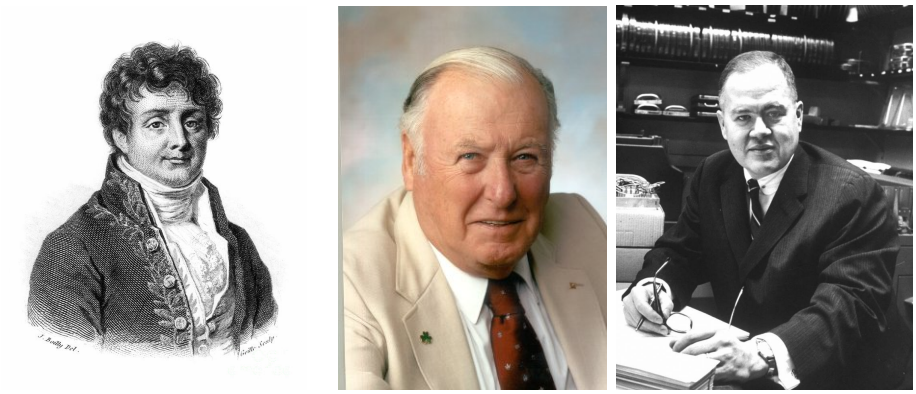
Math/CS 466/666: Lecture 2

The Fourier transform was originally developed by Joseph Fourier [3] for the study of heat transfer and vibrations. Fourier transforms are currently used in the study of differential equations, approximation theory, quantum mechanics, time-series analysis, implementation of high precision arithmetic, digital signal processing, GPS, sound and video compression, digital telephony and encryption. The fast Fourier transform is a divide and conquer algorithm developed by Cooley and Tukey [1] to efficiently compute a discrete Fourier transform on a digital computer. In 2000 Dongarra and Sullivan listed the fast Fourier transform among the top 10 algorithms of the 20th century [2].



Joseph Fourier of École Polytechnique, James Cooley of IBM Watson Laboratories and John Tukey of Princeton University and Bell Laboratories.

The Discrete Fourier Transform

The discrete Fourier transform is given by the matrix-vector multiplication Ax where A is an $N \times N$ matrix with general term given by $a_{kl} = e^{-i2\pi kl/N}$ with $k = 0, 1, \ldots, N-1$ and $l = 0, 1, \ldots, N-1$. While standard mathematical notation for matrices and vectors use index variables which range from 1 to N, we have shifted the indices by one so that the first column and first row of A are given by $k = 0$ and $l = 0$. Shifting indices in this way is both the natural for the C programming language and the mathematics. This shifted notation for indices will be used throughout our computational study of linear algebra.

Define $\overline{A}$ to be the matrix whose entries are exactly the complex conjugates of the entries of A. Our first result is

The Fourier Inversion Theorem. *Let A be the $N \times N$ Fourier transform matrix defined above. Then*

$$A^{-1} = \frac{1}{N}\overline{A}.$$

To see why this formula is true we first prove

The Orthogonality Lemma. *Suppose $l, p \in \{0, 1, \ldots, N-1\}$, then*

$$\sum_{q=0}^{N-1} e^{i2\pi(l-p)q/N} = \begin{cases} N & \text{for } l = p \\ 0 & \text{otherwise.} \end{cases}$$

Proof of The Orthogonality Lemma. Since

$$0 \le l \le N-1 \qquad \text{and} \qquad -(N-1) \le -p \le 0,$$

then $-(N-1) \le l-p \le N-1$ and consequently

$$-2\pi\Big(1-\frac{1}{N}\Big) \le 2\pi(l-p)/N \le 2\pi\Big(1-\frac{1}{N}\Big).$$

Define $\omega = e^{i2\pi(l-p)/N}$. Since the only time $e^{i\theta} = 1$ is when θ is a multiple of 2π, we conclude that

$$\omega = 1 \qquad \text{if and only if} \qquad l = p.$$

Clearly, if $l = p$ then

$$\sum_{q=0}^{N-1} e^{i2\pi(l-p)q/N} = \sum_{q=0}^{N-1} w^q = \sum_{q=0}^{N-1} 1 = N.$$

On the other hand, if $l \neq p$ then $\omega \neq 1$. In this case,

$$\omega^N = e^{i2\pi(l-p)} = 1$$

and the geometric sum formula yields that

$$\sum_{q=0}^{N-1} e^{i2\pi(l-p)q/N} = \sum_{q=0}^{N-1} \omega^q = \frac{1-\omega^N}{1-\omega} = \frac{1-1}{1-\omega} = 0.$$

This finishes the proof of the lemma. ////

We are now ready to explain the Fourier inversion theorem.

