Advanced Numerical Algorithms (MU4IN920)

Lecture 3: Fast Fourier Transform

Stef Graillat

Sorbonne Université

Summary of the previous lecture

1. Solving linear systems by iterative methods (Jacobi, Gauss-Seidel, SOR)
2. Conjugate gradient method
3. Krylov subspace methods
4. Efficient algorithms in practice, especially for the conjugate gradient algorithm
5. Algorithms mostly used for sparse matrices

Objectives

To present the Fast Fourier Transform (FFT) algorithm

One of the most used algorithms in the world with applications in

- signal processing
- image processing
- computer algebra (multiplication of polynomials, large integers, etc.)

Course outline

1. Multiplication of polynomials and choice of representation
2. Evaluation and interpolation
3. n-th roots of unity
4. Matrix version of the FFT

References

- Algorithms, S. Dasgupta, C.H. Papadimitriou and U.V. Vazirani, McGraw Hill, 2006
- Introduction to Algorithms, Thomas Cormen, Charles Leiserson, Ronald Rivest and Clifford Stein, 4th edition, The MIT Press, 2022
- The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Donald E. Knuth, 3rd edition, Addison-Wesley, 1997
- Modern Computer Algebra, Joachim von zur Gathen and Jürgen Gerhard, 3rd edition, Cambridge University Press, 2013
- Numerical Recipes. The Art of Scientific Computing, William Press, Saul Teukolsky, William Vetterling and Brian Flannery, 3rd edition, Cambridge University Press, 2007
- Computational Frameworks for the Fast Fourier Transform, Charles Van Loan, SIAM, 1987

Multiplication of polynomials

The product of 2 polynomials of degree n is a polynomial of degree at most $2n$.

$$(1 + 2x + 3x^2) \cdot (2 + x + 4x^2) = 2 + 5x + 12x^2 + 11x^3 + 12x^4$$

More generally, if

$$P(x) = a_0 + a_1 x + \cdots + a_n x^n \quad \text{and} \quad Q(x) = b_0 + b_1 x + \cdots + b_n x^n$$

then $R(x) = P(x)Q(x) = c_0 + c_1 x + \cdots + c_{2n} x^{2n}$ with

$$c_k = a_0 b_k + a_1 b_{k-1} + \cdots + a_k b_0 = \sum_{i=0}^{k} a_i b_{k-i}$$

Multiplication of polynomials (cont'd)

Algorithm 1 (Naive multiplication)

```
function R = mult(P,Q)
    n = length(P);
    R = zeros(1,2*n-1);
    for i=1:n
        for j=1:n
            R(i+j-1) = R(i+j-1) + P(i)*Q(j);
        end
    end
```

Cost : $\mathcal{O}(n^2)$

Can we do better than $\mathcal{O}(n^2)$?
$\rightarrow$ Karatsuba's algorithm : $\mathcal{O}(n^{\log_2 3}) \approx \mathcal{O}(n^{1.58})$

Can we do any better?

Change of representation (continued)

We generally represent the polynomial

$$P(x) = a_0 + a_1 x + \cdots + a_n x^n$$

by the vector of its coefficients $a = (a_0, a_1, \ldots, a_{n-1})$

Question: are there other representations of polynomials?

Definition 1

A complex number z is a root of $P(x)$ if $P(z) = 0$

Theorem 1 (D'Alembert-Gauss's theorem)

Any polynomial $P(x)$ of degree n with coefficients in C has n roots $z_1, \ldots, z_n$ (counted with their multiplicities). Then we have

$$P(x) = a_n (x - z_1) \cdots (x - z_n)$$

Change of representation (continued)

So we can represent the polynomial $P(x)$ by a_n and its roots $z_1, \ldots, z_n$.

Evaluation: P being given by a_n and $z_1, \ldots, z_n$, we can can evaluate P in x by $\mathcal{O}(n)$.

Multiplication: P being given by a_n and $z_1, \ldots, z_n$ and Q being given by b_n and $z'_1, \ldots, z'_n$, PQ is given by $a_n b_n$, $z_1, \ldots, z_n, z'_1, \ldots, z'_n$, PQ is given by $a_n b_n$, $z_1, \ldots, z_n, z'_1, \ldots, z'_n$.

Addition: difficult!

Change of representation (continued)

Theorem 2

A polynomial of degree n is only determined by its values on $n+1$ distinct points.

Let us fix $n+1$ distinct points $x_0, \dots, x_n$. We can specify a polynomial $P(x) = a_0 + a_1 x + \cdots + a_n x^n$ of degree n by one of the following ways:

1. its coefficients $a_0, a_1, \dots, a_n$
2. the values $P(x_0), P(x_1), \dots, P(x_n)$

It is easy to multiply two polynomials whose values are known. The product $R(z)$ at a point z is the product of $P(z)$ by $Q(z)$!

