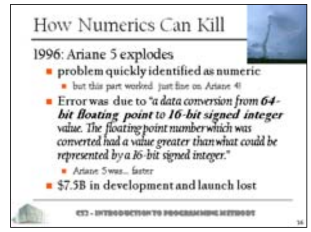
CS 2

Introduction to Programming Methods

Last Time

Numerics
- numbers are not as simple as it seems



Today's Lecture

Fourier transform

- seen from the signal processing viewpoint
  - lots of applications, from sound to images
    - editing, compression, ...
    - one of the most beloved and useful tools of our time
- and the polynomial viewpoint
  - to show that numerics can sometimes be done fast(er)

Polynomials

You all know about polynomials
- $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$
- or, more concisely: $p(x) = \sum_{i=0}^{n-1} a_i x^i$
- represented by vector $\boldsymbol{a}=(a_0,\ldots,a_{n-1})$ of coeffs

Addition of two polynomials?
- O(n) to find new coeffs, obviously

Evaluation?
- Horner scheme is optimal (n mults, n adds)

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

Product of Polynomials

Knowing two degree-n polynomials p and q

$$p(x) = \sum_{i=0}^{n-1} a_i x^i \qquad q(x) = \sum_{i=0}^{n-1} b_i x^i$$

compute coeffs of product $p(x)q(x) = \sum_{i=0}^{2(n-1)} c_i x^i$

- $c_k = \sum_{j=0}^{k} a_j b_{k-j} \quad \forall k \in [0, 2(n-1)]$ (convolution)
  - careful: indices out of bounds mean zero
- O(n^2), unfortunately...
  - unless you are clever about it
- notice that *evaluating* the product is trivial...

Transform

Coeffs not the only/best representation
- we saw that last time...

Idea: map the n coeffs to n *other* coeffs

$$\mathbf{a} = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} \longmapsto \hat{\mathbf{a}} = \begin{pmatrix} \hat{a}_0 \\ \hat{a}_1 \\ \vdots \\ \hat{a}_{n-1} \end{pmatrix}$$

- hopefully, this new representation is better...
  - i.e., convolution is simpler to compute there

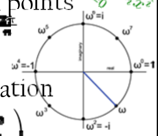
Gauss saves the day

    - n values of a polynomial suffice to know the polynomial itself
      - » convolution becomes a trivial pointwise product! $(p \cdot q)(x) = p(x)q(x)$

Discrete Fourier Transform

$\mathbf{a} \xrightarrow{\text{DFT}} \hat{\mathbf{a}}$ with $\hat{a}_k = p(\omega^k)$ $\quad p(x) = \sum_{i=0}^{n-1} a_i x^i$

- evaluate polynomial at n special points
  - $1, \omega, \omega^2, \ldots, \omega^{n-1}$ where $\omega = e^{-i\frac{2\pi}{n}}$
  - complex numbers… n^{th} root of 1
- equivalent to a matrix multiplication



$$\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} \hat{a}_0 \\ \hat{a}_1 \\ \vdots \\ \hat{a}_{n-1} \end{pmatrix}$$

  - very particular matrix (Vandermonde)

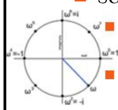
Efficient DFT (= FFT)

Fast (Discrete) Fourier Transform

- let's use an old friend: recursion (assume: n power of 2)

$$p(x) = \sum_{i=0}^{n-1} a_i x^i \quad p_{\text{even}}(x) = \sum_{i=0}^{n/2-1} a_{2i}\, x^i \quad p_{\text{odd}}(x) = \sum_{i=0}^{n/2-1} a_{2i+1}\, x^i \quad p(x) = p_{\text{even}}(x^2) + x\, p_{\text{odd}}(x^2)$$

- so, evaluation of *p* at $1, \omega, \omega^2, \ldots, \omega^{n-1}$ requires:



  - evaluate p_{even} and p_{odd} at $(1)^2, (\omega)^2, (\omega^2)^2, \ldots, (\omega^{n-1})^2$
    - *involves only **(n/2)** roots of unity, so recursive call perfect*
  - deduce *p* with $p(x) = p_{\text{even}}(x^2) + x\, p_{\text{odd}}(x^2)$

FFT Pseudocode

ComplexNumber[] FFT(**a**, ω, n)
if n=1 return **a**
evens = FFT((a_0, a_2, …, a_{n-2}), ω^2, n/2)
odds = FFT((a_1, a_3, …, a_{n-1}), ω^2, n/2)
x=1
for i=0 to n/2-1 *[assemble the result]*
- a[i] = evens[i] + x . odds[i]
- a[i+n/2] = evens[i] – x.odds[i] *[because $\omega^{n/2} = -1$]*
- x = x . ω *[i.e., $x = \omega^{i+1}$]*

return **a**

Back and Forth Conversion

$$\mathbf{a} \underset{\text{FFT}^{-1}\,?}{\overset{\text{FFT}}{\rightleftarrows}} \hat{\mathbf{a}}$$

- luckily, inverse Fourier matrix easy

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

- so inverse almost the same procedure

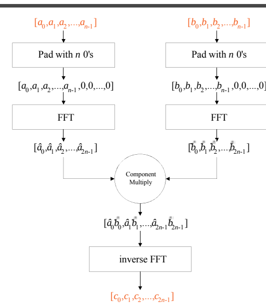
for i=0 to n/2-1
a[i] = [evens[i] + x . odds[i]]/n
a[i+n/2] = [evens[i] – x.odds[i]]/n
x = x . ω^{-1}