Proof of The Fourier Inversion Theorem. Let $b = Ax$ and $c = \frac{1}{N}\overline{A}b$. Claim that $c = x$. By definition

$$b_k = \sum_{l=0}^{N-1} e^{-i2\pi kl/N} x_l \qquad \text{and} \qquad c_p = \frac{1}{N}\sum_{q=0}^{N-1} e^{i2\pi pq/N} b_q.$$

Substituting yields

$$\begin{aligned} c_p &= \sum_{q=0}^{N-1} e^{-i2\pi pq/N}\Big(\frac{1}{N}\sum_{l=0}^{N-1} e^{i2\pi ql/N} x_l\Big) = \frac{1}{N}\sum_{l=0}^{N-1}\Big\{\sum_{q=0}^{N-1} e^{i2\pi(l-p)q/N}\Big\} x_l \\ &= \frac{1}{N}\sum_{l=0}^{N-1}\begin{Bmatrix} N & \textit{for } l=p \\ 0 & \textit{otherwise} \end{Bmatrix} x_l = \frac{N}{N}x_p = x_p. \end{aligned}$$

This finishes the proof of the theorem. ////

Let's pause for a moment to implement a computer program that computes the Fourier transform and the inverse Fourier transform directly from the definitions using matrix-vector multiplication. The resulting C code looks like

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <math.h>
#include <complex.h>
#include "matrixlib.h"

void dft(int N,complex x[N],complex b[N]){
    bzero(b,sizeof(complex)*N);
    for(int k=0;k<N;k++){
        for(int l=0;l<N;l++){
            b[k]+=cexp(-I*2*M_PI*k*l/N)*x[l];
        }
    }
}
void invdft(int N,complex x[N],complex b[N]){
    bzero(b,sizeof(complex)*N);
    for(int k=0;k<N;k++){
        for(int l=0;l<N;l++){
            b[k]+=cexp(I*2*M_PI*k*l/N)*x[l]/N;
        }
    }
}

#define FTSIZE 8
complex X[FTSIZE],B[FTSIZE],C[FTSIZE];

int main(){
    for(int i=0;i<FTSIZE;i++) {
        X[i]=1.0/(i+1)+I*1.0/(FTSIZE-i);
    }
    printf("N=%d\n",FTSIZE);
    printf("X=\n"); cvecprint(FTSIZE,X);
    dft(FTSIZE,X,B);
    printf("B=\n"); cvecprint(FTSIZE,B);
    invdft(FTSIZE,B,C);
    printf("C=\n"); cvecprint(FTSIZE,C);
    return 0;
}
```

and produces the output

