Homework 5

February 27, 2020

1 Introduction

For this homework, you will revise and extend the DFT and FFT functions to implement inverse DFT, inverse FFT, and two versions of low-pass filters using DFT and FFT. We will reuse the type class instances of `Vec` data type.

```
newtype Vec a = Vec {runVec :: [a]}
```

1. We will use the type alias `Signal`.

   ```
   type Signal = Vec (Complex Double)
   ```

2. The following type class instances are provided for your reference.

   ```
   instance Show a => Show (Vec a) where
     show (Vec lst) = "[" ++ drop 1 lst' ++ "]"
       where lst' = mconcat $ map (\x -> " " ++ show x) lst

   instance Functor Vec where
     fmap f (Vec x) = Vec $ map f x

   instance Applicative Vec where
                         pure = Vec . repeat
         (Vec f) <*> (Vec x) = Vec $ map (uncurry ($)) $ zip f x
     liftA2 f (Vec x) (Vec y) = Vec $ map (uncurry f) $ zip x y

   instance Num a => Num (Vec a) where
               (+) = liftA2 (+)
               (-) = liftA2 (-)
               (*) = liftA2 (*)
            negate = fmap negate
               abs = fmap abs
            signum = fmap signum
     fromInteger x = pure $ fromInteger x
   ```

```
instance (Floating a) => Fractional (Vec a) where
             (/) = liftA2 (/)
  fromRational x = pure $ fromRational x

instance (Floating a) => Floating (Vec a) where
     pi = pure pi
    exp = fmap exp
    log = fmap log
    sin = fmap sin
    cos = fmap cos
   asin = fmap asin
   acos = fmap acos
   atan = fmap atan
   sinh = fmap sinh
   cosh = fmap cosh
  asinh = fmap asinh
  acosh = fmap acosh
  atanh = fmap atanh

instance Foldable Vec where
  foldr f c (Vec a) = foldr f c a

imagV :: Num a => Vec a -> Vec (Complex a)
imagV (Vec a) = Vec $ map (0:+) a

realV :: Num a => Vec a -> Vec (Complex a)
realV (Vec a) = Vec $ map (:+0) a

instance Semigroup (Vec a) where
  Vec a <> Vec b = Vec $ a++b

instance Monoid (Vec a) where
  mempty = Vec []
```

3. The following auxiliary functions are provided for your reference.

```
range :: Double -> Double -> Double -> [Double]
range from to count = map (\x -> from + x * step) [0..count-1]
  where step = (to - from)/count

absolute :: Vec (Complex Double) -> Vec Double
absolute = fmap (\(r:+i) -> sqrt(r*r + i*i))

rd :: Int -> Vec Double -> Vec Double
rd n = fmap (\x -> fromIntegral (round $ c * x) / c)
  where c = 10^n
```

```
length' (Vec x) = length x
```

4. You should implement a function to calculate the twiddle factor for DFT.

$$twiddle\ N\ k = [e^{-\frac{2\pi}{N}\cdot k\cdot 0\cdot i},\ e^{-\frac{2\pi}{N}\cdot k\cdot 1\cdot i}, \ldots,\ e^{-\frac{2\pi}{N}\cdot k\cdot (N-1)\cdot i}]$$

   You should use this function to implement `dft` function. The inverse DFT function `idft` can be implemented by calling `dft` function.

```
twiddle :: Double -> Double -> Signal

dft :: Signal -> Signal

idft :: Signal -> Signal
```

   Recall that DFT equation is

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{2\pi}{N}\cdot k\cdot n\cdot i}$$

   We can compute inverse DFT using

$$x_k = \frac{1}{N}\left[\sum_{n=0}^{N-1} X_n^* \cdot e^{-\frac{2\pi}{N}\cdot k\cdot n\cdot i}\right]^*$$

   where x^* is the conjugate of the complex number x.

   Note that `conjugate (1 :+ 2) = 1 :+ (-2)`.

5. Next you should implement a low-pass filter function using DFT and inverse DFT functions. A low-pass filter reduces noises from an input signal by removing high-frequency values. It operates in three steps: DFT, element-wise product with a mask vector, and inverse DFT. The mask vector is simply a list of zeros and ones, where ones are in the lower and upper ends of the list while zeros are in the middle.

```
mask :: Int -> Int -> Signal
mask freq n = realV $ Vec $ one ++ zero ++ one
   where one = (take freq $ repeat 1)
         zero = (take (n-freq*2) $ repeat 0)

-- mask 2 8 = Vec [1,1,0,0,0,0,1,1]
```

