CLASSROOM NOTES FOR APPROXIMATION THEORY
MATH 441, FALL 2009

THOMAS SHORES

Last Rev.: 9/15//09

1. Text Chapter 3 Notes

(9/10/09) Recall that from our text we have these definitions and Theorems. Unless otherwise stated, we assume that $\mathcal{A}$ is a subset of the normed vector space $\mathcal{B}$ with norm $\|\cdot\|$.

Definition. The *norm* of a (not necessarily linear) operator $X : \mathcal{B} \to \mathcal{A}$ is the smallest number $\|X\|$ such that for all $f \in B$,

$$\|X(f)\| \leq \|X\| \cdot \|f\|,$$

if such an $\|X\|$ exists. In this case the operator is said to be *bounded* and otherwise, *unbounded.*

Notice that in the case of a linear operator X, i.e., $X(af + bg) = aX(f) + bX(g)$ for scalars a, b and $f, g \in B$, this is equivalent to the definition that we gave in the background notes.

Definition. An operator $X : \mathcal{B} \to \mathcal{A}$ is said to be an *approximation (projection) operator* if for all $f \in \mathcal{B}$, we have

$$X(X(f)) = X(f).$$

In the case of an approximation operator X, the quantity $\|X\|$ is often called the *Lebesgue constan*t of the operator.

A sufficient condition for an operator to be an approximation operator is the following (stronger) condition: for all $a \in \mathcal{A}$,

$$X(a) = a. \tag{3.2}$$

Most of the operators we deal with will have the property (3.2).

Next, we give a name to the distance from f to the subset cA.

Definition. Given $f \in \mathcal{B}$ and approximant subset $\mathcal{A}$, we define

$$d^*(f) = \min_{a \in \mathcal{A}}$$

Here is a situation in which we can actually estimate d^* in a useful way:

Theorem. *(text 3.2) If $c = \pi/2$, $\mathcal{B} = C^k[a, b]$ and $\mathcal{A} = \mathcal{P}_n$ with $1 \leq k \leq n$, then for any $f \in \mathcal{B}$,*

$$d^*(f) \leq \frac{(n-k)!}{n!} c^k \left\| f^{(k)} \right\|_\infty.$$

Specifically, the constant is Here is the connection between two of the definitions just discussed:

Theorem. *(text 3.1) Let $\mathcal{A}$ be a finite dimensional subspace of $\mathcal{B}$ and $X : \mathcal{B} \to \mathcal{A}$ a linear operator satisfying (3.2). Then*

$$\|f - X(f)\| \leq \{1 + \|X\|\} d^*(f).$$

OK, now let's do a bit of experimenting with a simple linear approximation operator:

Example. Let $\mathcal{B} = C^1[0,1]$, $\mathcal{A} = \mathcal{P}_1$, and $X : \mathcal{B} \to \mathcal{A}$ the interpolation operator that interpolates $f(x)$ at $x = x_0, x_1$, with $a \leq x_0 < x_1 \leq b$. Let's experiment with a reasonable interesting function first and see how Theorems 3.1 and 3.2 fare. Recall that we can find an interpolating polynomial $p(x) = ax+b$ through (x_0, y_0) and (x_1, y_1) by way of the vandermonde matrix system

$$\begin{aligned} ax_0 + b &= y_0 \\ ax_1 + b &= y_1. \end{aligned}$$

Furthermore, this is a linear operator (we checked this in class), so that text Theorem 3.1 applies. The norm of X we calculated in class to be 1.

```
% start with f(x) = exp(sin(3*x))
fcn = inline('exp(sin(3*x))','x')
fcnp = inline('exp(sin(3*x)).*cos(3*x)*3','x')
xnodes = (0:.01:1)';
plot(xnodes,fcn(xnodes))
hold on, grid
% start with the endpoints for interpolation nodes
p = [0,1;1,1]\fcn([0;1])
plot(xnodes,polyval(p,xnodes))
dfXf = norm(polyval(p,xnodes)-fcn(xnodes),inf)
% not so hot, so let's look for a "best approximation"
nn = 20
astar = [1,nn];
dstar = dfXf;
xgrid = linspace(0,1,nn)';
for ii = 1:nn-1
for jj = ii+1:nn
p = [xgrid(ii),1;xgrid(jj),1]\fcn([xgrid(ii);xgrid(jj)]);
d = norm(polyval(p,xnodes)-fcn(xnodes),inf);
if (d < dstar)
dstar = d;
astar = [ii,jj];
end
end
end
pstar = [xgrid(astar(1)),1;xgrid(astar(2)),1]\fcn([xgrid(astar(1));xgrid(astar(2))]);
plot(xnodes,polyval(pstar,xnodes))
dstar
% compare results to Theorem 3.1 prediction
dfXf, (1+1)*dstar
% compare results to Theorem 3.2 prediction
c = pi/2
n = 1
k = 1
dstar, factorial(n-k)/factorial(n)*c*norm(fcnp(xnodes),inf)
```

Finally, here are some calculations regarding complexity of linear system solving that is alluded to in BackgroundNotes.pdf.

```
n = 80; a = n*eye(n)+rand(n); b = rand(n,1);
tic; c = a\b; toc
% now repeat with n = 160, 320 and 640...how does time grow?
```

2. Text Chapter 4 Notes

Let's do some calculations

```
x = (-5:0.1:5)';
fcn = inline('1./(1+x.^2)','x')
plot(x,fcn(x))
grid, hold on
n = 5
nodes = linspace(-5,5,n)';
fnodes = fcn(nodes);
p = vander(nodes)\fnodes;grid,
pnodes = polyval(p,x);
plot(x,pnodes);% what do you think?
norm(pnodes-fcn(x),inf)
% now start over and do it with n = 10, 20
% next, look for the culprit -- close the plot window
n = 5
nodes = linspace(-5,5,n)';
prd = ones(size(x));
for ii =1:n
prd = prd.*(x - nodes(ii));
end
plot(x,prd)
hold on, grid
% now repeat the loop with n = 10, 20
```