Easy for the addition too!

The multiplication algorithm

$$\begin{tikzcd} \boxed{\begin{matrix}\text{Representation by coefficients} \\ a_0, a_1, \dots, a_n\end{matrix}} \arrow[r, shift left, "\text{Evaluation}"] & \boxed{\begin{matrix}\text{Representation by value} \\ P(x_0), P(x_1), \dots, P(x_n)\end{matrix}} \arrow[l, shift left, "\text{Interpolation}"] \end{tikzcd}$$

Algorithm 2 (Mutliplication of polynomials)

Enter: *2 polynomials P and Q of degree n given via their coefficients*
Exit: *the polynomial $R = PQ$ given via its coefficients*

Selection*Output Choose points* $x_0, x_2, \dots, x_{m-1}$ *with* $m \geq 2n + 1$

Evaluation *Calculate* $P(x_0), P(x_1), \dots, P(x_{m-1})$ *and* $Q(x_0), Q(x_1), \dots, Q(x_{m-1})$

Multiplication
Compute $R(x_k) = P(x_k)Q(x_k)$ *for all* $k = 0, \dots, m-1$

Interpolation
Find $R(x) = c_0 + c_1 x + \cdots + c_{2n} x^{2n}$

Cost of the algorithm

- Cost of the selection: $\mathcal{O}(n)$
- Cost of the multiplication: $\mathcal{O}(n)$

- What about the cost of evaluation and interpolation?

$$P(x) = a_0 + a_1 x + \cdots + a_n x^n$$

We have:

$$\begin{pmatrix} 1 & x_0 & \dots & x_0^{n-1} \\ 1 & x_1 & & x_1^{n-1} \\ \vdots & & & \vdots \\ 1 & x_{n-1} & \dots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} p(x_0) \\ p(x_1) \\ \vdots \\ p(x_{n-1})s \end{pmatrix}$$

Evaluation by Horner's algorithm

$$P(x) = a_0 + a_1 x + \cdots + a_n x^n = ((((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})x + \cdots)x + a_0$$

Algorithm 3 (Horner's algorithm)

```
function res = Horner(P, x)
    s_n = a_n
    for i = n-1 : -1 : 0
        s_i = s_{i+1} * x + a_i
    end
    res = s_0
```

Cost: $\mathcal{O}(n)$. If n evaluations to be made $\rightarrow$ $\mathcal{O}(n^2)$

Lagrange interpolation

$$P(x) = \sum_{j=0}^{n} P(x_j) \left(\prod_{i=0, i\neq j}^{n} \frac{x - x_j}{x_j - x_i} \right)$$

Algorithm 4 (Lagrange interpolation algorithm)

```
function P = interp([(x_i, y_i)], x)
A=1 and P=0
for i=0:n-1
    A = A(x - x_i)
end
for i=0:n-1
    A_i = A/(x - x_i)
    q_i = A(x_i)
    P = P + y_i A_i / q_i
end
```

Cost: $\mathcal{O}(n^2)$.

Multiplication in $\mathcal{O}(n^2)$ operations by evaluations and interpolation.

Evaluation in divide and conquer mode

We assume from now on that n is a power of 2 ($n = 2^k$ with $k \in \mathbb{N}$)

Idea: evaluate a polynomial $P(x)$ of degree $< n$ in n points by pair

$$\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1}$$

because we can overlap the computations of $P(x_i)$ and $P(-x_i)$.

Indeed, we can separate the even and odd powers,

$$P(x) = P_p(x^2) + xP_i(x^2)$$

Example:

$$3 + 4x + 6x^2 + x^4 + 10x^5 = (3 + 6x^2 + x^4) + x(4 + 2x^2 + 10x^4)$$

Evaluation in divide and conquer mode (cont'd)

We then notice that:

$$\begin{aligned} P(x_i) &= P_p(x_i^2) + x_i P_i(x_i^2) \\ P(-x_i) &= P_p(x_i^2) - x_i P_i(x_i^2) \end{aligned}$$

Therefore, to evaluate $P(x)$ in n points $\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1}$, it is enough to evaluate $P_p(x)$ and $P_i(x)$ in $n/2$ points $x_0^2, \ldots, x_{n/2-1}^2$

If we continue the process in a recursive manner, the number $T(n)$ of arithmetic operations arithmetic operations verifies

$$T(n) = 2T(n/2) + \mathcal{O}(n)$$

We show then that

$$T(n) = \mathcal{O}(n \log n)$$

Evaluation in divide and conquer mode: towards recursion

- We start from n points $\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1}$
- After one step, we have $n/2$ points $x_0^2, \ldots, x_{n/2-1}^2$
- To continue the recursion, we need

$$\{x_0^2, \ldots, x_{n/2-1}^2\} = \{\pm z_0, \pm z_1, \ldots, \pm z_{n/4-1}\}$$

- If $z_0 = x_0^2$ and $-z_0 = x_j^2$ then $x_0 = \pm i x_j$ with $i^2 = -1$
- To continue the recursion, we need the complex numbers!

n-th roots of unity

These are the solutions in $\mathbb{C}$ of the equation $z^n = 1$!

