Queen's University | FACULTY OF ENGINEERING AND APPLIED SCIENCE

Lab 8: Fourier Transform

The objective of this laboratory is to become familiar with the discrete Fourier transform.

Before You Start

Remember that MATLAB can only use files, functions, and data within the current directory. **Make sure** you make a new folder for each lab and that it is sest as your current folder before you start.

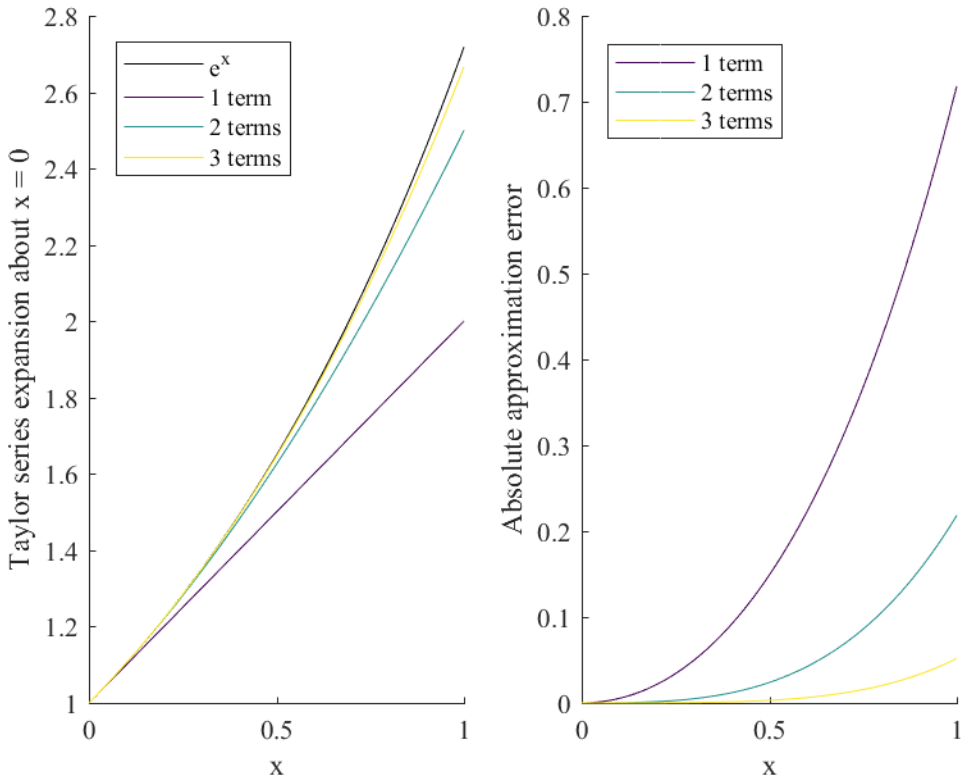
As we discussed in the Lab 2, any differentiable, continuous function $f(x)$ can be represented as a Taylor series about the point $x = a$:

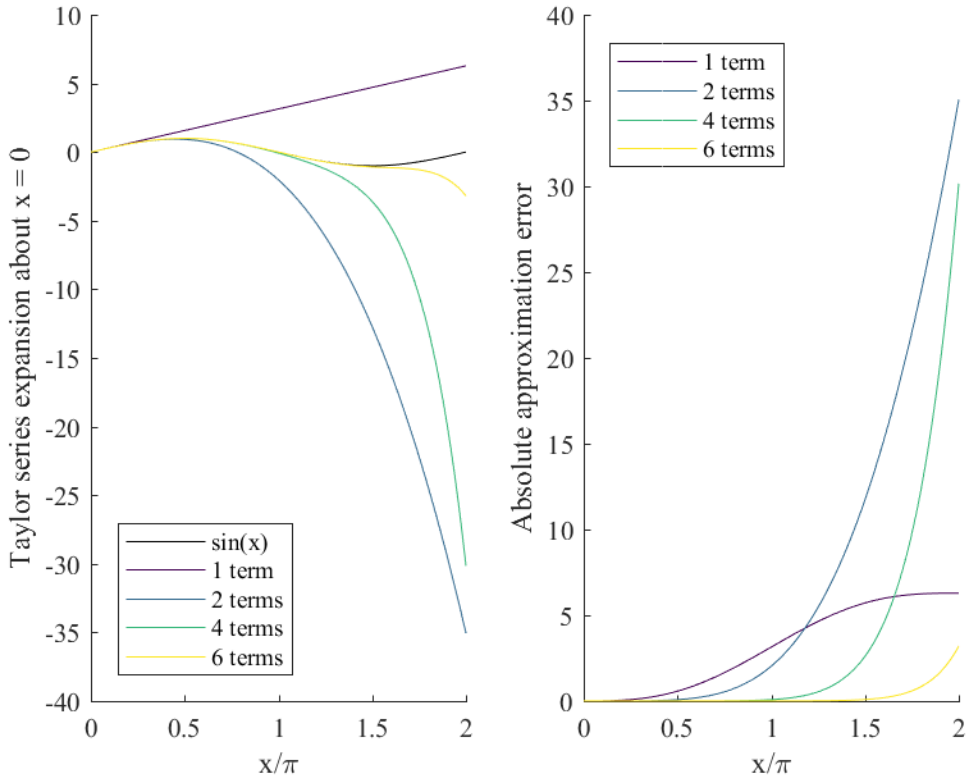
$$f(x) = f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2!}f''(a) + \cdots + \frac{(x-a)^{n-1}}{(n-1)!}f^{n-1}(a) + R_n(x)$$

