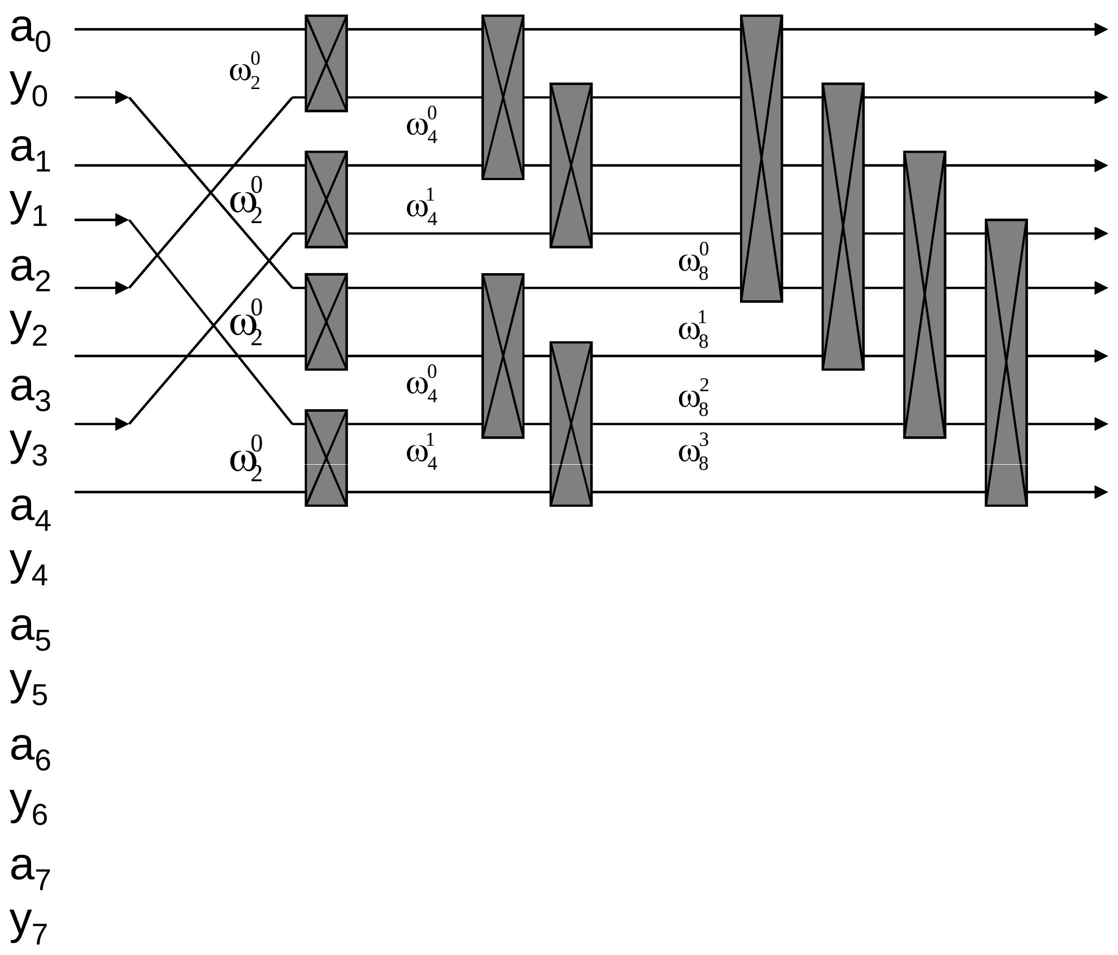
We take 2 (n/2)-element DFT's and combine them using n/2 butterfly operations into the final n-element DFT

Iterative-FFT Code with bit reversal.

