Numerical Analysis
FMN011

Carmen Arévalo

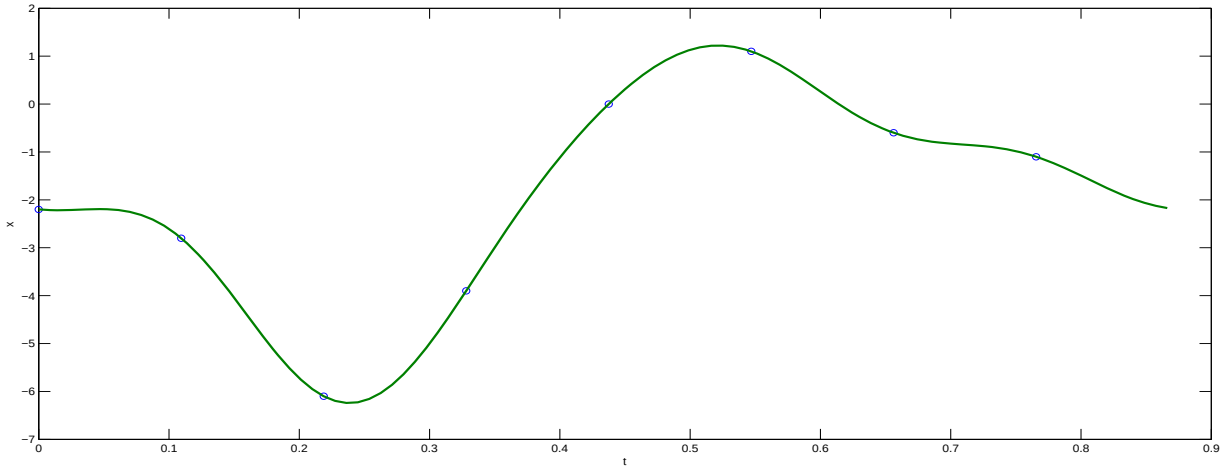
Lund University

carmen@maths.lth.se

Applications of the FFT

Trigonometric interpolation of data

```
>> t=linspace(0,7/8,8);
>> x=[-2.2 -2.8 -6.1 -3.9 0 1.1 -0.6 -1.1];
>> n=8;p=100;
>> inter=[0 1];
>> xp=dftinterp(inter,x,n,p);
```



Orthogonal Function Interpolation

Suppose you have real data $\{(t_0, x_0), (t_1, x_1), \ldots, (t_{n-1}, x_{n-1})\}$ and functions $\{f_0(t), f_1(t), \ldots, f_{n-1}(t)\}$ are chosen so that matrix

$$A = \begin{pmatrix} f_0(t_0) & f_0(t_1) & \cdots & f_0(t_{n-1}) \\ f_1(t_0) & f_1(t_1) & \cdots & f_1(t_{n-1}) \\ \vdots & \vdots & & \vdots \\ f_{n-1}(t_0) & f_{n-1}(t_1) & \cdots & f_{n-1}(t_{n-1}) \end{pmatrix}$$

is a real orthogonal matrix. Then a function that interpolates the points (t_j, x_j) is

$$F(t) = \sum_{k=0}^{n-1} y_k f_k(t),$$

where $y = Ax$.

Example

Let $\{t_0, t_1, \ldots, t_{n-1}\}, t_j = c + j(d-c)/n$ be the n (even) equally spaced points on $[c, d]$ and take the n functions

$$\left\{ \sqrt{\frac{1}{n}}, \sqrt{\frac{2}{n}} \cos \frac{2\pi(t-c)}{d-c}, \sqrt{\frac{2}{n}} \sin \frac{2\pi(t-c)}{d-c}, \right.$$
$$\left. \sqrt{\frac{2}{n}} \cos \frac{4\pi(t-c)}{d-c}, \sqrt{\frac{2}{n}} \sin \frac{4\pi(t-c)}{d-c}, \cdots, \sqrt{\frac{1}{n}} \cos \frac{n\pi(t-c)}{d-c} \right\}$$

The matrix is

$$A = \sqrt{\frac{2}{n}} \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \cdots & \frac{1}{\sqrt{2}} \\ 1 & \cos\frac{2\pi}{n} & \cdots & \cos\frac{2\pi(n-1)}{n} \\ 0 & \sin\frac{2\pi}{n} & \cdots & \sin\frac{2\pi(n-1)}{n} \\ \vdots & \vdots & & \vdots \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}}\cos\pi & \cdots & \frac{1}{\sqrt{2}}\cos(n-1)\pi \end{pmatrix}$$

and it is orthogonal.