Theorem 3

Let $\omega = e^{2i\pi/n}$. The solutions of the equation $z^n = 1$ are $\omega^j = e^{2ij\pi/n}$ for $j = 0, 1, \ldots, n-1$.

Property 1

If n is even then

- *the n-th roots of unity are even: $\omega^{n/2+j} = -\omega^j$*
- *squaring them produces the $n/2$th roots of the unit*

Evaluation in n-th roots of unity

Let's note $\omega_n = e^{2i\pi/n}$

- Problem: evaluate P in the n-th roots of the unit
- Recall that
$$P(x) = P_p(x^2) + xP_i(x^2)$$
- We observe that $(\omega_n^j)^2 = \omega_{n/2}^j$
- Therefore
$$P(\omega_n^j) = P_p((\omega_n^j)^2) + xP_i((\omega_n^j)^2)$$
- Thus evaluating P in n-th roots of the unit can be done by evaluating P_p and P_i in the $n/2$-th roots of the unit

Evaluation in n-th roots of unity (cont'd)

Algorithm 5 (Calculation of $FFT(P, \omega)$)

Input: *the polynomial P known by its coefficients of degree n with n a power of 2 and ω a primitive root n of unity*
Output: *the values of $P(\omega^0), P(\omega^1), P(\omega^{n-1})$*

- *if* $\omega = 1$ *then return* $P(1)$
- *express* $P(x)$ *in the form* $P(x) = P_p(x^2) + xP_i(x^2)$
- *call* $FFT(P_p, \omega^2)$ *to evaluate* P_p *in the even powers of* ω
- *call* $FFT(P_i, \omega^2)$ *to evaluate* P_i *in even powers of* ω
- *for* $j = 0 : n - 1$

  calculate $P(\omega^j) = P_p(\omega^{2j}) + \omega^j P_i(\omega^{2j})$
- *return* $P(\omega^0), P(\omega^1), \ldots, P(\omega^{n-1})$

Matrix formulation

The evaluation of $P(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$ in n points $x_0, x_1, \ldots, x_{n-1}$ can be written matrix-wise

$$\begin{pmatrix} P(x_0) \\ P(x_1) \\ \vdots \\ P(x_{n-1}) \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ & & \vdots & & \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix}}_{M} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

The matrix M is a matrix of Vandermonde.

Property 2

If $x_0, x_1, \ldots, x_{n-1}$ are distinct numbers then M is invertible.

The evaluation is the multiplication by M and the interpolation is multiplication by M^{-1}

Matrix formulation : evaluation in n-th roots

To evaluate in n-th roots is to multiply by

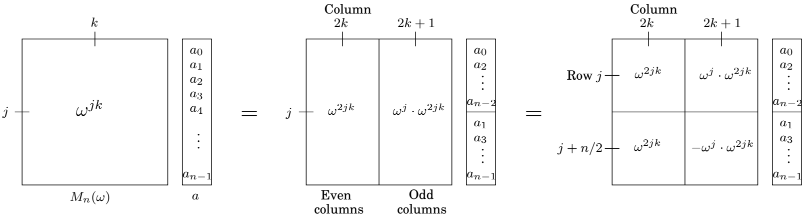
$$M_n(\omega) = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ & & \vdots & & \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{(n-1)j} \\ & & \vdots & & \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{pmatrix}$$

Theorem 4 (Inversion formula)

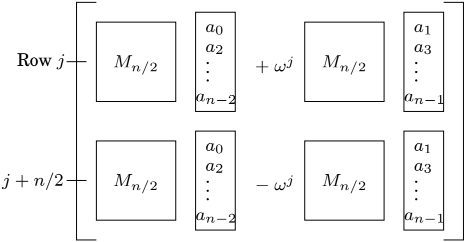
$$M_n(\omega)^{-1} = \frac{1}{n} M_n(\omega^{-1})$$

Algorithm divide and conquer matrix

The divide and conquer algorithm is seen in a matrix fashion



Which can also be seen as



FFT algorithm

Algorithm 6 (Calculation of $FFT(a, \omega)$)

Input: *an array* $a = (a_0, a_1, \ldots, a_{n-1})$ *with n a power of 2 and ω a primitive root n of the unit*

