Homework 1 on the Fourier Transform Answer Key

1. The divide-and-conquer step described in the lecture notes splits the terms of the sum for the discrete Fourier transform into odd and even terms. Construct a similar equation for use when $N = 3^n$ that divides the sum into three parts such that k (the index in the original sum) divided by 3 has remainder 0, 1 or 2.

Suppose $3K = N$ then a similar equation that divides the discrete Fourier transform into three parts is

$$\begin{aligned}
\sum_{l=0}^{N-1} e^{-i2\pi kl/N} x_l &= \Big(\sum_{l\equiv 0 \,(\mathrm{mod}\, 3)} + \sum_{l\equiv 1 \,(\mathrm{mod}\, 3)} + \sum_{l\equiv 2 \,(\mathrm{mod}\, 3)} \Big) e^{-i2\pi kl/N} x_l \\
&= \sum_{p=0}^{K-1} e^{-i2\pi (3p)l/N} x_{3p} + \sum_{p=0}^{K-1} e^{-i2\pi (3p+1)l/N} x_{3p+1} + \sum_{p=0}^{K-1} e^{-i2\pi (3p+2)l/N} x_{3p+2} \\
&= \sum_{p=0}^{K-1} e^{-i2\pi pl/K} x_{3p} + e^{-i2\pi l/N} \sum_{p=0}^{K-1} e^{-i2\pi pl/K} x_{3p+1} + e^{-i4\pi l/N} \sum_{p=0}^{K-1} e^{-i2\pi pl/K} x_{3p+2}.
\end{aligned}$$

Note that the last three sums are exactly Fourier transforms of size $N/3 = K$ for the vectors with components x_{3p}, x_{3p+1} and x_{3p+2} respectively.

2. Let $z = a + bi$ and $w = u + iv$ be complex numbers. It takes four real-valued multiplications when using the foil method to find the product zw. Look up fast complex multiplication, describe it and explain how many real-valued multiplications the fast algorithm uses to find zw.

For reference, note that by the foil method we have

$$zw = (a + bi)(u + iv) = au - bv + i(av + bu).$$

Now, set $M = a + b$ and $N = u + v$ and form the three products

$$\eta_1 = au, \qquad \eta_2 = MN \qquad \text{and} \qquad \eta_3 = bv.$$

Since

$$\eta_2 = (a + b)(u + v) = au + av + bu + bv = \eta_1 + av + bu + \eta_3,$$

it follows that

$$zw = \eta_1 - \eta_3 + i(\eta_2 - \eta_1 - \eta_3).$$

Therefore, fact complex multiplication finds the product zw using only three multiplications. It should be noted, however, that the number of additions and subtractions have increased from two to five.

Suppose α is the effort to perform either addition or subtraction and β is the computational effort to multiply two real floating-point numbers. For fast complex multiplication to be faster than the foil method we must have

$$5\alpha + 3\beta \le 2\alpha + 4\beta \qquad \text{or equivalently} \qquad 3\alpha \le \beta.$$

Therefore, if on any particular CPU architecture it happens that floating point multiplication is less than three times as long as an addition and subtraction, then the foil method would actually be faster.

3. Compute the number of real-valued double-precision floating-point multiplications and additions needed to perform a fast Fourier transform of length $N = 3^n$ using the divide-and-conquer step developed in your answer to Question 1. Explain your reasoning and how you counted the total number of operations. How many evaluations of the exponential function are performed?

As the only reasonable way to add or subtract complex numbers takes two real additions or subtractions we count all complex additions and subtractions as two of the corresponding real operations. In the case of the divide an conquer algorithm, two complex additions are needed to add the three terms together. As these additional need to be performed for $l = 0, \ldots, N-1$ that amounts to $2N$ complex additions, or $4N$ real additions.

We next count the number of exponentials. Each of the exponentials

$$e^{-i2\pi l/N} \qquad \text{and} \qquad e^{-i4\pi l/n} \qquad \text{for} \qquad l = 0, \ldots, N-1$$

are needed to combine the three transforms of length K. This appears to require the evaluation of $2N$ exponential functions; however, there one can use the algebraic properties of the exponential to reduce that count at the expense of additional complex multiplications. In particular, it would be possible to repeatedly multiply $e^{-i\pi/N}$ by itself the required number of times to obtain any of the desired exponentials. We do not go to this extreme, because such a procedure would result in an unacceptable increase in rounding error.