   Note that the actual output of `mask 2 8` is a vector of complex doubles

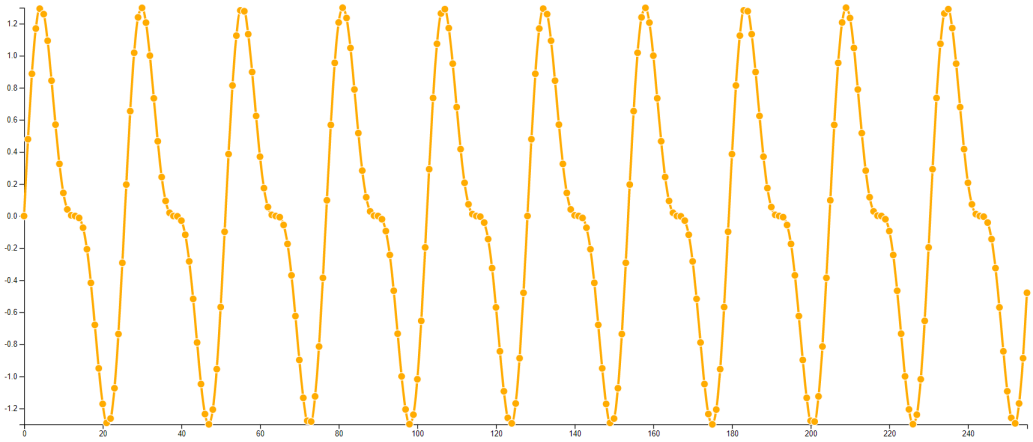
```
[1.0 :+ 0.0 1.0 :+ 0.0 0.0 :+ 0.0 0.0 :+ 0.0
 0.0 :+ 0.0 0.0 :+ 0.0 1.0 :+ 0.0 1.0 :+ 0.0]
```

You should implement a low-pass function with the following type.

```
low_pass' :: Int -> Signal -> Signal
```

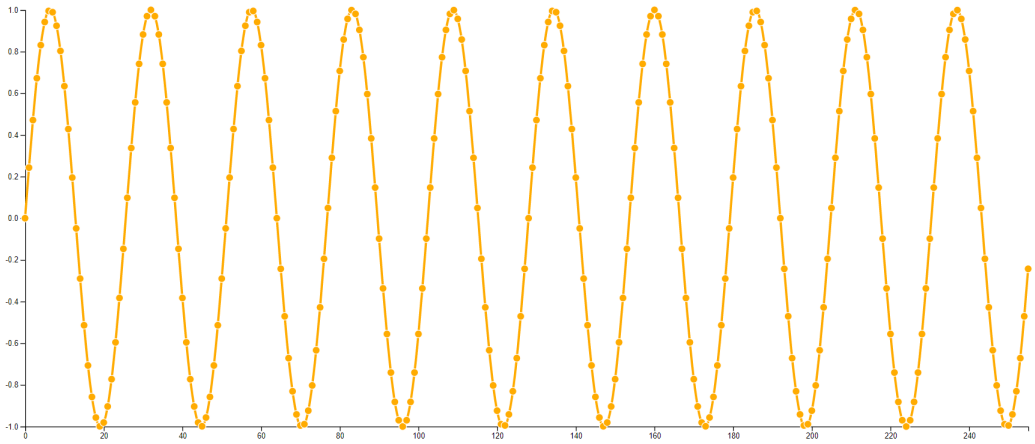
For example, we can generate a signal consisting of two sine waves of 10 and 20 Hz (shown below) with the following code (inside a do-block).

```
let n = fromIntegral 2^8
let s1 = fmap (\x -> sin(20*pi*x) + sin(40*pi*x)/2) $ Vec $ range 0 1 n
print(rd 3 s1)
```

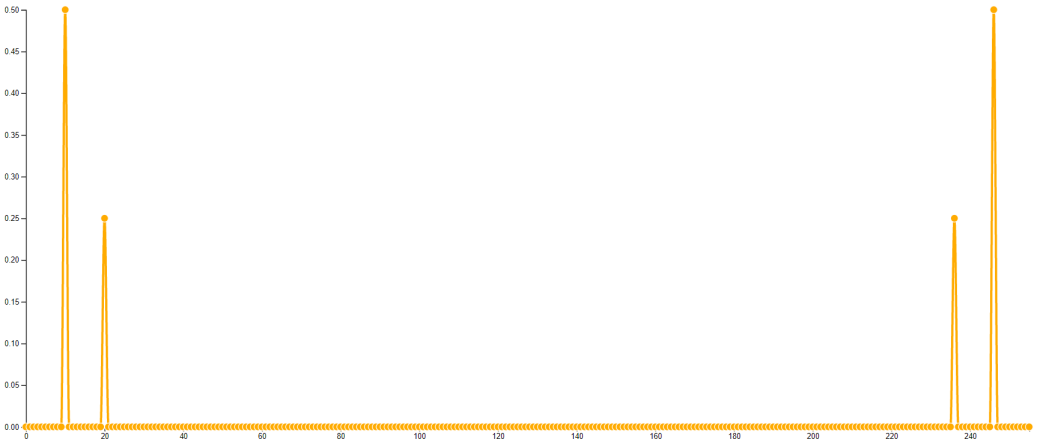


After applying a low-pass filter (cut off frequency is 15 Hz), the output signal is a pure sine wave (with only the low frequency at 10 Hz).