3. Pade Approximations

This is a rational function approximation whose basic idea is this: Let the approximating rational function to $f \in C^{N+1}[a,b]$ be a rational function

$$r(x) = \frac{p(x)}{q(x)} = \frac{p_0 + p_1 x + \cdots + p_n}{1 + q_1 x + \cdots + q_m x^m}$$

where $0 \in (a,b)$, $N = m + n$ is the "total degree" of $r(x)$ and require that

$$r^{(j)}(0) = f^{(j)}(0), \ j = 0, 1, \ldots, N,$$

so that $r(x)$ is something like a Taylor series approximation. This gives us $N+1$ conditions on $N+1$ coefficients $p_0, \ldots, p_n, q_1, \ldots, q_m$, so in principle we should be able to solve for all the coefficients (but there are no guarantees). Of course, we don't want to be in the business of computing all those derivatives, so here's an alternate approach: Let $g(x) = f(x) - r(x)$, so that the derivative condition is simply that all derivatives of g up to the N-th order vanish at $x = 0$. We have that

$$g(x) = \frac{q(x) f(x) - p(x)}{q(x)}$$

also has $N+1$ continuous derivatives defined at $x=0$, so has a Taylor series expansion

$$g(x) = g(0) + \frac{g'(0)}{1}x + \cdots + \frac{g^{(N)}(0)}{n!}x^N + \frac{g^{(N+1)}(\xi)}{(n+1)!}x^{N+1}$$

for some ξ between 0 and x. So the derivative conditions are equivalent to requiring that x^{N+1} be a factor of $g(x)$. However, the denominator has constant term 1, so x cannot be factored from it. Therefore, we must have that x^{N+1} is a factor of $q(x)f(x) - p(x)$.

Thus our strategy is to write out an expansion for $f(x)$, say

$$f(x) = a_0 + a_1 x + \cdots,$$

multiply terms and require that the first $N+1$ coefficients of

$$(1 + q_1 x + \cdots + q_m)(a_0 + a_1 x + \cdots) - (p_0 + p_1 x + \cdots + c_n x^n)$$

be zero. This won't require complicated derivatives, and can easily be done, once we have an expansion for f. In fact, the n-th term can be written out explicitly, so the conditions are

$$\begin{aligned} q_0 &= 1, \\ 0 &= \sum_{k=0}^{j} q_k a_{j-k} - p_j, \ j = 0, 1, \ldots, N. \end{aligned}$$

4. Trigonometric Polynomials

We covered the discrete Fourier series operator in class. Now let's put it to work. A review: With the standard inner product in $C[-\pi, \pi]$, we saw that the projection of of $f(x) \in C[-\pi, \pi]$ into the finite dimensional subspace

$$W_n = \operatorname{span}\{\cos kx, \sin kx \,|\, 0 \le k \le n\}$$

is the trig polynomial

$$q(x) = \frac{a_0}{2} + \sum_{k=1}^{n} (a_k \cos kx + b_k \sin kx),$$

where, for $0 \le k \le n$,

$$a_k = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x) \cos kx \, dx$$

and, for $1 \le k \le n$,

$$b_k = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x) \sin kx \, dx$$

If we use the trapezoidal method with $N+1$ equally spaced nodes $x_j = \frac{2\pi}{N} j$ on the interval $[0, 2\pi]$ (remember that the location of the interval makes no difference – only that it should be of length 2π) to approximate the Fourier coefficients a_k, b_k with a'_k, b'_k, then first and last values agree by periodicity and we obtain

$$a'_k = \frac{2}{N}\sum_{j=0}^{N-1} f(x_j) \cos\left(\frac{2\pi}{N} jk\right)$$

$$b'_k = \frac{2}{N}\sum_{j=0}^{N-1} f(x_j) \sin\left(\frac{2\pi}{N} jk\right)$$

First, let's create a function that actually evaluates a trig polynomial at vector argument x. We understand that such a trig polynomial will be specified by the vectors of coefficients

$$a = (a_1, \ldots, a_n, a_0)$$
$$b = (b_1, b_2, \ldots, b_n)\,.$$

OK, here it is:

```
function retval = trigeval(a,b,x)
% usage:  y = trigeval(a,b,x)
% description:  given vectors of trig coefficients a, b
% of length n+1 and n, respectively, and argument vector
% x, this function returns the value of the trig series
% p(x) = a(n+1)/2 + sum(a(j)*cos(j*x)+b(j)*sin(j*x),j=1..n).
%
n = length(b);
if (length(a) ~= n+1)
  error('incorrect vector lengths');
end
retval = a(n+1)/2*ones(size(x));
for j = 1:n
  retval = retval + a(j)*cos(j*x)+b(j)*sin(j*x);
end
```

Now create a script that does the work of building a discrete Fourier trig polynomial from 'myfcn.m'. It would look something like this:

```
% script:  fouriertest.m
% description:  perform some approximation experiments with
% Fourier polynomials on the function myfcn defined externally.

N = 3;
n = 1;
a = zeros(1,n+1);
b = zeros(1,n);
% construct the coefficients for myfcn
xnodes = linspace(0,2*pi,N+1);
xnodes = xnodes(1:N); % trim off 2*pi
xnodes = xnodes - pi; % let's work on [-pi,pi]
f = myfcn(xnodes);
% dispose of 0th coefficients
a(n+1) = 2*sum(f)/N;
% now the rest
for j = 1:n
  a(j) = 2*f*cos(j*xnodes)'/N;
  b(j) = 2*f*sin(j*xnodes)'/N;
end
% now try plotting
x = -pi:.001:pi;
```

```
clf
plot(x,myfcn(x))
hold on, grid
plot(x,trigeval(a,b,x))
```

Now for some experiments:

(1) Let's start with something really simple like $f(x) = \sin 2x$, $N = 3, 4$ and n from interpolation rate $\left\lfloor \frac{N-1}{2} \right\rfloor$ to N. Hmmm. Something's seriously wrong. Now try $N = 5, 7$.
(2) Now try $f(x) = \sin 2x - 0.2 * \cos 5x$, but start with $N = 5$.