```
N=8
X=
(1 0.125)
(0.5 0.142857)
(0.333333 0.166667)
(0.25 0.2)
(0.2 0.25)
(0.166667 0.333333)
(0.142857 0.5)
(0.125 1)
B=
(2.71786 2.71786)
(-0.0863919 -0.208568)
(2.22045e-16 -0.583333)
(0.285647 -0.689613)
(0.634524 -0.634524)
(1.01973 -0.422384)
(1.44762 -1.27676e-15)
(1.98102 0.820565)
C=
(1 0.125)
(0.5 0.142857)
(0.333333 0.166667)
(0.25 0.2)
(0.2 0.25)
(0.166667 0.333333)
(0.142857 0.5)
(0.125 1)
```

Note that the value for c is the same as x. This is consistent with the Fourier Inversion Theorem and leads us to believe that the above code is producing correct results. Making sure the code is producing the correct answer is an important first step before any sort of optimization is attempted.

We now analyze the performance of the above simple Fourier transform code. Observe that the `dft` and `invdft` routines each consist of two loops of length N. As the loops are nested, the resulting number of operations is N^2. We will obtain a significant performance increase by changing the code to use the fast Fourier transform algorithm which only takes about $N \log N$ number of operations. Before doing this, we instrument the above slow algorithm with timing routines and also create a parallel version for an example of parallel programming. The modified code looks like

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <strings.h>
```

```
4  #include <math.h>
5  #include <complex.h>
6  #include <cilk/cilk.h>
7  #include "matrixlib.h"
8  #include "tictoc.h"
9
10 void dft(int N,complex x[N],complex b[N]){
11     bzero(b,sizeof(complex)*N);
12     for(int k=0;k<N;k++){
13         for(int l=0;l<N;l++){
14             b[k]+=cexp(-I*2*M_PI*k*l/N)*x[l];
15         }
16     }
17 }
18 void pdft(int N,complex x[N],complex b[N]){
19     bzero(b,sizeof(complex)*N);
20     cilk_for(int k=0;k<N;k++){
21         for(int l=0;l<N;l++){
22             b[k]+=cexp(-I*2*M_PI*k*l/N)*x[l];
23         }
24     }
25 }
26
27 #define FTSIZE 4096
28 complex X[FTSIZE],B[FTSIZE],C[FTSIZE];
29
30 int main(){
31     for(int i=0;i<FTSIZE;i++) {
32         X[i]=1.0/(i+1)+I*1.0/(FTSIZE-i);
33     }
34     printf("N=%d\n",FTSIZE);
35     tic();
36     dft(FTSIZE,X,B);
37     double t=toc();
38     printf("dft took %g seconds.\n",t);
39     tic();
40     pdft(FTSIZE,X,B);
41     t=toc();
42     printf("parallel dft took %g seconds.\n",t);
43     return 0;
44 }
```

To use multiple processor cores the only change needed is to add a parallel loop for the matrix multiplication denoted by `cilk_for` on line 20. On a 2.8Ghz dual-core AMD Athlon64 X2 5400+ based system the above code produces the output

```
N=4096
dft took 3.38255 seconds.
parallel dft took 1.69255 seconds.
```

For this system, a factor-of-two performance increase is obtained when switching from one to two CPUs. The same program when run on a 2.4Ghz twelve-core Intel Xeon E5-2620 based system produces the output

```
N=4096
dft took 1.97895 seconds.
parallel dft took 0.191574 seconds.
```

In this case the performance increase was about 10.3 times faster, which is 86 percent of the optimal 12 times speedup. One reason a program may not scale linearly is because speed of access to main memory does not increase with the number of cores. Even though there are more cores available to perform the calculation, efficiency can eventually be limited by the read-write speed of main memory. Runtime overhead related to scheduling the parallel `cilk_for` loop on the multiple cores can also affect parallel efficiency. If you are following these lectures hands-on, please check how well the parallel code scales to multiple cores on your own computer.

The Fast Fourier Transform

While a factor ten speedup was easy to obtain by parallelizing the slow algorithm, in the case of the Fourier transform much more significant gains can be achieved by using a conquer and divide approach. This is possible because the matrix A corresponding to the Fourier transform has a significant number of symmetries in it based on the factors of the length N of the transform. For simplicity we will assume that $N = 2^n$ for some positive integer n. Thus, N is divisible by 2 and we can write $2K = N$. It follows that

$$\begin{aligned}\sum_{l=0}^{N-1} e^{-i2\pi kl/N} x_l &= \sum_{l \text{ even}} e^{-i2\pi kl/N} x_l + \sum_{l \text{ odd}} e^{-i2\pi kl/N} x_l \\ &= \sum_{p=0}^{K-1} e^{-i2\pi kp/K} x_{2p} + e^{-i2\pi k/N} \sum_{p=0}^{K-1} e^{-i2\pi kp/K} x_{2p+1}\end{aligned}$$

Note that the original Fourier transform of size N has been rewritten as two smaller Fourier transforms of size K which then need to be combined. The combining is done by multiplying the second transform by the factor $e^{-i2\pi k/N}$ for $k = 0, 1, \ldots, N-1$ which results in N additional multiplications. Therefore, the total number of operations has been reduced to

$$K^2 + N + K^2 = 2\Big(\frac{N}{2}\Big)^2 + N = \frac{1}{2}N^2 + N$$

which is a reduction of almost half the original N^2.