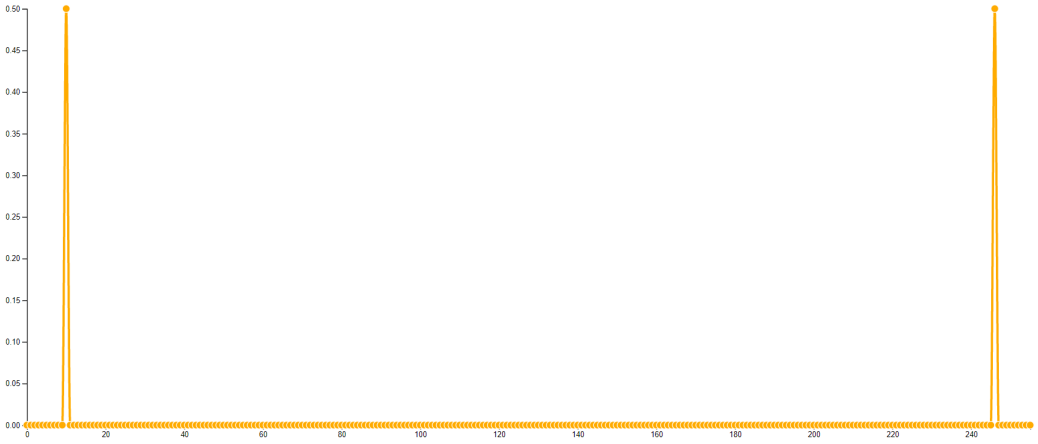
```
print(rd 3 $ fmap (\(r:+_) -> r) $ low_pass' 15 $ realV s1)
```



Note that the scale of the signal after DFT has peaks at the 10th and 20th indices (and the opposite end).



After applying the mask, the scale of the transformed sigal only has peaks at the 10th index (and the opposite end).



6. You should also implement a more efficient version of FFT that can reuse precomputed twiddle factors. Recall that FFT has the following formula:

$$\begin{aligned} X_k &= E_k + e^{-\frac{2\pi}{N} \cdot k \cdot i} \cdot O_k \\ X_{k+\frac{N}{2}} &= E_k - e^{-\frac{2\pi}{N} \cdot k \cdot i} \cdot O_k \end{aligned}$$

where E and O are the FFT of the even and odd elements of the input vector. In other words, $E = \mathsf{fft}[x_0, x_2, \ldots, x_{N-2}]$ and $O = \mathsf{fft}[x_1, x_3, \ldots, x_{N-1}]$.

For this implementation, you should have the base case so that if $N \leq 1$, then $X = x$. Notice that the twiddle factor $e^{-\frac{2\pi}{N} \cdot k \cdot i}$ can be reused

throughout the recursion. For example, for $N = 8$, at the initial call, we need the twiddle factors below:

$$[e^{-\frac{2\pi}{8}\cdot 0\cdot i},\ e^{-\frac{2\pi}{8}\cdot 1\cdot i},\ e^{-\frac{2\pi}{8}\cdot 2\cdot i},\ e^{-\frac{2\pi}{8}\cdot 3\cdot i}]$$

At the next level of recursion, we need the twiddle factors below:

$$[e^{-\frac{2\pi}{4}\cdot 0\cdot i},\ e^{-\frac{2\pi}{4}\cdot 1\cdot i}]$$

These are just the even indexed elements of the previous list.

$$[e^{-\frac{2\pi}{8}\cdot 0\cdot i},\ e^{-\frac{2\pi}{8}\cdot 2\cdot i}]$$

Therefore, we can have a more efficient implementation by precomputing the twiddle factor needed for the initial call and then pass its even-indexed half to each recursive calls. Since the inverse FFT can be implemented using fFT, they can share the twiddle factor as well. To make this more interesting, you should implement the new version of FFT and inverse FFT using Reader Monad.

Recall that Reader Monad is simply a function type, where `Reader r a` is simply a function type `Reader {runReader :: r -> a}`. The first type parameter is the information we want to read during the computation and the second type parameter is the value we want to compute. Since the twiddle factor has a Signal type and the result of FFT is a Signal value, you should implement FFT and inverse FFT with the following types.

```
fft :: Signal -> Reader Signal Signal

ifft :: Signal -> Reader Signal Signal
```

You should also implement a version of the low-pass filter using FFT and inverse FFT. Please make sure the call `runReader` and pass the precomputed twiddle factor.

```
low_pass :: Int -> Signal -> Signal
```

2 Testing

You can test the implementation using the following `main`

```
main = do
        let n = fromIntegral 2^8
        let s1 = fmap (\x -> sin(20*pi*x) + sin(40*pi*x)/2) $ Vec $ range 0 1 n
        print(rd 3 s1)

        print(rd 3 $ fmap (\(r:+_) -> r) $ low_pass' 15 $ realV s1)

        print(rd 3 $ fmap (\(r:+_) -> r) $ low_pass  15 $ realV s1)
```

You can copy/paste the output to a HTML file I provide to visualize the signal output.

3 Submission

Please write your solution in a file – `hwk5.hs` and submit it to the dropbox.