0,4,2,6,1,5,3,7 → 000,100,010,110,001,101,011,111 → 000,001,010,011,100,101,110,111

```
BIT-REVERSE-COPY(a,A)
n←*length* [a]
**for** k←0 **to** n-1
        **do** A[rev(k)]←$a_k$
ITERATIVE-FFT
1.      BIT-REVERSE-COPY(a,A)
2.      n←*length* [a]
3.      **for** s←1 to log n
4.              **do** m←$2^s$
5.                  $\omega_m$ ←$e^{2\pi i/m}$
6.       **for** j←0 **to** n-1 **by** m ←1
7.                      **for** j←0 **to** m/2-1
8.                          **do for**  k←j **to** n-1 **by** m
9.                                  **do** t← $\omega$A[k+m/2]
10.                                     u←A[k]
11.                                     A[k]←u+t
12.                                     A[k+m/2]←u-t
13.     $\omega$← $\omega$ $\omega_m$
14.     **return A**
```

A parallel FFT circuit



FFT Example

Example: Q=A*B

Usual multiplication the following polynomials in O(n^2):

A(x):=1+x+2x^2;

B(x):=1+2x+3x^2;

Q(x):=A(x)*B(x);

=1+3x+7x^2+7x^3+6x^4

Fast Polynomial Multiplication

Multiply the polynomials in O(n log(n))

using *DFT* of the coefficient vectors:

```
A=(1,1,2,0,0)
B=(2,1,3,0,0)
DFT(A)=
 [4.000,  (-0.309 - 2.126i), (0.809 + 1.314i),
   (0.809 - 1.314i), ( -0.309+ 2.126i)]
DFT(B)=
  [6.000,  (-0.809 - 3.665i),  (0.309 + 1.677i),
   (0.309 -1.677i),  (-0.809 + 3.665i)]
```

A x B = iDFT(DFT(A)*DFT(B))

DFT(A) · DFT(B) =

[24.00, (-7.545 + 2.853*i*),

(-1.954 + 1.763*i*), (-1.954 - 1.763*i*),

(-7.545 - 2.853*i*)]

and

A x B = iDFT (DFT(A) * DFT(B)) = (1,3,7,7,6).

FFT speedup

If one complex multiplication takes 500ns (.5us):

N	T_{DFT}	T_{FFT}
2^{12}	8 sec.	0.013 sec.
2^{16}	0.6 hours	0.26 sec.
2^{20}	6 days	5 sec.

FFT in Maple

```
# multiply polynomials A and B using FFT
A(x):=1+x+2*x^2;
B(x):=1+2*x+3*x^2;
Q(x):=expand(A(x)*B(x));
readlib(FFT);
ar := array([1,1,2,0,0,0,0,0]); ai := array([0,0,0,0,0,0,0,0]);
br := array([1,2,3,0,0,0,0,0]); bi := array([0,0,0,0,0,0,0,0]);
FFT(3,ar,ai); af := evalm(ar+I*ai);
FFT(3,br,bi); bf := evalm(br+I*bi);
abf := zip( (x,y)->x * y, af, bf );
abfr := evalm(map(x->Re(x), abf));
abfi := evalm(map(x->Im(x), abf));
iFFT(3,abfr, abfi);
evalm(abfr+I*abfi);
# coeffs of Q := [1,3,7,7,6]
```

FFT Examples

With numpy in python

- #!python2.5
- # What: fft in python using numpy.

- from numpy.fft import fft
- from numpy.fft import ifft
- from numpy import array

- from numpy import set_printoptions, get_printoptions
- # print get_printoptions()
- set_printoptions(precision = 4)
- set_printoptions(suppress=True)

- a = array((0, 1, 7, 2, -1, 3, 7, 8, 0, -23, -7, 31, 1, 31, -7, -31))

- print "data =", a
- y = fft(a)
- print "fft(data) = ", y
- z = ifft(y)
- print "ifft(fft(data)) = ", z