Fast Polynomial Coeffs Product

From the two polynomial *p* and *q*
- first pad their coefficients with 0
- then, $\mathbf{c} = \mathbf{a} \otimes \mathbf{b} = \text{FFT}^{-1}_{2n}(\text{FFT}_{2n}(\mathbf{a}) \cdot \text{FFT}_{2n}(\mathbf{b}))$

Claim: the imaginary parts of ***c*** will be *0*
- FFT of a vector of reals has special structure
- i.e., $\hat{a}_k = \hat{a}^*_{n-k}$ (conjugate)
  - so redundancy present in this case; could be optimized…
- property preserved after point-wise mult.
  - so **c** is real too, and math is not broken

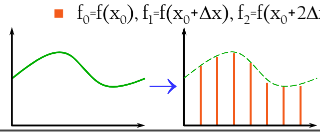
FFT-based Polynomial Mult.



FFT of Discrete Signals (I)

FFT can be applied to other types of data
- FFT takes and returns n complex numbers

Examples: discrete signals
- can't store continuous function f(t)
- so store "samples" at regular time ntervals
  - $f_0=f(x_0)$, $f_1=f(x_0+\Delta x)$, $f_2=f(x_0+2\Delta x)$, $f_3=f(x_0+3\Delta x)$, ...



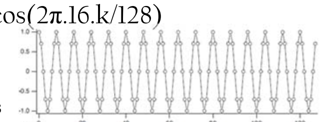
CD: 44.1K samples/second
DVD: 720*480 at 30 frames/sec
➔ 10.4 M samples/sec

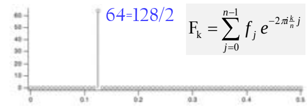
FFT of Discrete Signals (II)

Let's try it
- 128 samples of $\cos(2\pi.16.k/128)$
  - notice: periodic!
- surprise
  - two non-0 values
    - $e^{2i\pi.16/128}/2$, $e^{-2i\pi.16/128}/2$
      - » (times n)
  - frequency content!

$\hat{a}_k = \hat{a}^*_{n-k}$

FFT decomposes a signal into freqs





64=128/2

$$F_k = \sum_{j=0}^{n-1} f_j e^{-2\pi i \frac{k}{n} j}$$

0.125=16/128

Signal Processing Viewpoint

FFT returns *complex values*
- real part represents cosine components
- imaginary part represents sine components
- can be converted to *magnitude* and *phase*
  - squared magnitude represents **signal power**
    - what you see on your stereo

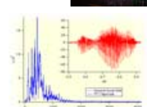
Time vs Frequency domains

$$F_k = \sum_{j=0}^{n-1} f_j e^{-2\pi i \frac{k}{n} j}$$

DFT

$$f_j = \frac{1}{n}\sum_{k=0}^{n-1} F_k e^{2\pi i \frac{j}{n} k}$$

IDFT

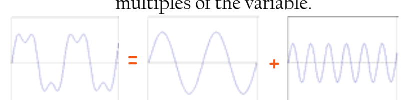


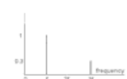
[Note: Joseph Fourier]

DFT is discrete version of Fourier Transform

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(w) e^{-jwt}\,dw \qquad F(w) = \int_{-\infty}^{\infty} f(t) e^{jwt}\,dt$$

- Fourier initially claimed that:
  - any function of a variable, whether continuous or discontinuous, can be expanded in a series of sines of multiples of the variable.

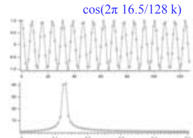


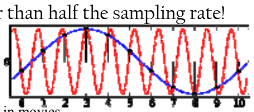


Careful

To things to watch for…
- non-periodic signal
  - presence of "jump(s)"
    - creates "leakage" in frequency
    - solution? windowing
- undersampling
  - can't capture freqs higher than half the sampling rate!
    - Nyquist frequency limit
  - **aliasing**
    - interpreted as lower freqs
      - » wheels rolling backwards in movies…

cos(2π 16.5/128 k)





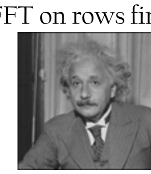
FFT of nD Signals?

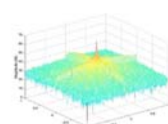
Easy to generalize to arbitrary dimension

E.g., in 2D:

$$F(u,v) = \sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(a,b) e^{-j2\pi(ua/M+vb/N)}$$

- FFT on rows first, then FFT on columns

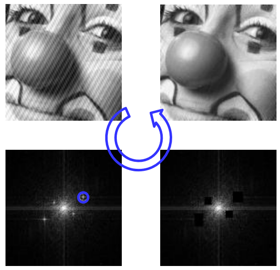




Signal Processing: Example

Edit image by altering frequency content

- can also do:
  - Darth Vader voice
  - (de)noising
  - (de)blurring
  - compression
  - etc…

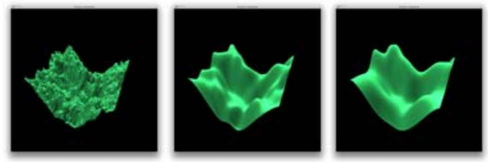


Works for Shapes Too

From a 1D FFT…

can create height fields from 2D spectra

- then play with frequencies



Other Numerical Methods

Numerics important in lots of applications

- from medical diagnosis
- to physical simulation
  - see HMW
  - even a google search is heavy numerics
    - linear algebra (eigenvalue problem to be exact)