Mathematical Tripos Part II: Michaelmas Term 2020

Numerical Analysis – Lecture 4

Algorithm 1.15 (The fast Fourier transform (FFT)) We assume that n is a power of 2, i.e. $n = 2m = 2^p$, and for $\boldsymbol{y} \in \Pi_{2m}$, denote by

$$\boldsymbol{y}^{(\mathrm{E})} = \{y_{2j}\}_{j\in\mathbb{Z}} \qquad \text{and} \qquad \boldsymbol{y}^{(\mathrm{O})} = \{y_{2j+1}\}_{j\in\mathbb{Z}}$$

the even and odd portions of $\boldsymbol{y}$, respectively. Note that $\boldsymbol{y}^{(\mathrm{E})}, \boldsymbol{y}^{(\mathrm{O})} \in \Pi_m$.

Suppose that we already know the inverse DFT of both 'short' sequences,

$$\boldsymbol{x}^{(\mathrm{E})} = \mathcal{F}_m^{-1}\boldsymbol{y}^{(\mathrm{E})}, \qquad \boldsymbol{x}^{(\mathrm{O})} = \mathcal{F}_m^{-1}\boldsymbol{y}^{(\mathrm{O})}.$$

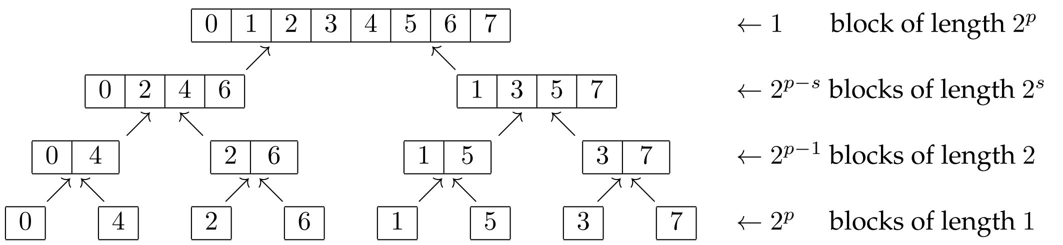
It is then possible to assemble $\boldsymbol{x} = \mathcal{F}_{2m}^{-1}\boldsymbol{y}$ in a small number of operations. Since $\omega_{2m}^{2m} = 1$, we obtain $\omega_{2m}^2 = \omega_m$, and

$$\begin{aligned} x_\ell &= \sum_{j=0}^{2m-1} \omega_{2m}^{j\ell} y_j &&= \sum_{j=0}^{m-1} \omega_{2m}^{2j\ell} y_{2j} + \sum_{j=0}^{m-1} \omega_{2m}^{(2j+1)\ell} y_{2j+1} \\ & &&= \sum_{j=0}^{m-1} \omega_m^{j\ell} y_j^{(\mathrm{E})} + \omega_{2m}^\ell \sum_{j=0}^{m-1} \omega_m^{j\ell} y_j^{(\mathrm{O})} = x_\ell^{(\mathrm{E})} + \omega_{2m}^\ell x_\ell^{(\mathrm{O})}, \qquad \ell = 0, \dots, m-1. \end{aligned}$$

Therefore, it costs just m products to evaluate the first half of $\boldsymbol{x}$, provided that $\boldsymbol{x}^{(\mathrm{E})}$ and $\boldsymbol{x}^{(\mathrm{O})}$ are known. It actually costs nothing to evaluate the second half, since

$$\omega_m^{j(m+\ell)} = \omega_m^{j\ell}, \qquad \omega_{2m}^{m+\ell} = -\omega_{2m}^\ell \qquad \Rightarrow \qquad x_{m+\ell} = x_\ell^{(\mathrm{E})} - \omega_{2m}^\ell x_\ell^{(\mathrm{O})}, \qquad \ell = 0, \dots, m-1.$$

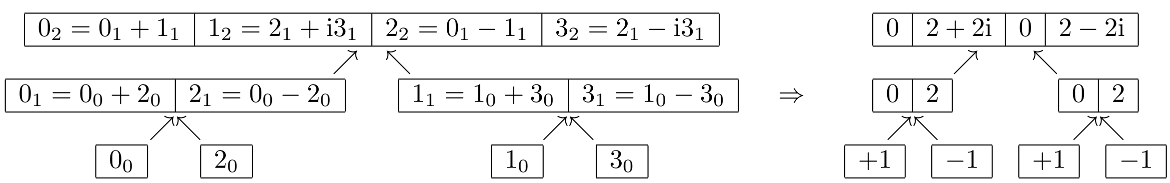
To execute FFT, we start from vectors of unit length and in each s-th stage, $s = 1...p$, assemble 2^{p-s} vectors of length 2^s from vectors of length 2^{s-1}: this costs $2^{p-s}2^{s-1} = 2^{p-1}$ products. Altogether, the cost of FFT is $p2^{p-1} = \frac{1}{2}n\log_2 n$ products.



For $n = 1024 = 2^{10}$, say, the cost is $\approx 5\times 10^3$ products, compared to $\approx 10^6$ for naive matrix multiplication! For $n = 2^{20}$ the respective numbers are $\approx 1.05\times 10^7$ and $\approx 1.1\times 10^{12}$, which represents a saving by a factor of more than 10^5.

Matlab demo: Check out the online animation for computing the FFT at `http://www.damtp.cam.ac.uk/user/naweb/ii/fft_gui/fft_gui.php` and download the Matlab GUI from there to follow the computation of each single FFT term.