Running fft-numpy.py

```
•   > fft-numpy.py
•   data = [  0   1   7   2  -1   3   7   8
•             0 -23  -7  31   1  31  -7 -31]

•   fft(data) =  [
•     22  +0j     -14.2 +10.7j -79.1  +0j      -4.8-101.8j
•      0  -2.j       4.8 -58.2j  79.1  -0j      14.2 +46.3j
•    -22  +0j      14.2 -46.3j  79.1  -0j       4.8 +58.2j
•      0  +2.j      -4.8+101.8j -79.1  +0j     -14.2 -10.7j]

•   ifft(fft(data)) =  [
•      0+0j   1-0j   7-0j   2-0j  -1+0j   3+0j   7-0j   8+0j
•      0+0j -23-0j  -7+0j  31+0j   1+0j  31+0j  -7+0j -31+0j]
```

fft.c 1 of 2.

```
•   double PI = 3.14159265358979;
•   const double eps = 1.e-7;
•   typedef double complex cplx;

•   void fft_rec(cplx buf[], cplx out[], int n, int step, int inv) {
•     if (step < n) {
•       int i;
•       fft_rec(out       ,     buf          , n, step * 2, inv);
•       fft_rec(out + step, buf + step, n, step * 2, inv);
•       for (i = 0; i < n; i += 2 * step) {
•         int sign = inv? -1 : 1;
•         cplx t = cexp(-I * PI * i * sign / n) * out[i + step];
•         buf[ i / 2]       = out[i] + t;
•         buf[(i + n)/2] = out[i] - t;
•       }
•     }
•   }
```

fft.c 2 of 2.

```c
void fft(cplx buf[], int n, int inv) {
  cplx out[n];
  int i;
  assert(is_power_of_two(n));
  for (i = 0; i < n; i++)
    out[i] = buf[i];
  fft_rec(buf, out, n, 1, inv);
  if (inv) for (i = 0; i < n; i++) buf[i] /= n;
}

int main() {
  cplx buf[] = {1, 2, 5, 1, 0, 0, 0, 99};
  fft(buf, 8, 0);
  fft(buf, 8, 1); // invfft
}
```

Running fft.c

```
> gcc -std=gnu99 -g -o fft.exe fft.c
> ./fft.exe
•  Data: x[0]=(1, 0), x[1]=(2, 0), x[2]=(5, 0), x[3]=(1, 0),
•      x[4]=(0, 0), x[5]=(0, 0), x[6]=(0, 0), x[7]=(99, 0),

•  FFT: x[0]=(108, 0), x[1]=(71.71, 62.88),
•      x[2]=(-4, 98),  x[3]=(-69.71, 72.88),
•      x[4]=(-96, 0),  x[5]=(-69.71, -72.88),
•      x[6]=(-4, -98), x[7]=(71.71, -62.88),

•  iFFT: x[0]=(1, 0), x[1]=(2, 0), x[2]=(5, 0), x[3]=(1, 0),
•      x[4]=(0, 0), x[5]=(0, 0), x[6]=(0, 0), x[7]=(99, 0),
```

Integers are also polynomials

- Cryptography involves multiplying large (100s of digits) integers. So it must be done efficiently.

- E.g. 2345 = 2x^3+3x^2+4x+5 (x=10)
- E.g. 0x45f = 4x^2 + 5x + f   (x=16, hex).

Now we can use fast polynomial multiplication algorithm for integers also.

Example: Compute E to 100,000 decimals using fft-mul.c in 1 second

- > e2718.exe 100000 | head
- Total Allocated memory = 4608 K
- Starting series computation
- Starting final division
- Total time : 1.01 seconds
- Worst error in FFT (should be less than 0.25): 0.0003662109
- E = 2.7182818284 5904523536 0287471352 …