What's going on here? Nyquist discovered that in order to recover a periodic signal by sampling, a *sufficient* condition is that we have a sample rate more than twice the highest frequency (in a sinusoidal component) of the signal. It is not a necessary condition. This number is called the *Nyquist sampling rate*. Recall that frequency is defined $f = 1/T$, where T is the period of a function. In the first example, the highest frequency is $1/\pi$, so our sampling rate must be greater than $2/\pi$. Thus in an interval of width 2π, we could sample with

$$N > \left\lfloor \frac{2\pi}{2/\pi} \right\rfloor \approx 9.87,$$

so $N = 10$ will work. However, we found that $N = 5$ and $n = 2$ worked. What went wrong with larger n? Consider this simple example: If we sample with $N = 2$, we cannot distinguish between $\sin x$, $\sin 2x$, $\sin 3x$, etc. This phenonemon is called *aliasing*, and is well studied in the area of signal processing.

Now consider the second example. The highest frequency occurring is $5/(2\pi)$, so the Nyquist sampling rate gives that in an interval of width 2π a sufficient sampling number is

$$N > \left\lfloor \frac{2\pi}{2/(5\pi)} \right\rfloor \approx 49.35.$$

This is far too conservative for our second example, but it would work.

Finally, let's break the rules and change the function to a square wave: $f(x) = \text{sign}(\text{x})$. Experiment with increasing sample sizes. This time, settle on a trig polynomial size and increase sampling until we are satisfied. Draw a separate figure for $n = 5, 11, 21$. Conclusions?

Note that we saw several interesting features:

(1) The discrete Fourier series does seem to converge pointwise to $f(x)$.
(2) At the discontinuity it appears to be converging to a point half way between the left and right-hand limits.
(3) The "bumps" near the discontinuity appears to gradually contract in width, but their height does not go to zero. This is the so-called "Gibbs effect" that was observed around the end of the nineteenth century by J. Gibbs.

5. Complex Trigonometric Polynomials

Here the trig functions are replaced the by the complex exponentials $e^{\mathrm{i}kx}$, $k = -n, \ldots 0, \ldots, n$, which turn out to be an orthogonal set of vectors in the space $C[-\pi, \pi]$ of continuous complex-valued functions with domain $[-\pi, \pi]$. One can define the complex function e^z in terms of the power series for the exponential function learned in Calculus, but for our purposes, it suffices to use this fact as definition: For real x,

$$e^{\mathrm{i}x} = \cos x + \mathrm{i} \sin x.$$

We use the standard complex inner product

$$\langle f, g\rangle = \int_{-\pi}^{\pi} f(x)\,\overline{g(x)}dx$$

and induced norm $\|f\|^2 = \langle f, f\rangle$ in this setting. Remember that to integrate a complex function, you simply integrate its real and complex parts, that is, if $f(x) = g(x) + \mathrm{i}h(x)$, with $g(x)$ and $h(x)$ real-valued, then

$$\int_a^b f(x)\,dx = \int_a^b g(x)\,dx + \mathrm{i}\int_a^b h(x)\,dx.$$

Let W_n be the finite dimensional subspace spanned by these exponentials and, as with the trig polynomials, we obtain the standard projection formula for the best approximation to $f(x) \in C[-\pi, \pi]$ from W_n is given by

$$q(x) = \sum_{k=-n}^{n} c_k e^{\mathrm{i}kx},$$

where

$$c_k = \frac{\langle f, e^{\mathrm{i}kx}\rangle}{\langle e^{\mathrm{i}kx}, e^{\mathrm{i}kx}\rangle} = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(x)\,e^{-\mathrm{i}kx}dx.$$

The bridge between these coefficients and the real Fourier coefficients consists of the following identities for $k \geq 0$, which are easily checked from definitions:

$$\begin{aligned} a_k &= c_k + c_{-k} \\ b_k &= \mathrm{i}(c_k - c_{-k}) \\ c_k &= \frac{1}{2}(a_k - \mathrm{i}b_k) \\ c_{-k} &= \frac{1}{2}(a_k + \mathrm{i}b_k). \end{aligned}$$

We can approximate the c_k by a trapezoidal integration using $N+1$ equally spaced nodes $x_j = \frac{2\pi}{N}j$ on the interval $[0, 2\pi]$, as with real Fourier coefficients, and obtain the approximation c_k' to c_k as

$$c_k' = \frac{1}{N}\sum_{j=0}^{N-1} f(x_j)\,e^{-\frac{2\pi\mathrm{i}}{N}jk}.$$

We use the notation $\omega_N = e^{2\pi\mathrm{i}/N}$ and $\omega = \omega_N^{-1} = \overline{\omega}_N$ (check these equalities for yourself).

```
Let's do some experiments with Matlab:
% construct DFT matrices
N = 5
zeta = exp(-2*pi/N)
z = zeta.^(0:N-1)
help vander % this doesn't quite do what we want
F = fliplr(vander(z).'/N
Finv = N*conj(F)
F*Finv % it works!
```

O.K. now we are going to do some experiments that will be better handled with a script. So start Matlab and edit 'myfcn'. Change it to a square wave on $[0, 2\pi]$ by

```
retval = sign(x-pi);
```