Instead, note that the identity

$$e^{-4\pi l/N} = (e^{-2\pi l/N})^2$$

allows us to replace half of these exponential evaluations by squares. Moreover, by nesting the multiplication in the last two terms of the divide and conquer method as

$$\cdots + e^{-i2\pi l/N}\Big(\sum_{p=0}^{K-1} e^{-i2\pi pl/K} x_{3p+1} + e^{-i2\pi l/N} \sum_{p=0}^{K-1} e^{-i2\pi pl/K} x_{3p+2}\Big),$$

see also the Julia code in question 6, one doesn't actually need to compute the squares ahead of time. They can be done free. Now, since

$$e^{-i2\pi(l+K)/N} = e^{-i2\pi l/N} e^{-i2\pi/3} \qquad \text{and} \qquad e^{-i2\pi(l+2K)/N} = -e^{-i2\pi l/N} e^{-i\pi/3},$$

then two out of every three of the remaining exponential functions can also be replaced by another complex multiplication. This leaves K exponential functions to evaluate at each step plus $2K$ complex multiplications.

Before continuing our analysis of the exponential note the relation

$$e^{-i\pi/3} = \overline{e^{-i2\pi/3}}$$

implies there could be additional savings of the underlying real operations needed for some of the complex multiplications described above. For simplicity these are not considered here. To finish our analysis of the exponential, note that a real multiplication is performed inside each exponential to multiply $2\pi/N$ by l where $2\pi/N$ needs to be formed only once. This yields K real multiplications and 1 division.

The only thing left are the $2N$ complex multiplications needed to multiply the exponentials by the results of the length K transforms before adding.

Let α denote real addition and subtraction, β denote real multiplication, γ denote the exponential of an imaginary argument, δ division of a real number and μ complex multiplication. Using this notation we may express the total number of operations of each type sufficient to perform a discrete Fourier transform of length N as

$$T_N = c_0\alpha + c_1\beta + c_2\gamma + c_3\delta + c_4\mu \qquad \text{for some} \qquad c_j \in \mathbf{N}.$$

The conquer and divide analysis performed above then implies

$$\begin{aligned} T_{3^n} &\leq 3T_{3^{n-1}} + 4\cdot 3^n\alpha + 3^{n-1}\gamma + 2\cdot 3^{n-1}\mu + 3^{n-1}\beta + \delta + 2\cdot 3^n\mu \\ &\leq 3T_{3^{n-1}} + 3^{n-1}(12\alpha + \beta + \gamma + 11\mu) + \delta. \end{aligned}$$

By induction it follows that

$$\begin{aligned} T_{3^n} &\leq 3\big(3T_{3^{n-2}} + 3^{n-2}(12\alpha + \beta + \gamma + 8\mu) + \delta\big) + 3^{n-1}(12\alpha + \beta + \gamma + 8\mu) + \delta \\ &= 3^2T_{3^{n-2}} + 2\cdot 3^{n-1}(12\alpha + \beta + \gamma + 8\mu) + (3+1)\delta \\ &\leq 3^3T_{3^{n-3}} + 3\cdot 3^{n-1}(12\alpha + \beta + \gamma + 8\mu) + (3^2+3+1)\delta \\ &\leq 3^nT_{3^0} + n\cdot 3^{n-1}(12\alpha + \beta + \gamma + 8\mu) + (3^{n-1}+\cdots+3^2+3+1)\delta. \end{aligned}$$

Since the transform of length 1 is the identity operation, then $T_{3^0} = 0$. Further summing the geometric series in front of δ yields

$$3^{n-1} + \cdots + 3^2 + 3 + 1 = (3^n - 1)/2.$$

Therefore, recalling that $N = 3^n$ and $\log_3 N = n$, we obtain

$$T_N \leq \frac{N\log_3 N}{3}(12\alpha + \beta + \gamma + 8\mu) + \frac{N-1}{2}\delta.$$

At this point one can choose whether the complex multiplications are computed using the foil method or by the fast multiplication described in question 2. For the former we have $\mu = 2\alpha + 4\beta$, which then implies

$$T_N \leq \frac{N\log_3 N}{3}(28\alpha + 33\beta + \gamma) + \frac{N-1}{2}\delta.$$