Output: $M_n(\omega)a$

$if\ \omega = 1\ then\ return\ a$

$(s_0, s_1, \ldots, s_{n/2-1}) = FFT((a_0, a_2, \ldots, a_{n-2}), \omega^2)$

$(s'_0, s'_1, \ldots, s'_{n/2-1}) = FFT((a_1, a_3, \ldots, a_{n-1}), \omega^2)$

$for\ j = 0 : n/2 - 1$

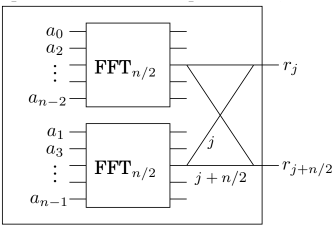
$\quad r_j = s_j + \omega^j s'_j$

$\quad r_{j+n/2} = s_j - \omega^j s'_j$

$return\ (r_0, r_1, \ldots, r_{n-1})$

FFT iterative version

The recursion step of the FFT algorithm can be represented by the circuit

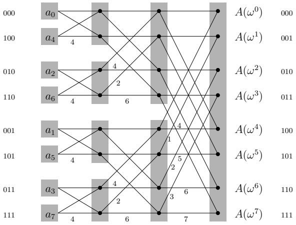


meaning that

$$\begin{aligned} r_j &= s_j + \omega^j s'_j \\ r_{j+n/2} &= s_j - \omega^j s'_j \end{aligned}$$

FFT iterative version (continued)

If we unroll the recursion for the circuit with $n = 8$, we obtain



- there are $\log_2(n)$ levels each with n nodes for a total of $n \log n$ operations
- the entries are arranged in a particular order : 0,4,2,6,1,5,3,7

$\rightarrow$ one notices that the entries are arranged in order of of the last bits of the binary representation of their indices!

The resulting order in binary is 000,100,010,110,001,101,011,111
it is the classical ascending order 000,001,010,011,100,101,110,111 but the bits are inverted

And elsewhere than in $\mathbb{C}$? Finite fields!

The ring $\mathbb{Z}/p\mathbb{Z}$ is a body if, and only if, p is prime. It is usually denoted $\mathbb{F}_p$.
For any p prime and $r > 1$, there exists a field with p^r elements noted $\mathbb{F}_{p^r}$. Be careful, it is not $\mathbb{Z}/p^r\mathbb{Z}$.
The group of inversibles of $\mathbb{F}_{p^r}$ is a cyclic group with $p^r - 1$ elements. It therefore admits the primitive roots $(p^r - 1)$-sixths of the unit.

If $p^r = 2^k \times M + 1$, then 1 has a primitive root 2^k-th of unity, we can do FFT on it!
Since $17 - 1 = 2^4$, there are 16-second primitive roots of unit in $\mathbb{F}_{17}$ (6 and 11). We can therefore multiply polynomials whose product is of degree at most 15 by FFT.
Since $7937 - 1 = 2^8 \times 31$, we can do FFT in $\mathbb{F}_{7937}$ to multiply polynomials whose product is of degree at most 255.

The inventors of the FFT

Gauss, Carl Friedrich, "Nachlass: Theoria interpolationis methodo nova tractata", Werke, Band 3, 265-327 (Königliche Gesellschaft der Wissenschaften, Göttingen, 1866)

Johann Carl Friedrich Gauss (1777-1855)

The inventors of the FFT (continued)

James W. Cooley, and John W. Tukey, 'An algorithm for the machine calculation of complex Fourier series', Math. Comput. 19, 297-301 (1965)

James Cooley (1926-)

John Tukey (1915-2000)

Software tools

The FFT is implemented in many tools for scientific computing.

- in MATLAB, you must use the command `fft` for the FFT and the command `ifft` for the inverse Fourier transform
- there is a very efficient code in C to calculate FFTs : FFTW (`http://www.fftw.org/`)

Main complexity of multiplication algorithms

name	complexity in $\mathbb{Z}$	complexity in $\mathbb{K}[x]$
naive	$\mathcal{O}(n^2)$	
Karatsuba	$\mathcal{O}(n^{\log_2 3})$	
Toom-Cook	$\mathcal{O}(n^{2\log(2k-1)/\log k}),\ k \geq 2$	
Schönhage – Strassen Cantor – Kaltofen	$\mathcal{O}(n \log n \log\log n)$ 	 $\mathcal{O}(n \log n \log\log n)$
Fürer Harvey *et al.* Harvey - van der Hoeven	$\mathcal{O}(n \log n\, 2^{\Theta(\log^* n)})$ $\mathcal{O}(n \log n\, 8^{\log^* n})$ $\mathcal{O}(n \log n)$	 $O(n \log n\, 8^{\log^* n}),\ \mathbb{K} = \mathbb{F}_p$
Cooley – Tukey (FFT)		$\mathcal{O}(n \log n)$

Conclusion

- a very efficient algorithm (in $\mathcal{O}(n \log n)$)
- it has applications in many fields (imaging, signal processing, formal calculation, number theory, etc.)
- there are other algorithms than the Cooley-Tukey one
- if the data are real, a discrete cosine transform is used instead