Next, edit 'fouriertest', or whatever you called the file from the previous experiments. Now save it as 'DFTtest'. Here is what you need:

```
% script:  DFTtest.m
% description:  perform some approximation experiments with complex
% Fourier polynomials on the function myfcn defined externally.

N = 17; % use odd value to get perfect match with real coefs
n = round(N/2) - 1
a = zeros(1,n+1);
b = zeros(1,n);
c = zeros(1,N);
% construct the coefficients for myfcn
xnodes = linspace(0,2*pi,N+1);
xnodes = xnodes(1:N); % trim off 2*pi
f = myfcn(xnodes);
% dispose of 0th coefficients
a(n+1) = 2*sum(f)/N;
% now the rest of real Fourier coefs
for j = 1:n
  a(j) = 2*f*cos(j*xnodes)'/N;
  b(j) = 2*f*sin(j*xnodes)'/N;
end
% next construct the complex coefficients
for j = 1:N
  c(j) = f*exp(-i*(j-1)*xnodes).'/N;
end
% ok, let's compare
% a(n+1) is exceptional Fourier a_0
disp('|a0 - 2*c0| =')
disp(abs(a(n+1) - 2*c(1)))
% rest of the a(k) are Fourier a_k
ac = c(2:n+1)+c(N:-1:n+2);
disp('norm(a(1:n) - (c(2:n+1)+c(N:-1:n+2)),inf) =')
disp(norm(a(1:n) - ac,inf))
ac(n+1) = 2*c(1);
% the b(k) are Fourier b_k
bc = i*(c(2:n+1)-c(N:-1:n+2));
disp('norm(b(1:n) - i*(c(2:n+1)-c(N:-1:n+2)),inf) =')
disp(norm(b(1:n) - bc,inf))
```

Once you have run DFTtest, try plotting:

```
x = 0:.01:2*pi;
plot(x,trigeval(a,b,x));
```

```
plot(x,trigeval(ac,bc,x));
norm(imag(ac),inf)
norm(imag(bc),inf)
```

6. An Application of the DFT

As usual, $\omega_N = e^{\frac{2\pi}{N}i}$, a primitive Nth root of unity. To recap, given periodic data $[y_k] = \{y_k\}_{k=0}^{N-1}$, we obtain transformed data $[Y_k] = \{Y_k\}_{k=0}^{N-1}$ (we're replacing c_k' by Y_k), and conversely, by way of the formulas

$$Y_k = \frac{1}{N}\sum_{j=0}^{N-1} y_j \omega_N^{-kj} \equiv \mathcal{F}_N\left([y_k]\right)$$
$$y_k = \sum_{j=0}^{N-1} Y_j \omega_N^{kj} \equiv \mathcal{F}_N^{-1}\left([Y_k]\right).$$

Next, let's alter the definition of the data $[y_k]$. Since the data is periodic, we can interpret it as a doubly infinite sequence

$$[y_k] \ldots, y_{-j}, \ldots, y_{-1}, y_0, y_1, \ldots, y_{N-1}, y_N, \ldots, y_j, \ldots$$

where we understand that in general, that if $j = qN + r$, with integers j, q, r and $0 \leq r < N$, then $y_j = y_r$. (If you know congruences, you will recognize that $j \equiv r \,(\mathrm{mod} N)$.) We have the same interpretation with transformed data $[Y_k]$, which is also periodic, given that $[y_k]$ is. Here is a key idea:

Definition 1. Given periodic data sets $[y_k]$ and $[z_k]$, we define the convolution of these to be

$$[w_k] = [y_k] * [z_k],$$

where

$$w_k = \sum_{j=0}^{N-1} y_k z_{k-j},\ k \in \mathbb{Z}.$$

The complexity of the convolution operator is $2N^2$, or simply $\mathcal{O}\left(N^2\right)$.

We can interpret polynomial multiplication as convolution in the following way: Given polynomials

$$p(x) = a_0 + a_1 x + \cdots + a_m x^m$$
$$q(x) = b_0 + b_1 x + \cdots + b_n x^n,$$

we know that

$$P(x)\,Q(x) = a_0 b_0 + (a_0 b_1 + a_1 b_0)\,x + \cdots + a_m b_n x^{m+n}.$$

The kth term is actually

$$\sum_{j+\ell=k} a_j b_\ell,$$

where $0 \leq j \leq m$ and $0 \leq \ell \leq n$. As far as complexity, we see that the complexity is about $2(m+1)(n+1)$ flops, so certainly $\mathcal{O}(mn)$.

Let $N \geq m + n + 1$ and pad the vectors of coefficients with zeros to obtain vectors

$$a = (a_0, a_1, \ldots, a_m, 0, \ldots, 0)$$
$$b = (b_0, b_1, \ldots, b_n, 0, \ldots, 0)$$

in the space $\mathbb{C}^N$. Next form the periodic data sets $[a_k]$ and $[b_k]$, form $[c_k] = [a_k] * [b_k]$ and observe that for $0 \leq k \leq m+n$,

$$\begin{aligned} c_k &= \sum_{j=0}^{N-1} a_j b_{k-j} \\ &= \sum_{j+\ell=k} a_j b_\ell, \end{aligned}$$

Here the padded zeros ensure that $0 \leq j \leq m$ and $0 \leq \ell \leq n$. Moreover, if $k > m+n$, then the padded zeros will make the sum identically 0, since one factor in each term will always be zero. So this is exactly the same sum as for the coefficients of the product polynomial, padded with zeros beyond the $(m+n+1)$th entry. We need one more key fact:

Theorem 2. *If* $\mathcal{F}_N([y_k]) = [Y_k]$ and $\mathcal{F}_N([z_k]) = [Z_k]$, then

$$\mathcal{F}_N([y_k] * [z_k]) = [NY_k Z_k]$$

and

$$[y_k] * [z_k] = \mathcal{F}_N^{-1}([NY_k Z_k]).$$

Proof. Let $[w_k] = [y_k] * [z_k]$ and calculate

$$\begin{aligned} W_k &= \frac{1}{N} \sum_{j=0}^{N-1} \left(\sum_{\ell=0}^{N-1} y_\ell z_{j-\ell} \right) \omega_N^{-kj} \\ &= \frac{1}{N} \sum_{\ell=0}^{N-1} \left(\sum_{j=0}^{N-1} y_\ell z_{j-\ell} \right) \omega_N^{-kj} \\ &= \frac{1}{N} \sum_{\ell=0}^{N-1} y_\ell \omega_N^{-k\ell} \left(\sum_{j=0}^{N-1} z_{j-\ell} \omega_N^{-k(j-\ell)} \right) \\ &= \frac{1}{N} \sum_{\ell=0}^{N-1} y_\ell \omega_N^{-k\ell} \left(\sum_{m=-\ell}^{N-1-\ell} z_m \omega_N^{-km} \right) \\ &= N \left(\frac{1}{N} \sum_{\ell=0}^{N-1} y_\ell \omega_N^{-k\ell} \right) \left(\frac{1}{N} \sum_{m=0}^{N-1} z_m \omega_N^{-km} \right) \\ &= N Y_k Z_k. \end{aligned}$$

The last equality is immediate.