For the latter we have $\mu = 5\alpha + 3\beta$, which alternatively implies

$$T_N \leq \frac{N\log_3 N}{3}(52\alpha + 16\beta + \gamma) + \frac{N-1}{2}\delta.$$

Given the floating point hardware present in modern processors, the speed difference for the double-precision arithmetic used in most scientific computation between multiplication and addition not great enough such that the fast algorithm provides any advantage. We therefore conclude a reasonable upper bound on the number of operations needed to perform a fast Fourier transform of length $N = 3^n$ may be summarized as

Operation	Number
Real Additions and Subtractions	$28N(\log_3 N)/3$
Real Multiplications	$11N(\log_3 N)$
Exponential Evaluations	$N(\log_3 N)/3$
Real Divisions	$(N-1)/2$

On the other hand, if Fourier transforms computed using multi-precision arithmetic consisting of thousands of digits were needed, then the bound and corresponding algorithm given by fast multiplication could result in significant computational savings.

4. Install the FFTW library `https://github.com/JuliaMath/FFTW.jl` in Julia and verify that it produces the same answers as the `myfft` routine we wrote in class. Test both routines for vectors of length $N = 2^n$ for at least 5 different vectors.

The program I wrote computes the Fourier transform for randomly generated vectors of size $N = 2^n$ for $n = 7, \ldots, 11$ using `myfft` and the `fft` routine from FFTW. The norm of the difference of the two results is computed to make sure the results are consistent. The program was

```
function myfft(x)
    N=length(x)
    if N == 1
        return x
    end
    if N % 2 == 1
        println("N = ",N," was not even!")
        throw(ArgumentError)
    end
    ye=myfft(x[1:2:N-1])
    yo=myfft(x[2:2:N])
    M = N ÷ 2
    y = Array{Complex}(undef,N)
    for l = 1:M
        z0 = ye[l]
        z1 = exp(-1im*2*pi*(l-1)/N)*yo[l]
        y[l] = z0 + z1
        y[l+M] = z0 - z1
    end
    return y
end
```

```
22
23 using FFTW,LinearAlgebra,Printf
24
25 @printf("#%7s %22s\n","N","|myfft-fft|")
26 for n=7:11
27     N=2^n
28     x=rand(N)
29     myxhat=myfft(x)
30     xhat=fft(x)
31     e=norm(myxhat-xhat)
32     @printf("%8d %22.14e\n",N,e)
33 end
```

and the output

```
#      N            |myfft-fft|
     128   9.95835474053927e-15
     256   2.81419638000483e-14
     512   6.02166536198006e-14
    1024   1.31078003116496e-13
    2048   2.75922888706280e-13
```

which implies to within rounding error that the two routines produce the same results.

5. Modify the program `fft2.jl` so that it compares the speed of FFTW to the `myfft` routine. How much faster is FFTW compared to `myfft`? Does the difference in speed become more or less noticeable as the vector length N increases? If known, please provide details about the computer used, for example what model CPU it has.

A program was created to compare the relative speed of `myfft` with `fft` using Fourier transforms of sizes $N = 2^n$ where $n = 5, \ldots, 18$. The Julia code is

```
1  function myfft(x)
2      N=length(x)
3      if N == 1
4          return x
5      end
6      if N % 2 == 1
7          println("N = ",N," was not even!")
8          throw(ArgumentError)
9      end
10     ye=myfft(x[1:2:N-1])
11     yo=myfft(x[2:2:N])
12     M = N ÷ 2
13     y = Array{Complex}(undef,N)
14     for l = 1:M
15         z0 = ye[l]
```

```
        z1 = exp(-1im*2*pi*(l-1)/N)*yo[l]
        y[l] = z0 + z1
        y[l+M] = z0 - z1
    end
    return y
end

using FFTW,LinearAlgebra,Printf

@printf("#%11s %22s %22s %10s\n","N","T(myfft)","T(fft)","speedup")
for n=5:18
    N=2^n
    myelap=0; elap=0
    for j=1:10
        x=rand(N)
        start=time(); myxhat=myfft(x); dt=time()-start
        if dt<myelap || myelap==0
            myelap=dt
        end
        start=time(); xhat=fft(x); dt=time()-start
        if dt<elap || elap==0
            elap=dt
        end
        if norm(myxhat-xhat) > 1e-7
            println("Difference too large!")
            throw(ArgumentError)
        end
    end
    @printf("%12d %22.14e %22.14e %10.4f\n",
        N,myelap,elap,myelap/elap)
end
```