We are now ready to prove

The Fast Fourier Transform Theorem. *Suppose $N = 2^n$, then the Fourier transform can be computed in $N(1 + \log_2 N)$ number of operations.*

Proof of The Fast Fourier Transform Transform Theorem. Since $K = 2^{n-1}$ then K is either equal to 1 or again divisible by 2. In the case K is divisible by 2 we compute the resulting Fourier transforms of size K by further dividing them into Fourier transforms of size $K/2$. Continue dividing the resulting Fourier transforms into smaller ones until a Fourier transform of size 1 is reached which is then trivial. For example, after the second division the number of operations is given by

$$\begin{aligned}\Big\{\Big(\frac{K}{2}\Big)^2 + K + \Big(\frac{K}{2}\Big)^2\Big\} + N + \Big\{\Big(\frac{K}{2}\Big)^2 + K + \Big(\frac{K}{2}\Big)^2\Big\} \\ = 4\Big(\frac{K}{2}\Big)^2 + 2K + N = 4\Big(\frac{N}{4}\Big)^2 + 2N = 2^2\big(2^{n-2}\big)^2 + 2\cdot 2^n.\end{aligned}$$

After, the third division each transform of size $K/2$ would then computed using transforms of size $K/4$. This results in

$$8\Big(\frac{K}{4}\Big)^2 + 3N = 2^3\big(2^{n-3}\big)^2 + 3\cdot 2^n$$

number of operations. This divide and conquer process may be continued n times and then yields an algorithm that takes

$$2^n\big(2^{n-n}\big)^2 + n\cdot 2^n = 2^n(1+n) = N(1+\log_2 N)$$

number of operations. ////

We remark that $N(1+\log_2 N)$ number of operations can be much smaller than N^2 when N is large. When $N = 4096$, as used for our previous numerical test, it follows that

$$N(1+\log_2 N) = 53248 \qquad \text{and} \qquad N^2 = 16777216.$$

Since $16777216/53248 \approx 315$, using the fast Fourier transform has the performance advantage of about 315 additional processor cores when $N = 4096$. For larger values of N the advantages are even more pronounced. When $N = 65536$ the slow algorithm takes an impractically long time; for values of N corresponding to vectors that are sized to the limits of available memory, the fast algorithm is the only way to complete the computation.

We finish by presenting a recursive routine to compute the fast Fourier transform. The code

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <strings.h>
4 #include <math.h>
5 #include <complex.h>
6 #include "matrixlib.h"
7
8 void fft(int N,int s,complex x[],complex b[]){
9     if(N==1) {
```

```
10          b[0]=x[0];
11          return;
12      }
13      if(N%2){
14          printf("N not divisible by 2!\n");
15          exit(1);
16      }
17      int K=N/2;
18      fft(K,2*s,&x[0],&b[0]);
19      fft(K,2*s,&x[s],&b[K]);
20      for(int k=0;k<K;k++){
21          complex even=b[k],odd=b[k+K];
22          complex w=cexp(-I*2*M_PI*k/N);
23          b[k]=even+w*odd;
24          b[k+K]=even-w*odd;
25      }
26  }
27
28  #define FTSIZE 8
29  complex X[FTSIZE],B[FTSIZE];
30
31  int main(){
32      for(int i=0;i<FTSIZE;i++) {
33          X[i]=1.0/(i+1)+I*1.0/(FTSIZE-i);
34      }
35      printf("N=%d\n",FTSIZE);
36      fft(FTSIZE,1,X,B);
37      printf("fft_B=\n"); cvecprint(FTSIZE,B);
38      return 0;
39  }
```

produces the output