All of this suggests an algorithm for computing convolutions: First, apply the DFT to each sequence, then take the coordinate-wise product of the transforms, and finally take the inverse transform of this product. Although we can say that the complexity is $\mathcal{O}(N^2)$, clearly there is a factor of 10 or so that would seem to be too much work when contrasted with doing it directly. So what's the point? Enter the FFT, which will reduce the work of a transform (or inverse transform) to $\mathcal{O}(N \log_2 N)$. Now this idea will become profitable in some situations.

Let's confirm the convolution theorem in a special case, namely for the multiplication of $p(x) = (1-x)^2$ and $q(x) = (1-x)^3$ by means of the convolution trick.

```
N = 8
zeta = exp(-2*pi*i/N).^(0:N-1);
```

```
F = fliplr(vander(zeta))/N;
Finv = N*conj(F);
y = [1,-2,1,0,0,0,0,0]'
z = [1,-3,3,-1,0,0,0,0]'
Y = F*y;
Z = F*z;
W = Y.*Z
w = Finv*(N*W)
```

7. The Fast Fourier Transform (FFT)

Description. We'll do pretty much everything you need to know about the development of the FFT. OK, let's start with a review of the definition of the discrete Fourier transform (DFT): given a data sequence (y_k) of complex numbers, periodic of period N, we define the DFT of this data to be the periodic sequence (Y_n) of period N given by the formula

$$Y_k = \frac{1}{N}\left\{y_0 + y_1\omega_N^{-k} + y_2\omega_N^{-2k} + \cdots + y_{N-1}\omega_N^{-(N-1)k}\right\}, \quad k = 0, 1, \ldots, N-1$$

(Since the Y_k are a periodic sequence, these are all the values we need to know.) Symbolically, we write $[Y_k] = \mathcal{F}_N([y_k])$. Now assume that N is even, say $N = 2m$ and we can split this sum into even and odd indexed terms and factor ω_N^{-k} from the odd terms to get that for $k = 0, 1, \ldots, N-1$,

$$Y_k = \frac{1}{2}\left\{E_k + \omega_N^{-k}O_k\right\}$$

where

$$\begin{aligned} E_k &= \tfrac{1}{m}\left\{y_0 + y_2\omega_N^{-2k} + y_4\omega_N^{-4k} + \cdots + y_{N-2}\omega_N^{-(N-2)k}\right\} \\ O_k &= \tfrac{1}{m}\left\{y_1 + y_3\omega_N^{-2k} + y_5\omega_N^{-4k} + \cdots + y_{N-1}\omega_N^{-(N-2)k}\right\} \end{aligned}$$

Here are some simple facts:

$$\begin{aligned} \omega_N^{-(k+m)} &= -\omega_N^{-k} \\ E_{k+m} &= E_k \\ O_{k+m} &= O_k \end{aligned}$$

It follows that for $k = 0, 1, \ldots, N/2 - 1$, we have these *DFT doubling* formulas

$$\begin{aligned} Y_k &= \tfrac{1}{2}\left\{E_k + \omega_N^{-k}O_k\right\} \\ Y_{k+m} &= \tfrac{1}{2}\left\{E_k - \omega_N^{-k}O_k\right\}. \end{aligned}$$

Observe that this essentially cuts the work of computing Y_k in half, requiring about $N^2/2$ multiplications and additions instead of N^2. This observation lies at the heart of the FFT.

However, the real power of this idea is realized when we team it up with recursion. To this end, we suppose that N is purely even, that is, $N = 2^p$ for some integer p. Now make the key observation that calculation of the E_k and O_k's is itself a DFT. For we have that $\omega_N^2 = \omega_{N/2}$, from which it follows that

$$\begin{aligned} E_k &= \tfrac{1}{m}\left\{y_0 + y_2\omega_{N/2}^{-n} + y_4\omega_{N/2}^{-2n} + \cdots + y_{N-2}\omega_{N/2}^{-(m-1)k}\right\} \\ O_k &= \tfrac{1}{m}\left\{y_1 + y_3\omega_{N/2}^{-n} + y_5\omega_{N/2}^{-2n} + \cdots + y_{N-1}\omega_{N/2}^{-(m-1)k}\right\} \end{aligned}$$

In other words, the two quantities above are simply the DFT of the pairs of the half sized data $y_0, y_2, \ldots, y_{N-2}$ and $y_1, y_3, \ldots y_{N-1}$. Thus, we see that we can halve these two data sets again and reduce the problem to computing four quarter sized DFTs. Recursing all the way down to the bottom, we reduce the original problem to computing N DTFs of $1/N$-th sized data sets. It takes us p of these steps to get down to the bottom, namely size $N/2^0$ to $N/2^1$, then $N/2^1$ to $N/2^2$, all the way down to $N/2^{p-1}$ to $N/2^p = 1$. However, at the bottom, the DFT of a singleton data point is simply the data point itself. Are we done?

Well, not quite. This is only half the problem, because once we've reached the bottom, we have to reassemble the data a step at a time by way of DFT doubling formulas to build the DFT at at the top. Let's examine how we have to rearrange our original data set for this recursion in the case $p = 3$, that is, $N = 8$. Designate levels by ℓ.

$\ell = 3$	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7
$\ell = 2$	y_0	y_2	y_4	y_6	y_1	y_3	y_5	y_7
$\ell = 1$	y_0	y_4	y_2	y_6	y_1	y_5	y_3	y_7
$\ell = 0$	y_0	y_4	y_2	y_6	y_1	y_5	y_3	y_7

Now we're ready to work our way back up. Notice, BTW, $\ell = 1$ really is the same as the bottom.