with output

```
#          N                T(myfft)                 T(fft)    speedup
          32   4.00543212890625e-05   1.90734863281250e-05     2.1000
          64   9.58442687988281e-05   2.88486480712891e-05     3.3223
         128   2.00033187866211e-04   3.48091125488281e-05     5.7466
         256   4.35829162597656e-04   3.71932983398438e-05    11.7179
         512   9.72986221313477e-04   4.79221343994141e-05    20.3035
        1024   2.05707550048828e-03   6.69956207275391e-05    30.7046
        2048   4.58717346191406e-03   1.04904174804688e-04    43.7273
        4096   1.00038051605225e-02   1.80959701538086e-04    55.2819
        8192   2.16269493103027e-02   4.06980514526367e-04    53.1400
       16384   5.11789321899414e-02   7.43865966796875e-04    68.8013
```

```
 32768   1.12089157104492e-01   1.52897834777832e-03    73.3098
 65536   2.63868093490601e-01   3.16214561462402e-03    83.4459
131072   6.67751073837280e-01   7.75885581970215e-03    86.0631
262144   1.56996893882751e+00   5.86919784545898e-02    26.7493
```

Here the times `T(myfft)` and `T(fft)` were measured in seconds. The FFTW routine `fft` varies from 2.1 to 86.0631 times faster than the routine `myfft` that we wrote in class. As the size of the transform increases the difference in speed becomes more noticeable, except for the last transform of size 252144 where the relative difference in speed between the two methods decreases to the factor 26.7493. For reference the computer used was an ACER C720 Chromebook with a 1.4GHz Intel Celeron 2955U processor.

6. [Extra Credit] Modify the code we wrote in class to perform fast Fourier transforms of length $N = 2^p 3^q$ where p and q are both non-negative integers. Do this by selecting the appropriate divide and conquer formula when dividing by the corresponding prime. Comparing the output of your routine with FFTW. Does it make any difference with regards to speed or rounding error if the powers of 3 or 2 are divided out first?

The program

```
function myfft(x)
    N=length(x)
    if N == 1
        return x
    end
    y = Array{Complex}(undef,N)
    if N % 2 == 0
        ye=myfft(x[1:2:N-1])
        yo=myfft(x[2:2:N])
        M = N ÷ 2
        for l = 1:M
            z0 = ye[l]
            z1 = exp(-1im*2*pi*(l-1)/N)*yo[l]
            y[l] = z0 + z1
            y[l+M] = z0 - z1
        end
    elseif N % 3 == 0
        ym0=myfft(x[1:3:N-2])
        ym1=myfft(x[2:3:N-1])
        ym2=myfft(x[3:3:N])
        M = N ÷ 3
        for l = 1:M
            e1a = exp(-2im*pi*(l-1)/N)
            e1b = e1a*ep1m; e1c = e1a*ep2m
            z0 = ym0[l]; z1 = ym1[l]; z2 = ym2[l]
            y[l] = z0+e1a*(z1+e1a*z2)
```

```
            y[l+M] = z0+e1b*(z1+e1b*z2)
            y[l+2*M] = z0+e1c*(z1+e1c*z2)
        end
    else
        println("N = ",N," was not divisible by two or three!")
        throw(ArgumentError)
    end
    return y
end

using FFTW,LinearAlgebra,Printf

ep1m=exp(-2im*pi/3)
ep2m=-exp(-1im*pi/3)

println("#Powers of 2 divided out first...")
@printf("#%3s %4s %8s %22s\n","p","q","N","|myfft-fft|")
for p=3:7
    for q=3:7
        N=2^p*3^q
        x=rand(N)
        myxhat=myfft(x)
        xhat=fft(x)
        e=norm(myxhat-xhat)
        @printf("%4d %4d %8d %22.14e\n",p,q,N,e)
    end
end
```