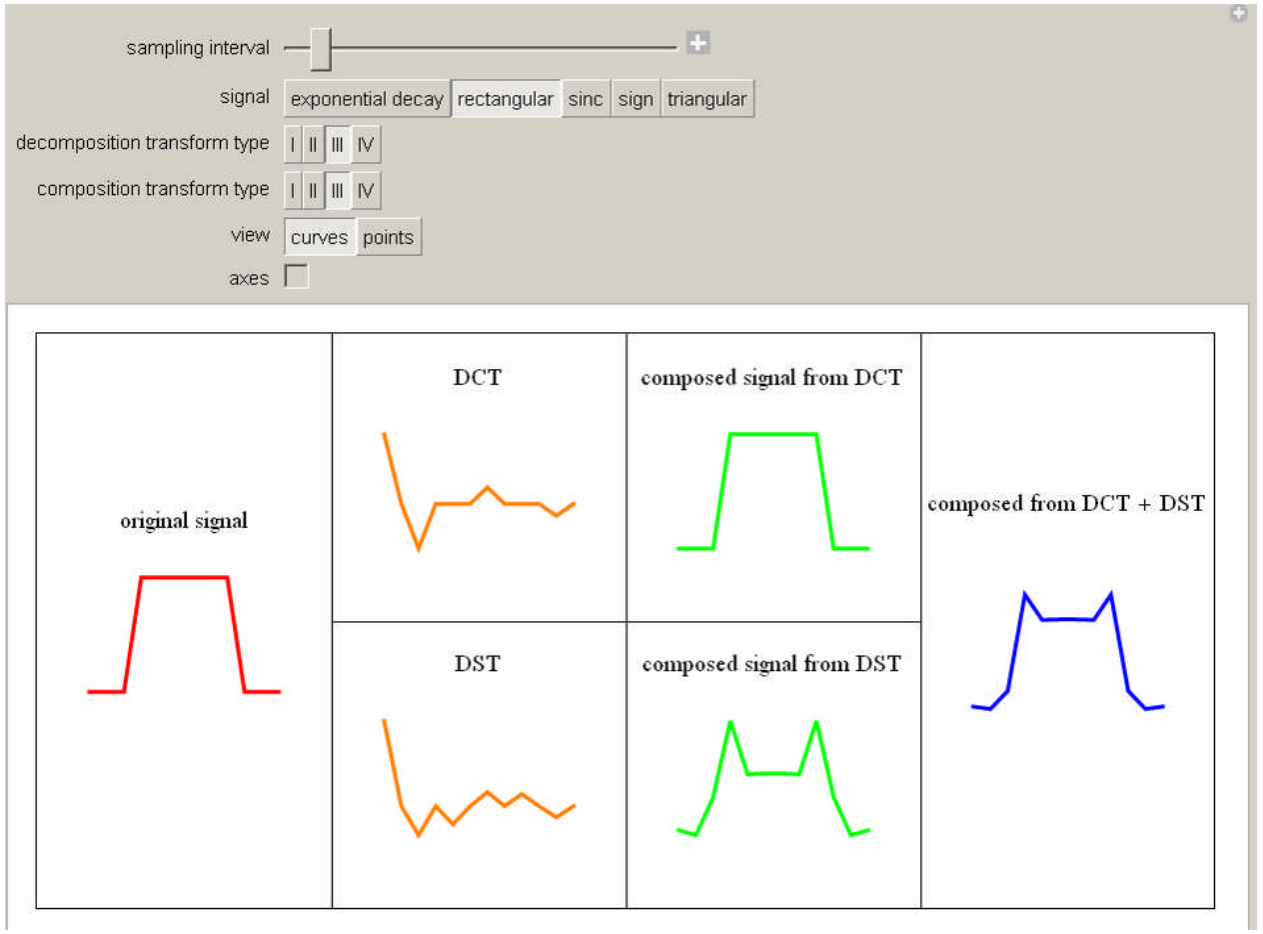
from http://xavier.gourdon.free.fr/Constants/constants.html