If $x = [-2.2, -2.8, -6.1, -3.9, 0, 1.1, -0.6, -1.1]^T$, $t \in [0, 1]$, then $n = 8$,
$y = Ax = [-5.5154, -1.4889, -5.1188, 2.2500, 1.6500, -0.7111, 0.3812, -0.7778]^T$

The function that interpolates the points (t_j, x_j) is $F(t) = \sum_{k=0}^{7} y_k f_k(t)$

$$\begin{aligned} F(t) &= -1.95 - 0.7445\cos 2\pi t - 2.5594\sin 2\pi t + 1.125\cos 4\pi t \\ &\quad +0.825\sin 4\pi t - 0.3555\cos 6\pi t + 0.1906\sin 6\pi t - 0.2750\cos 8\pi t \end{aligned}$$

agreeing with the DFT formula with $n = 8$ and $a + bi = F_n x$:

$$\begin{aligned} P_n(t) &= \frac{a_0}{\sqrt{n}} + \frac{2}{\sqrt{n}} \sum_{k=1}^{n/2-1} \left(a_k \cos \frac{2\pi k(t-c)}{d-c} - b_k \frac{2\pi k(t-c)}{d-c} \right) \\ &\quad + \frac{a_{n/2}}{\sqrt{n}} \cos \frac{n\pi(t-c)}{d-c} \end{aligned}$$

Least squares with DFT

If the number of points is large, we may want to fit a trigonometric polynomial of lower degree.

Let $\{t_0 = c, t_1, \ldots, t_{n-1} = c + (n-1)(d-c)/n\}$ be the n (even) equally spaced points on $[c, d]$ and suppose we want to have only the $m < n$ functions $\{f_0(t), f_1(t), \ldots, f_{m-1}(t)\}$, where m is even.

The least squares problem is to find $\{c_0, c_1, \ldots, c_{m-1}\}$ that minimizes

$$\left\| \sum_{k=0}^{m-1} c_k f_k(t) - x \right\|_2 \quad \text{for the given data vector } x$$

This is equivalent to minimizing the 2-norm of $A_m^T c - x$, where A_m is the matrix of the first m rows of A.

The normal equations

The normal equations are $A_m A_m^T c = A_m x$, but $A_m A_m^T = I$, so we get

$$c = A_m x \text{ (no solving, just a matrix-vector product!)}$$

The interpolating polynomial for points $(t_0, x_0), \ldots, (t_{n-1}, x_{n-1})$ is

$$F_n(t) = \sum_{k=0}^{n-1} y_k f_k(t), \quad \text{where} \quad y = Ax$$

and the least squares approximation using the first m basis functions is

$$F_m(t) = \sum_{k=0}^{m-1} y_k f_k(t)$$

Filtering

We can think of least squares as filtering out some frequencies.

If we take the first m basis functions, ordered in increasing frequencies, we call the DFT a low-pass filter, because we remove the higher frequencies and let pass only the m lowest.

By choosing which rows of the matrix A we keep, we can design different filters to let pass any frequencies of interest.