produced the output

```
#Powers of 2 divided out first...
#  p    q        N            |myfft-fft|
   3    3      216   5.38426118314508e-14
   3    4      648   1.94698527790608e-13
   3    5     1944   6.64018794991560e-13
   3    6     5832   2.29776115736782e-12
   3    7    17496   7.76968246855713e-12
   4    3      432   1.04435700445245e-13
   4    4     1296   3.99681539791472e-13
   4    5     3888   1.38092017096323e-12
   4    6    11664   4.74365178037615e-12
   4    7    34992   1.57336726449478e-11
   5    3      864   2.35192834314066e-13
   5    4     2592   8.13742438133776e-13
   5    5     7776   2.82327734173326e-12
```

```
5    6    23328   9.58978864411643e-12
5    7    69984   3.21747656002058e-11
6    3     1728   4.72447281294631e-13
6    4     5184   1.80975400083728e-12
6    5    15552   6.03296925153049e-12
6    6    46656   2.08550665805578e-11
6    7   139968   6.74341569264210e-11
7    3     3456   1.01281010309223e-12
7    4    10368   3.50824622336540e-12
7    5    31104   1.17944050698531e-11
7    6    93312   4.00711646928212e-11
7    7   279936   1.34149796586094e-10
```

while the program

```
function myfft(x)
    N=length(x)
    if N == 1
        return x
    end
    y = Array{Complex}(undef,N)
    if N % 3 == 0
        ym0=myfft(x[1:3:N-2])
        ym1=myfft(x[2:3:N-1])
        ym2=myfft(x[3:3:N])
        M = N ÷ 3
        for l = 1:M
            e1a = exp(-2im*pi*(l-1)/N)
            e1b = e1a*ep1m; e1c = e1a*ep2m
            z0 = ym0[l]; z1 = ym1[l]; z2 = ym2[l]
            y[l] = z0+e1a*(z1+e1a*z2)
            y[l+M] = z0+e1b*(z1+e1b*z2)
            y[l+2*M] = z0+e1c*(z1+e1c*z2)
        end
    elseif N % 2 == 0
        ye=myfft(x[1:2:N-1])
        yo=myfft(x[2:2:N])
        M = N ÷ 2
        for l = 1:M
            z0 = ye[l]
            z1 = exp(-1im*2*pi*(l-1)/N)*yo[l]
            y[l] = z0 + z1
            y[l+M] = z0 - z1
        end
    else
```

```
        println("N = ",N," was not divisible by two or three!")
        throw(ArgumentError)
    end
    return y
end

using FFTW,LinearAlgebra,Printf

ep1m=exp(-2im*pi/3)
ep2m=-exp(-1im*pi/3)

println("#Powers of 3 divided out first...")
@printf("#%3s %4s %8s %22s\n","p","q","N","|myfft-fft|")
for p=3:7
    for q=3:7
        N=2^p*3^q
        x=rand(N)
        myxhat=myfft(x)
        xhat=fft(x)
        e=norm(myxhat-xhat)
        @printf("%4d %4d %8d %22.14e\n",p,q,N,e)
    end
end
```

produced the output

```
#Powers of 3 divided out first...
#  p    q        N            |myfft-fft|
   3    3      216   5.70324722484054e-14
   3    4      648   2.08088452810020e-13
   3    5     1944   7.05133426048045e-13
   3    6     5832   2.39157626605692e-12
   3    7    17496   8.07154531380795e-12
   4    3      432   1.12661196296191e-13
   4    4     1296   4.01462916048588e-13
   4    5     3888   1.42228808088904e-12
   4    6    11664   4.82939239144728e-12
   4    7    34992   1.64526329030565e-11
   5    3      864   2.42086717565762e-13
   5    4     2592   8.83220301218190e-13
   5    5     7776   2.98420942538996e-12
   5    6    23328   9.92146490964623e-12
   5    7    69984   3.40856827355689e-11
   6    3     1728   4.85088020422243e-13
   6    4     5184   1.73561537982483e-12
```

```
6    5    15552   6.12431957204718e-12
6    6    46656   2.03236549761285e-11
6    7   139968   6.82520778329779e-11
7    3     3456   1.02962852347354e-12
7    4    10368   3.63193611382776e-12
7    5    31104   1.23031393750261e-11
7    6    93312   4.23639756557958e-11
7    7   279936   1.40775536190812e-10
```

From the output we see dividing out by 2 first resulted in slightly less rounding error than dividing out by 3 first. Different results may be obtained depending on the way in which the exponential functions represented by `e1a`, `e1b` and `e1c` are computed.