Example 1.16 Computation of FFT for $n = 4$ in general, and for the vector $\boldsymbol{y} = (1, 1, -1, -1)$ in particular.



2 Partial differential equations of evolution

Method 2.1 We consider the solution of the *diffusion equation*

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}\,, \qquad 0 \le x \le 1, \quad t \ge 0,$$

with *initial conditions* $u(x,0) = u_0(x)$ for $t=0$ and Dirichlet *boundary conditions* $u(0,t) = \phi_0(t)$ at $x=0$ and $u(1,t) = \phi_1(t)$ at $x=1$. By Taylor's expansion

$$\begin{aligned} \tfrac{\partial u(x,t)}{\partial t} &= \tfrac{1}{k}\big[u(x,t+k) - u(x,t)\big] + \mathcal{O}(k), & k = \Delta t\,, \\ \tfrac{\partial^2 u(x,t)}{\partial x^2} &= \tfrac{1}{h^2}\big[u(x-h,t) - 2u(x,t) + u(x+h,t)\big] + \mathcal{O}(h^2), & h = \Delta x\,, \end{aligned}$$

so that, for the true solution, we obtain

$$u(x,t+k) = u(x,t) + \tfrac{k}{h^2}\big[u(x-h,t) - 2u(x,t) + u(x+h,t)\big] + \mathcal{O}(k^2 + kh^2)\,. \tag{2.1}$$

That motivates the numerical scheme for approximation $u_m^n \approx u(x_m,t_n)$ on the rectangular mesh $(x_m,t_n) = (mh,nk)$:

$$u_m^{n+1} = u_m^n + \mu\left(u_{m-1}^n - 2u_m^n + u_{m+1}^n\right), \qquad m = 1...M\,. \tag{2.2}$$

Here $h = \frac{1}{M+1}$ and $\mu = \frac{k}{h^2} = \frac{\Delta t}{(\Delta x)^2}$ is the so-called *Courant number.* With μ being fixed, we have $k = \mu h^2$, so that the local truncation error of the scheme is $\mathcal{O}(h^4)$. Substituting whenever necessary initial conditions u_m^0 and boundary conditions u_0^n and u_{M+1}^n, we possess enough information to advance in (2.2) from $\boldsymbol{u}^n := [u_1^n, \ldots, u_M^n]$ to $\boldsymbol{u}^{n+1} := [u_1^{n+1}, \ldots, u_M^{n+1}]$.

Similarly to ODEs or Poisson equation, we say that the method is *convergent* if, for a fixed μ, and for every $T > 0$, we have

$$\lim_{h\to 0} |u_m^n - u(x_m,t_n)| = 0 \quad \text{uniformly for} \quad (x_m,t_n) \in [0,1]\times[0,T]\,.$$

In the present case, however, a method has an extra parameter μ, and it is entirely possible for a method to converge for some choice of μ and diverge otherwise.

Theorem 2.2 *If $\mu \le \frac{1}{2}$, then method (2.2) converges.*

Proof. Let $e_m^n := u_m^n - u(mh,nk)$ be the error of approximation, and let $\boldsymbol{e}^n = [e_1^n, \ldots, e_M^n]$ with $\|\boldsymbol{e}^n\| := \max_m |e_m^n|$. Convergence is equivalent to

$$\lim_{h\to 0} \max_{1\le n\le T/k} \|\boldsymbol{e}^n\| = 0$$

for every constant $T > 0$. Subtracting (2.1) from (2.2), we obtain

$$\begin{aligned} e_m^{n+1} &= e_m^n + \mu(e_{m-1}^n - 2e_m^n + e_{m+1}^n) + \mathcal{O}(h^4) \\ &= \mu e_{m-1}^n + (1-2\mu)e_m^n + \mu e_{m+1}^n + \mathcal{O}(h^4). \end{aligned}$$

Then

$$\|\boldsymbol{e}^{n+1}\| = \max_m |e_m^{n+1}| \le (2\mu + |1-2\mu|)\,\|\boldsymbol{e}^n\| + ch^4 = \|e^n\| + ch^4,$$

by virtue of $\mu \le \frac{1}{2}$. Since $\|\boldsymbol{e}^0\| = 0$, induction yields

$$\|\boldsymbol{e}^n\| \le cnh^4 \le \tfrac{cT}{k}\,h^4 = \tfrac{cT}{\mu}\,h^2 \to 0 \qquad (h \to 0)$$

□

Discussion 2.3 In practice we wish to choose h and k of comparable size, therefore $\mu = k/h^2$ is likely to be large. Consequently, the restriction of the last theorem is disappointing: unless we are willing to advance with tiny time step k, the method (2.2) is of limited practical interest. The situation is similar to stiff ODEs: like the Euler method, the scheme (2.2) is simple, plausible, explicit, easy to execute and analyse – but of very limited utility....

Matlab demo: Download the Matlab GUI for *Stability of 1D PDEs* from `http://www.damtp.cam.ac.uk/user/naweb/ii/pde_stability/pde_stability.php` and solve the diffusion equation in the interval $[0,1]$ with method (2.2) and $\mu = 0.51 > \frac{1}{2}$. Using (as preset) 100 grid points to discretise $[0,1]$ will then require the time steps to be $5.1 \cdot 10^{-5}$. The solution will evolve very slowly, but wait long enough to see what happens!