Complexity. The recursive way of thinking is very handy. For one thing, it enables us to do a complexity analysis without much effort. We'll ignore the successive divisions by 2. We could just ignore them during the computations, then at the end divide our final result by $2^p = N$. This only involves N real multiplications, which is negligible compared to the rest of the work. Also notice that the number of multiplications for a full reconstruction is about half the number of additions, so we'll focus on additions. (Bear in mind that we assume the values of ω_N are precomputed and available.) How much work is involved in going from the two data sets to the single one of length 2^p? Let's denote the total number of additions by A_p. We see from the formulas for Y_k and Y_{k+m} that

$$A_p = 2A_{p-1} + 2^p$$

and down at the bottom $A_0 = 0$. Recursion yields

$$\begin{aligned} A_p &= 2A_{p-1} + 2^p \\ &= 2(2A_{p-2} + 2^{p-1}) + 2^p \\ &\vdots \\ &= 2^p A_0 + p2^p = N \log_2 N \end{aligned}$$

Similarly, one could count multiplications M_p recursively, with the recursion formula being

$$M_p = 2M_{p-1} + 2^{p-1} - 1$$

(don't count multiplication by $\omega_N^0 = 1$) and $M_0 = 0$ to obtain that $M_p = \frac{1}{2}N(\log_2 N - 2)$.

Implementation. We might be tempted to use recursion to program the algorithm. From a practical point of view, this is usually a bad idea, because recursion, the elegant darling of computer science aficionados, is frequently inefficient as a programming paradigm.

Rather, let's assume that we have decomposed that data set into the lowest level. That process alone is interesting, so suppose we have a routine that takes an index vector $0 : N-1$

and rearranges it in such a way that the returned index vector is what is needed to work our way back up the recursion chain of the FFT. The syntax of the routine should be

retval=FFTsort(p)

where $N = 2^p$.

With the routine FFTsort in hand, let's address the issue of practical coding of the FFT. Rather than thinking from the top down (recursion), let's think from the bottom up (induction). Of course, they're equivalent. It suffices to understand how things work from the $m = N/2$ to N. So let's re-examine the formulas from the point of view of DFTs. Consider this tabular form for the relevant data:

$\ell = p$	Y_0	Y_1	$\dots$					Y_{N-1}
$\ell = p-1$	E_0	E_1	$\dots$	E_{m-1}	O_0	O_1	$\dots$	O_{m-1}

Notice that the first half of the second row is the DFT of the first half of the second level of rearranged data (like $\ell = 2$ in our $N = 8$ example above). Let's call the process of applying Fact 2 to the second row to get the first row *DFT doubling*. Back to the general case, imagine that we're at the ℓth level in our reconstruction, where $0 \le \ell < p$. At this level we have constructed $2^{p-\ell}$ consecutive DFTs of size 2^ℓ, where each DFT is the DFT of the corresponding data at the ℓth level of rearranged input data. Now we pair these DFTs up to construct $2^{p-\ell-1}$ consecutive DFTs of size $2^{\ell+1}$. Here's a picture of what the DFT doublings look like in our $N = 8$ example (start at the bottom):

$\ell = 3$	$E = Y_0$	$E = Y_1$	$E = Y_2$	$E = Y_3$	$E = Y_4$	$E = Y_5$	$E = Y_6$	$E = Y_7$
$\ell = 2$	E	E	E	E	O	O	O	O
$\ell = 1$	E	E	O	O	E	E	O	O
$\ell = 0$	$y_0 = E$	$y_4 = O$	$y_2 = E$	$y_6 = O$	$y_1 = E$	$y_5 = O$	$y_3 = E$	$y_7 = O$

Keep going until we get to the top. When we get there we have reconstructed the $Y_n's$. One can envision two nested for loops that do the whole job. In fact, here's a simple Matlab code for the FFT. All divisions by 2 are deferred to the last line, as suggested by our earlier discussion.

FFT Algorithm:

```
function Y = FFTransform(y)
% usage:  Y = FFTransform(y)
% description:  This function accepts input vector y of length
% 2^p (NB: assumed and not checked for) to and outputs
% the DFT Y of y using the FFT algorithm.

y = y(:); % turn y into a column
N = length(y);
p = log2(N);
omegaN = exp(-pi*i/N).^(0:N-1).'; % all the omegas we need
Y = y(FFTsort(p)); % initial Y at bottom level
stride = 1;
for l=1:p
  omegal = omegaN(1:N/stride:N);
  stride = 2*stride;
  for j=1:stride:N
```

```
    evens = Y(j:j+stride/2-1);
    odds = omegal.*Y(j+stride/2:j+stride-1);
    Y(j:j+stride/2-1) = evens + odds;
    Y(j+stride/2:j+stride-1) = evens - odds;
  end
end
Y = Y/N;
```

We might ask what one has to do about the *inverse* DFT. Recall that this transform is given by the equation

$$y_k = \left\{ Y_0 + Y_1\omega_N^k + Y_2\omega_N^{2k} + \cdots + Y_{N-1}\omega_N^{(N-1)k} \right\}, \quad k = 0, 1, \ldots, N-1.$$

Symbolically, we write $(y_k) = \mathcal{F}_N^{-1}(Y_n)$. This operation does what it says, that is, compute untransformed data from transformed data. The good news is that we need do nothing more than write a program for the FFT. The reason is that from the definition we can see that

$$\overline{y}_k = N\frac{1}{N}\left\{ \overline{Y}_0 + \overline{Y}_1\omega_N^{-k} + \overline{Y}_2\omega_N^{-2k} + \cdots + \overline{Y}_{N-1}\omega_N^{-(N-1)k} \right\}, \quad k = 0, 1, \ldots, N-1,$$

from which it follows that

$$(y_k) = N\overline{\mathcal{F}_N(\overline{Y}_n)}.$$

Therefore, it suffices to have an fast algorithm for the DFT and a fast algorithm for the inverse DFT is more or less free.