FFT

Mathematica

demonstrations

Mathematica demonstration of DCT/DST



sampling interval

signal: exponential decay [✓ rectangular] sinc sign triangular

decomposition transform type: I II [✓ III] IV

composition transform type: I II [✓ III] IV

view: [✓ curves] points

axes ☐

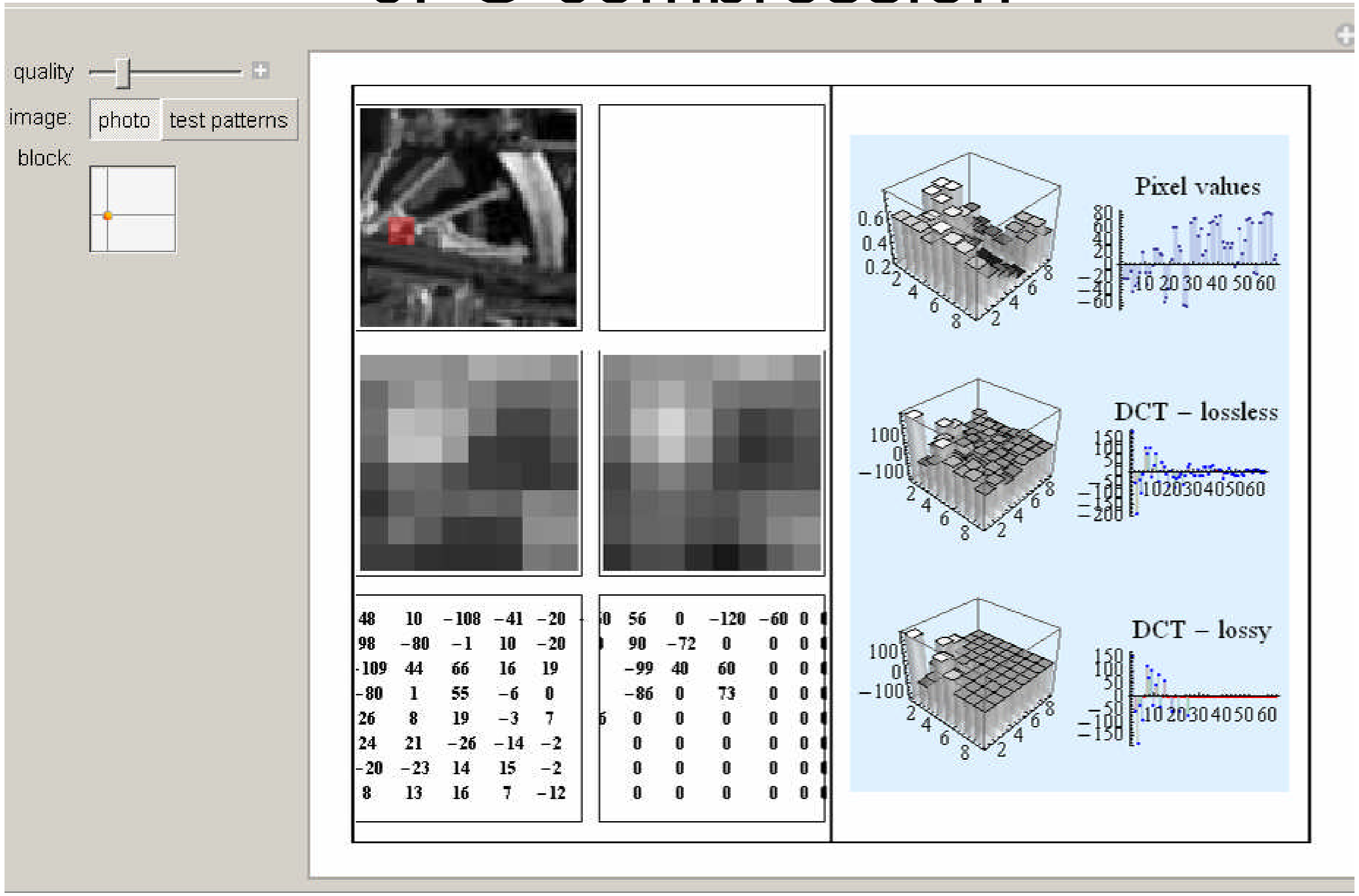
2D FT

- 2 dimensional Fourier transforms simply involve a number of 1 dimensional fourier transforms.
- More precisely, a 2 dimensional transform is achieved by first transforming each row, replacing each row with its transform and then transforming each column, replacing each column with its transform.
- From http://paulbourke.net/miscellaneous/dft/

Images are also integers

- The JPEG compression algorithm (which is also used in MPEG compression) is based on the two-dimensional discrete cosine transform (DCT) applied to image blocks that are 8×8 pixels in size.
- DCT concentrates information about the pixels in the top-left corner of the 8×8 matrix so that the importance of information in the direction of the bottom-right corner decreases.
- It is then possible to degrade the low information value coefficients by dividing and multiplying them by the so-called quantization matrix.
- These operations are rounded to whole numbers, so the smallest values become zero and can be deleted to reduce the size of the JPEG.
- From http://demonstrations.wolfram.com/JPEGCompressionAlgorithm/