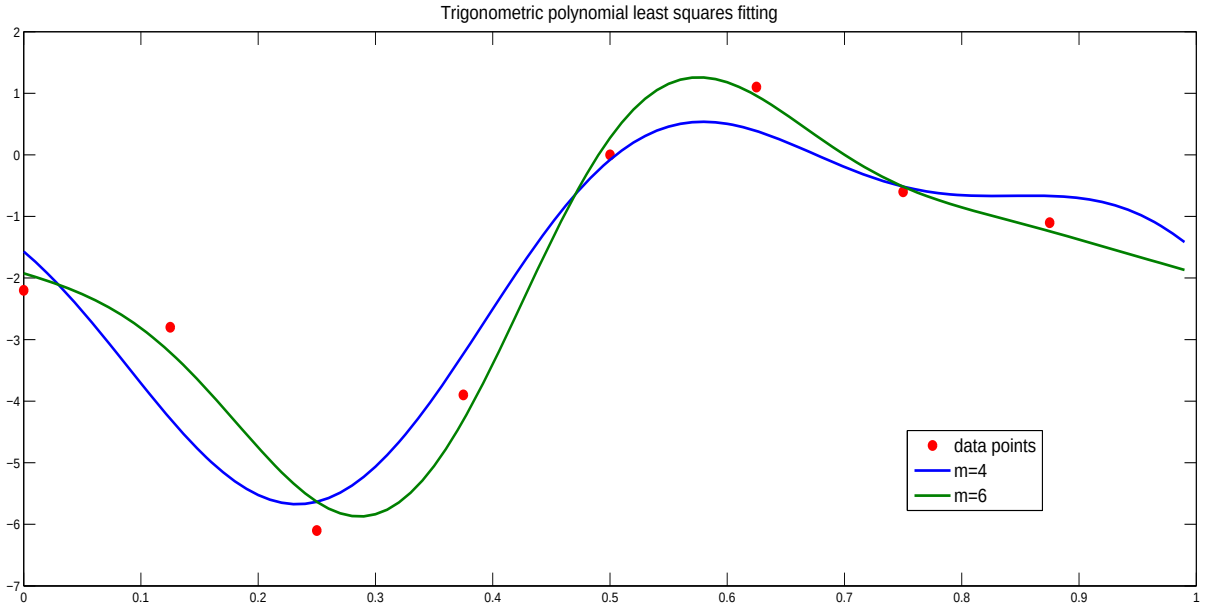
Least squares in MATLAB

```
%Lst sq for n points on [c,d], m <= n, plot at p evenly spaced pts
%Input: interval [c,d], data points x,
%  even number of data points n, even m, even number p>=n
%Output: filtered points xp
function xp=dftfilter(inter,x,m,n,p)
c=inter(1);d=inter(2);
t=c+(d-c)*(0:n-1)/n;          % n evenly-spaced time points
tp=c+(d-c)*(0:p-1)/p;         % p evenly-spaced time points
y=fft(x);                     % apply DFT
yp=zeros(p,1);                % yp will hold coefficients for ifft
yp(1:m/2)=y(1:m/2);           % keeps first m frequencies
yp(m/2+1)=real(y(m/2+1)); % keep cos terms only
if (m<n)
   yp(p-m/2+1)=yp(m/2-1);  % add complex conjugate to
end                           % corresponding place in upper tier
```

```
xp=real(ifft(yp))*(p/n);    % invert fft to recover data
plot(t,x,'o',tp,xp)         % plot data pts and least sq approx
```

Example: low-pass filters with m=4 and m=6

```
>> t=linspace(0,7/8,8);
>> x=[-2.2 -2.8 -6.1 -3.9 0 1.1 -0.6 -1.1];
>> xp=dftfilter([0 1],x,4,8,100);
```



Audio filtering

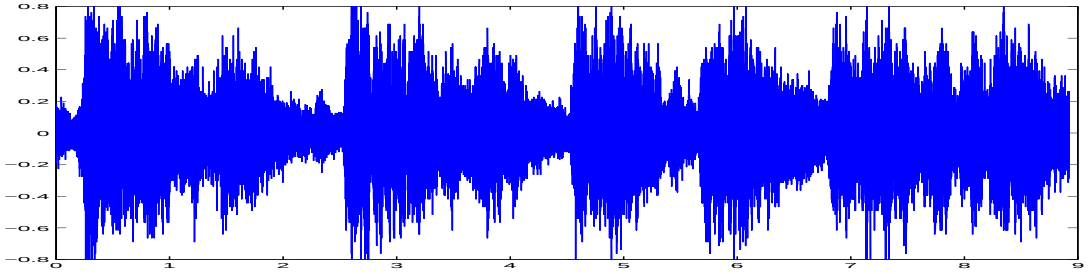
Filtering is a form of lossy compression. The goal is to reduce the amount of data that records a signal, without compromising the musical or audio effect.