8. All About Cubic Splines

We study cubic splines with a finite number of (ordered) knots at $x_0, x_1, \ldots, x_n$. The space of all such objects is$\mathcal{S}(x_0, x_1, \ldots, x_n; 4)$ and such an object is a function $s(x) \in C^2[x_0, x_n]$, such that on each subinterval $[x_j, x_{j+1}]$, $j = 0, 1, \ldots, n-1$, we have $s(x) = s_j(x)$, a cubic polynomial defined for $x \in [x_j, x_{j+1}]$.

General Equations for All Cubic Splines.

Here is the problem presented by knot interpolation alone.

Problem: Given a function $f(x) \in C^2[x_0, x_n]$, find all cubic splines that interpolate $f(x)$ at the knots $x_0, x_1, \ldots, x_n$, that is, $s(x_j) = f(x_j)$, $j = 0, 1, \ldots, n$.

Notation:

(1) $s_j(x) \equiv a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3$, $j = 0, 1, \ldots, n$-1.
(2) $y_j = f(x_j)$, $j = 0, 1, \ldots, n$.
(3) $M_j = s''(x_j)$, $j = 0, 1, \ldots, n$. (The M_j's are called the *second moments* of the spline.)
(4) $h_j = x_{j+1} - x_j$, $j = 0, 1, \ldots, n-1$.

Here is how we solve this problem: Start with $s''(x)$ which must be a piecewise linear continuous function. Since we know its endpoint values in the subinterval $[x_j, x_{j+1}]$, we may use simple Lagrange interpolation to obtain

$$s_j''(x) = \frac{(x_{j+1} - x)M_j + (x - x_j)M_{j+1}}{h_j}.$$

Integrate twice and we have to add in an arbitrary linear term which, for convenience we write as follows

$$s_j(x) = \frac{(x_{j+1}-x)^3 M_j + (x-x_j)^3 M_{j+1}}{6h_j} + C_j(x_{j+1}-x) + D_j(x-x_j),$$

where D_j and C_j are to be determined. Now differentiate this expression and obtain

$$s_j'(x) = \frac{-(x_{j+1}-x)^2 M_j + (x-x_j)^2 M_{j+1}}{2h_j} - C_j + D_j.$$

For later use we note that if we shift the indices down one, we obtain

$$s_{j-1}'(x) = \frac{-(x_j-x)^2 M_{j-1} + (x-x_{j-1})^2 M_j}{2h_{j-1}} - C_{j-1} + D_{j-1}.$$

Our strategy is to reduce the determination of every unknown to the determination of the M_j's. Start with the interpolation conditions for s, which implies that for $j = 0, 1, \ldots, n-1$,

$$s_j(x_j) = y_j = \frac{h_j^2}{6} M_j + C_j h_j,$$

so that

$$C_j = \frac{y_j}{h_j} - \frac{h_j M_j}{6}.$$

Similarly, from

$$s_j(x_{j+1}) = y_{j+1} = \frac{h_j^2}{6} M_{j+1} + D_j h_j,$$

we have

$$D_j = \frac{y_{j+1}}{h_j} - \frac{h_j M_{j+1}}{6}.$$

Next, use the continuity of $s'(x)$ to obtain that for $j = 1, \ldots, n-1$,

$$s_{j-1}'(x_j) = s_j'(x_j)$$

which translates into

$$\frac{h_{j-1}^2 M_j}{2h_{j-1}} - C_{j-1} + D_{j-1} = -\frac{h_j^2 M_j}{2h_j} - C_j + D_j$$

$$\frac{h_{j-1} M_j}{2} - \frac{y_{j-1}}{h_{j-1}} + \frac{h_{j-1} M_{j-1}}{6} + \frac{y_j}{h_{j-1}} - \frac{h_{j-1} M_j}{6} = -\frac{h_j M_j}{2} - \frac{y_j}{h_j} + \frac{h_j M_j}{6} + \frac{y_{j+1}}{h_j} - \frac{h_j M_{j+1}}{6}.$$

After simplifying, we obtain

$$\frac{h_{j-1}}{6} M_{j-1} + \left(\frac{h_{j-1}}{2} - \frac{h_{j-1}}{6} + \frac{h_j}{2} - \frac{h_j}{6} \right) M_j + \frac{h_j}{6} M_{j+1} = \frac{y_{j+1}}{h_j} - \frac{y_j}{h_j} + \frac{y_{j-1}}{h_{j-1}} - \frac{y_j}{h_{j-1}},$$

that is,

$$\frac{h_{j-1}}{6} M_{j-1} + \left(\frac{h_{j-1} + h_j}{3} \right) M_j + \frac{h_j}{6} M_{j+1} = f[x_j, x_{j+1}] - f[x_{j-1}, x_j].$$

Thus we are two equations short of what we need to determine the $n+1$ unknowns $M_0, M_1, \ldots, M_n$.

Standard Form for the Defining Cubics.

Notice that if we express the cubic polynomials defining $s(x)$ in standard form, then for $j = 0, 1, \ldots, n-1$, we have

$$\begin{aligned} s_j(x) &= a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3, \\ s_j'(x) &= b_j + 2c_j(x - x_j) + 3d_j(x - x_j)^2, \\ s_j''(x) &= 2c_j + 3d_j(x - x_j). \end{aligned}$$

We can use these equations along with our earlier description of $s_j(x)$ to obtain that for $j = 0, 1, \ldots, n-1$,

$$\begin{aligned} a_j &= f(x_j) \\ b_j &= f[x_j, x_{j+1}] - \frac{h_j}{6}(2M_j + M_{j+1}) \\ c_j &= \frac{M_j}{2} \\ d_j &= \frac{1}{6h_j}(M_{j+1} - M_j). \end{aligned}$$

These are the formulas that we need to put the spline in standard PP form.

Extra Conditions to Uniquely Define the Interpolating Cubic Spline.