Mathematica Demonstration of JPG compression



quality

image: photo | test patterns

block:

48	10	−108	−41	−20
98	−80	−1	10	−20
-109	44	66	16	19
−80	1	55	−6	0
26	8	19	−3	7
24	21	−26	−14	−2
−20	−23	14	15	−2
8	13	16	7	−12

56	0	−120	−60	0
90	−72	0	0	0
−99	40	60	0	0
−86	0	73	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

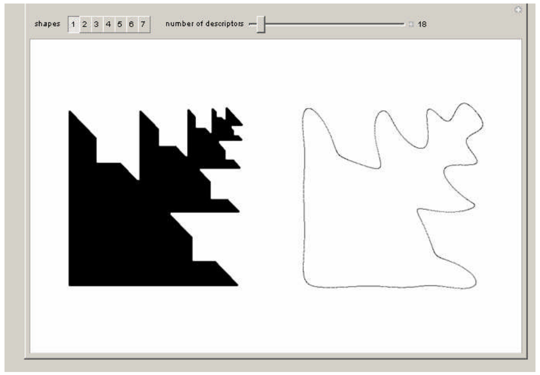
Fourier descriptors for shape approximation

Fourier descriptors are a way of encoding the shape of a two-dimensional object by taking the Fourier transform of the boundary, where every (x,y) point on the boundary is mapped to a complex number x+i y.

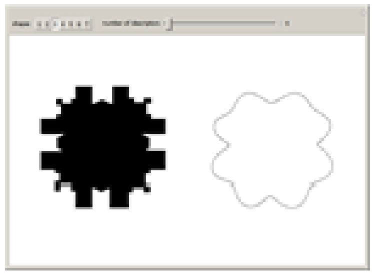
The original shape can be recovered from the inverse Fourier transform.

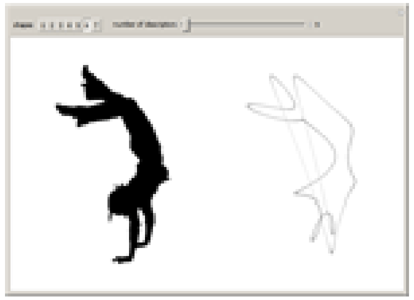
However, if only a few terms of the inverse are used, the boundary becomes simplified, providing a way to smooth or filter the boundary.

From http://demonstrations.wolfram.com/FourierDescriptors/



shapes 1 2 3 4 5 6 7 number of descriptors 18





Fourier descriptors details

The Fourier descriptors of a shape are calculated as follows.

- 1. Find the coordinates of the edge pixels of a shape and put them in a list in order, going clockwise around the shape.
  2. Define a complex-valued vector using the coordinates obtained. For example: .
  3. Take the discrete Fourier transform of the complex-valued vector.

Fourier descriptors inherit several properties from the Fourier transform.

- 4. Translation invariance: no matter where the shape is located in the image, the Fourier descriptors remain the same.
  5. Scaling: if the shape is scaled by a factor, the Fourier descriptors are scaled by that same factor.
  6. Rotation and starting point: rotating the shape or selecting a different starting point only affects the phase of the descriptors.

Because the discrete Fourier transform is invertible, all the information about the shape is contained in the Fourier descriptors. A common thing to do with Fourier descriptors is to set the descriptors corresponding to values above a certain frequency to zero and then reconstruct the shape. The effect of this is a low-pass filtering of the shape, smoothing the boundary. Since many shapes can be approximated with a small number of parameters, Fourier descriptors are commonly used to classify shapes.

- The slider lets you choose how many terms to use in the reconstruction. With more terms, the shape looks more like the original. With fewer terms, the shape becomes smoother and rounder.
- The basic method of Fourier descriptors is discussed in R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, Englewood Cliffs, NJ: Prentice Hall, 2007.