Filtering is also used to remove noise (high frequencies).

The audio signal consists of real numbers (sound intensities) sampled in time. We say the information is in the time domain. To compress or clean up the signal, we apply a DFT and manipulate the frequency components in the frequency domain. Then we invert the DFT to get back the modified audio signal.

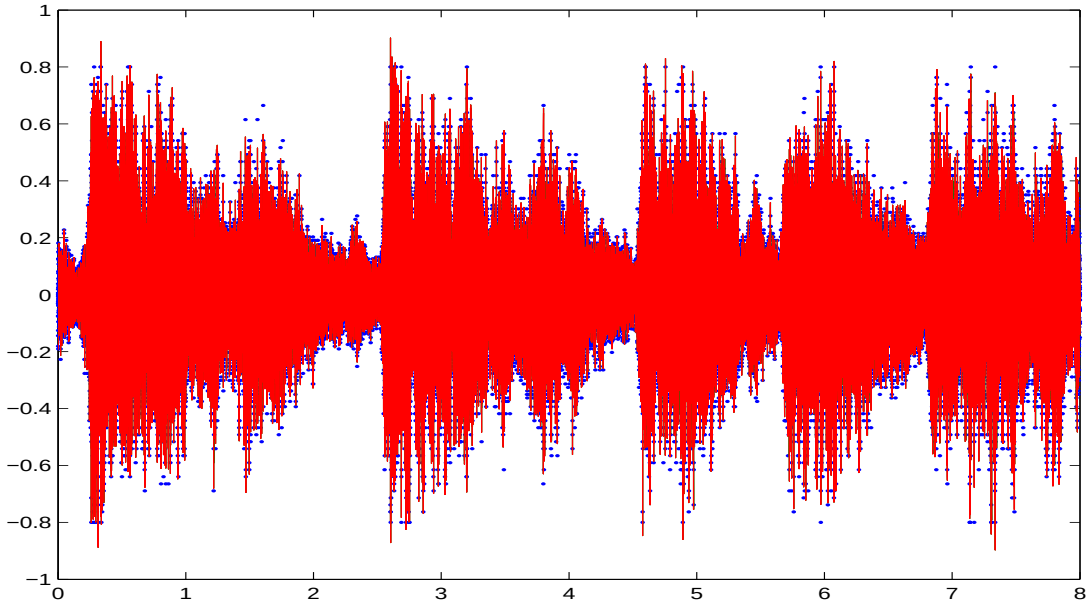
Handel's Hallelujah

```
>> load handel
>> sound(y,Fs)
>> secs=length(y)/Fs;
>> t=linspace(0,secs*(Fs-1)/Fs,length(y));
>> plot(t,y)
```



Compression 2:1

```
>> length(y)
ans =
       73113
>> 2^16
ans =
       65536
>> yn=y(1:2^16);
>> number_of_seconds=length(yn)/Fs
number_of_seconds =
     8
>> y2=dftfilter([0 8],yn,length(yn)/2,length(yn),2*length(yn));
>> sound(y2,Fs*2);
>> hold on
>> t=linspace(0,8*(2*Fs-1)/(2*Fs),length(y2));
>> plot(t,y2,'r')
```


1
0.8
0.6
0.4
0.2
0
−0.2
−0.4
−0.6
−0.8
−1
0
1
2
3
4
5
6
7
8

Noise removal

```
>> load handel
>> ynew=y(1:2^16);
>> p=1.3;
>> noise=std(ynew)/2;
>> r=noise*rand(2^16,1);
>> x=ynew+r;
>> sound(x,Fs)
>> fx=fft(x);sfx=conj(fx).*fx;
>> sfyapprox=max(sfx-2^16*(p*noise)^2,0);
>> phi=sfyapprox./sfx;
>> xout=real(ifft(phi.*fx));
>> pause
>> sound(xout,Fs)
```