```
N=8
fft_B=
(2.71786 2.71786)
(-0.0863919 -0.208568)
(0 -0.583333)
(0.285647 -0.689613)
(0.634524 -0.634524)
(1.01973 -0.422384)
(1.44762 5.55112e-17)
(1.98102 0.820565)
```

Compare the output for the slow routine to the fast routine. When optimizing an computer program it is important to compared results produced by the optimized code to known

correct results. The fact that the output is the same in this case, suggests that the optimized code performs the same calculation as the original program. Although one test case—or even a hundred—would not be sufficient guarantee two different algorithms always produce the same results, such testing is useful and can catch many errors.

For now, we assume the code is correct and proceed to check performance by instrumenting the code with timing routines and also creating a parallel version as was done for the slow Fourier transform. The modified code looks like

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <strings.h>
4  #include <math.h>
5  #include <complex.h>
6  #include <cilk/cilk.h>
7  #include "matrixlib.h"
8  #include "tictoc.h"
9
10 void fft(int N,int s,complex x[],complex b[]){
11     if(N==1) {
12         b[0]=x[0];
13         return;
14     }
15     if(N%2){
16         printf("N not divisible by 2!\n");
17         exit(1);
18     }
19     int K=N/2;
20     fft(K,2*s,&x[0],&b[0]);
21     fft(K,2*s,&x[s],&b[K]);
22     for(int k=0;k<K;k++){
23         complex even=b[k],odd=b[k+K];
24         complex w=cexp(-I*2*M_PI*k/N);
25         b[k]=even+w*odd;
26         b[k+K]=even-w*odd;
27     }
28 }
29 void pfft(int N,int s,complex x[],complex b[]){
30     if(s>16) {
31         fft(N,s,x,b);
32         return;
33     }
34     if(N==1) {
35         b[0]=x[0];
36         return;
37     }
```

```
38      if(N%2){
39          printf("N not divisible by 2!\n");
40          exit(1);
41      }
42      int K=N/2;
43      cilk_spawn pfft(K,2*s,&x[0],&b[0]);
44      pfft(K,2*s,&x[s],&b[K]);
45      cilk_sync;
46      cilk_for(int k=0;k<K;k++){
47          complex even=b[k],odd=b[k+K];
48          complex w=cexp(-I*2*M_PI*k/N);
49          b[k]=even+w*odd;
50          b[k+K]=even-w*odd;
51      }
52  }
53
54  #define FTSIZE 524288
55  complex X[FTSIZE],B[FTSIZE],C[FTSIZE];
56
57  int main(){
58      for(int i=0;i<FTSIZE;i++) {
59          X[i]=1.0/(i+1)+I*1.0/(FTSIZE-i);
60      }
61      printf("N=%d\n",FTSIZE);
62      tic();
63      fft(FTSIZE,1,X,B);
64      double t=toc();
65      printf("fft took %g seconds.\n",t);
66      tic();
67      pfft(FTSIZE,1,X,B);
68      t=toc();
69      printf("parallel fft took %g seconds.\n",t);
70      return 0;
71  }
```

For the parallel code `cilk_spawn` in line 43 schedules one of the recursive calls to compute a smaller Fourier transform in a separate worker thread while the current thread recursively computes the other Fourier transform. The `cilk_sync` on line 45 makes sure both of the recursive calls have completed before the results are combined with a parallel loop in line 46. After `pfft` recurses 5 times, the stride given by `s` is equal 32 and 32 parallel tasks have been created to perform the computation. At this point a single call is made to the serial fast Fourier transform, because none of the available computers have more than 32 cores. While switching to the serial algorithm is not strictly necessary, doing so helps reduce scheduling overhead. On a 2.8Ghz dual-core AMD Athlon64 X2 5400+ based system the above code produces the output

```
N=524288
fft took 0.941795 seconds.
parallel fft took 0.456732 seconds.
```

We note that the fast Fourier transform performed a transform of size $N = 524288$ faster than the slow algorithm could handle a transform of size $N = 4906$. Again the factor of two parallel speedup occurs when working computing two cores. The same program when run on a 2.4Ghz twelve-core Intel Xeon E5-2620 based system produces the output

```
N=524288
fft took 0.548995 seconds.
parallel fft took 0.087564 seconds.
```

In this case the performance increase was about 6.2 times faster, which is only 52 percent of the optimal 12 times speedup. We observe that the memory access patterns of the fast Fourier transform involve recursively skipping by odd and even indexes. In particular the fast Fourier transform does not access memory sequentially. It is likely that this creates additional pressure on the memory subsystem which limits parallel performance. Again, if you are following these lectures hands-on, please check how well the parallel code scales to multiple cores on your own computer.

References

1. James Cooley and John Tukey, An algorithm for the machine calculation of complex Fourier series, *Math. Comput.*, Vol. 19, 1965.
2. Jack Dongarra and Francis Sullivan, Top Ten Algorithms of the Century, *Computing in Science and Engineering*, 2000.
3. Joseph Fourier, Théorie analytique de la chaleur, *Firmin Didot Père at Fils*, 1822.