Clamped (complete) cubic spline. Define the spline $s(x) = s_c(x)$ by the two additional conditions

$$s'(x_0) = f'(x_0), \quad s'(x_n) = f'(x_n).$$

Use the formula for $s'(x)$ to obtain additional equations

$$\begin{aligned} \frac{h_0}{3}M_0 + \frac{h_0}{6}M_1 &= f[x_0, x_1] - f'(x_0) \\ \frac{h_{n-1}}{6}M_{n-1} + \frac{h_{n-1}}{3}M_n &= f'(x_n) - f[x_{n-1}, x_n]. \end{aligned}$$

The resulting system has symmetric positive definite coefficient matrix, from which it follows that there is a unique clamped cubic spline interpolating $f(x)$ at the knots of the spline. The system is (the convention is that blank entries are understood to have value zero)

$$\begin{bmatrix} \frac{h_0}{3} & \frac{h_0}{6} & & & \\ \frac{h_0}{6} & \frac{h_0+h_1}{} & \frac{h_1}{6} & & \\ & \ddots & \ddots & \ddots & \\ & & \frac{h_{n-2}}{6} & \frac{h_{n-2}+h_{n-1}}{} & \frac{h_{n-1}}{6} \\ & & & \frac{h_{n-2}}{6} & \frac{h_{n-1}}{3} \end{bmatrix} \begin{bmatrix} M_0 \\ M_1 \\ \vdots \\ M_{n-1} \\ M_n \end{bmatrix} = \begin{bmatrix} f[x_0, x_1] - f'(x_0) \\ f[x_1, x_2] - f[x_0, x_1] \\ \vdots \\ f[x_{n-1}, x_n] - f[x_{n-2}, x_{n-1}] \\ f'(x_n) - f[x_0, x_1] \end{bmatrix}$$

Error analysis leads to this theorem:

Theorem 3. *If $f \in C^4[a, b]$, knots are defined as $a = x_0 < x_1 < \cdots < x_n = b$, $h = \max_{0 \le j \le n-1}(x_{j+1} - x_j)$, and $s_c(x)$ is the clamped cubic spline for f, then for $x \in [a, b]$ and constants c_j, $j = 0, 1, 2$, where $c = \left(\frac{5}{384}, \frac{1}{24}, \frac{3}{8}\right)$, such that*

$$\left| f^{(j)}(x) - s_c^{(j)}(x) \right| \le c_j h^{4-j} \left\| f^{(4)} \right\|_\infty.$$

Another striking fact is that cubic spline minimize "wiggles" in the following sense.

Theorem 4. *Among all $g(x) \in C^2[a,b]$ interpolating $f(x) \in C^2[a,b]$ at the nodes $a = x_0 < x_1 < \cdots < x_n = b$, and $f'(x)$ at a and b, the choice $g(x) = s_c(x)$ minimizesf*

$$\int_a^b \left(g''(x)\right)^2 dx.$$

Natural cubic spline. Define the spline $s(x) = s_{nat}(x)$ by the two additional conditions

$$s''(x_0) = 0, \quad s''(x_n) = 0.$$

Obtain additional equations

$$M_0 = 0$$
$$M_n = 0.$$

The resulting system has symmetric positive definite coefficient matrix, from which it follows that there is a unique clamped cubic spline interpolating $f(x)$ at the knots of the spline. The system is

$$\begin{bmatrix} 1 & & & & \\ \frac{h_0}{6} & \frac{h_0+h_1}{} & \frac{h_1}{6} & & \\ & \ddots & \ddots & \ddots & \\ & & \frac{h_{n-2}}{6} & \frac{h_{n-2}+h_{n-1}}{} & \frac{h_{n-1}}{6} \\ & & & & 1 \end{bmatrix} \begin{bmatrix} M_0 \\ M_1 \\ \vdots \\ M_{n-1} \\ M_n \end{bmatrix} = \begin{bmatrix} 0 \\ f[x_1,x_2] - f[x_0,x_1] \\ \vdots \\ f[x_{n-1},x_n] - f[x_{n-2},x_{n-1}] \\ 0 \end{bmatrix}.$$

One can obtain similar error bounds as with clamped cubic splines, but of course not as good.

"Not-a-knot" cubic spline. Define the spline $s(x) = s_{nk}(x)$ to be the cubic interpolating spline at knots $x_0$by the two additional conditions

$$s(x_1) = f(x_1), \quad s(x_{n-1}) = f(x_{n-1}).$$

One obtain two additional equations which again give a linear system with a unique solution by evaluating the expression for $s(x)$ (in terms of moments M_j) at the nodes x_1 and x_{n-1}. In the limit, as $x_1 \to x_0$ and $x_{n-1} \to x_n$, the not-a-knot cubic spline tends to the clamped cubic spline.

As a final example, estimate error on this function

$$f(x) = \left(\frac{1}{10}x^3 - x^2 + x\right)\sin(x), \quad 0 \le x \le 2\pi$$

using the various cubic splines. Tools for constructing these three splines are the function files CCSpp.m, NCSpp.m and NKCSpp.m. These are located in the file PPfcns.m, but they are also unpacked along with all the other functions from PPfcns.m in the folder PPfncs to be found in Course Materials.

For example, here is how to construct the natural cubic spline for $f(x)$ and plot it along with the plot of $f(x)$. You might find it more helpful to plot the error. This could be helpful for deciding how to add knots or change their location. First edit 'myfcn.m' to the following:

```
retval = (0.1*x.^3 - x.^2 + x).*sin(x);
```

Then do the following in Matlab:

```
knots = linspace(0, 2*pi, 6);
pp = NCSpp(knots,myfcn(knots));
```

```
x = 0:0.001:2*pi;
plot(x, myfcn(x))
hold on, grid
plot(x, PPeval(pp,x))
```

References

[1] M.J.D. Powell, *Approximation Theory and Methods*, Cambridge University Press, Cambridge, 1981.
[2] T. Shores, *Applied Linear Algebra and Matrix Analysis*, Springer, New York, 2007.