where $R_n(x)$ is the remainder or error of the Taylor series expansion, given by:

$$R_n(x) = \frac{(x-a)^n}{n!}f^n(\xi)$$

where ξ lies in the range $[a, x]$. I've plotted some examples for you below:





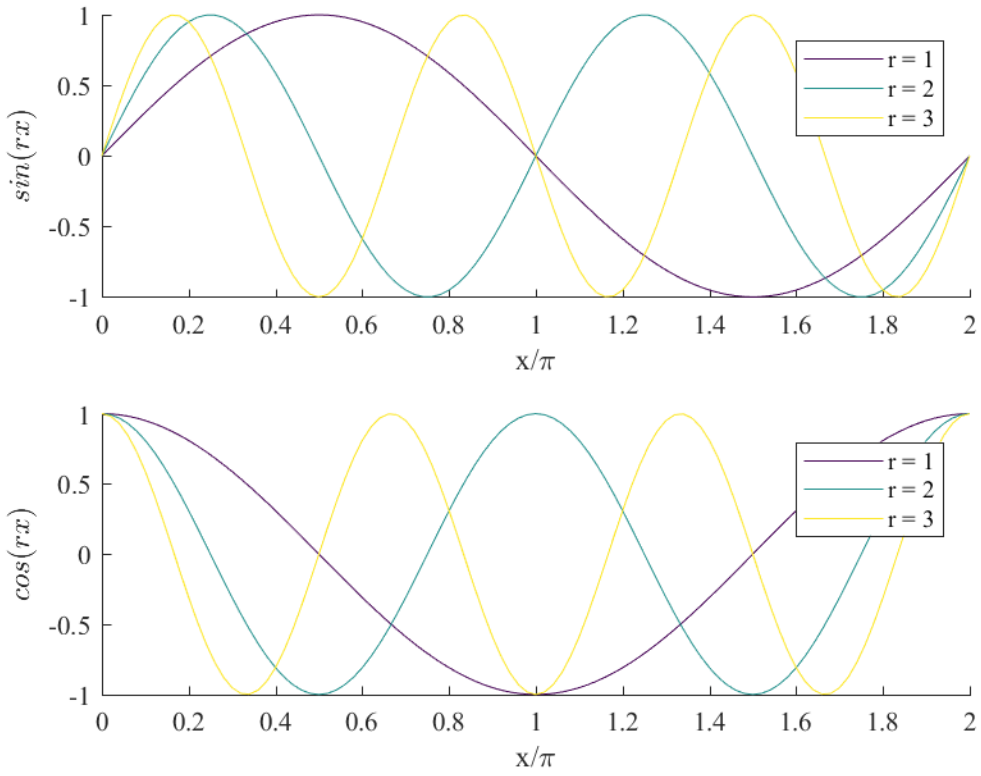
However, if your function is *not* everywhere continuous and differentiable then it can no longer be expressed as a Taylor series. Luckily, Taylor series are not the only way to represent a function. Instead of expanding a function in terms of the polynomial basis vectors:

$$(x-a), (x-a)^2, (x-a)^3, \dots (x-a)^{n-1}$$

a Fourier series can be used to expand periodic functions in terms of the sinusoidal basis vectors:

$$\sin\left(\frac{2\pi x}{L}\right), \cos\left(\frac{2\pi x}{L}\right), \dots, \sin\left(\frac{2\pi r x}{L}\right), \cos\left(\frac{2\pi r x}{L}\right), \dots, \sin\left(\frac{2\pi (n-1) x}{L}\right), \cos\left(\frac{2\pi (n-1) x}{L}\right)$$

instead, where L is the period of the function you're trying to represent. The basis vectors each have one characteristic frequency. For a period $L = 2\pi$, the first three pairs of basis functions would look like:



The Fourier expansion is then written as:

$$f(x) = \frac{a_0}{2} + \sum_{r=1}^{\infty}\left[a_r \cos\left(\frac{2\pi r x}{L}\right) + b_r \sin\left(\frac{2\pi r x}{L}\right)\right]$$

where $a0, a_r$ and b_r are called the Fourier coefficients. These can be thought of as the contribution of each basis function to the function $f(x)$ and are analogous to the coefficients $f^{(n)}(x)/n!$ from the Taylor expansion. The essential step in computing a Fourier series is the calculation of these coefficients, called a Fourier transform of $f(x)$, commonly denoted by $\mathcal{F}\big(f(x)\big)$. A discrete Fourier transform is one for which the input function $f(x)$ is sampled at a discrete number of points.

Let us assume we have N evenly spaced samples of an underlying function, $f(x)$. This means we have N knowns, allowing us to solve for N unknown Fourier coefficients:

$$a_0, a_1, a_2, \dots, a_{((N-1)/2)}, b_1, b_2, \dots, b_{((N-1)/2)}$$

Note that this assumes N is odd. If N is even, it is somewhat ambiguous as to whether you should solve for $a_{N/2}$ or $b_{N/2}$. This is addressed below. Of interest, the maximum frequency that N samples can be used to represent is the called the Nyquist limit $f_{Nyquist} = {}^{N}/_{2}$.

It can be shown that the Fourier coefficients are given by:

$$a_r = \frac{2}{L}\int_{x_0}^{x_0+L} f(x)\cos\left(\frac{2\pi r x}{L}\right)dx$$

$$b_r = \frac{2}{L}\int_{x_0}^{x_0+L} f(x)\sin\left(\frac{2\pi r x}{L}\right)dx$$

where x_0 is an arbitrary starting point. The derivation of these equations is not given here, but uses the mutual orthogonality of the basis functions.

When $f(x)$ is discretely sampled, it can be represented as a series of delta functions (also called a delta comb), replacing the integral in the above equations with a sum over the sampled points:

$$a_r = \frac{2}{N}\sum_{k=1}^{N} f(x_k)\cos\left(\frac{2\pi r x_k}{L}\right)$$

$$b_r = \frac{2}{N}\sum_{k=1}^{N} f(x_k)\sin\left(\frac{2\pi r x_k}{L}\right)$$

It can also be shown that since the function $f(x)$ is assumed to be periodic, x can be shifted to an arbitrary starting point. To make the math easier, we will shift x to start at zero:

$$x = [0, L/N, 2L/N, \ldots, kL/N, \ldots, (N-1)L/N]$$

Substituting this into our equations for a_r and b_r above we have that:

$$a_r = \frac{2}{N}\sum_{k=0}^{N-1} f(x_k)\cos\left(\frac{2\pi r k}{N}\right)$$

$$b_r = \frac{2}{N}\sum_{k=0}^{N-1} f(x_k)\sin\left(\frac{2\pi r k}{N}\right)$$

Complex Fourier Transforms

Recall the Taylor series representation of $f(x) = e^x,\ f(x) = \sin(x)$ and $f(x) = \cos(x)$ about $x = 0$:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots$$

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

If we replace x with $i * x$ where i is the imaginary number $i = \sqrt{-1}$ we have that:

$$\begin{aligned} e^{ix} &= 1 + ix - \frac{x^2}{2!} - i\frac{x^3}{3!} + \frac{x^4}{4!} + i\frac{x^5}{5!} - \frac{x^6}{6!} - i\frac{x^7}{7!} + \cdots \\ &= \left(1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots\right) + i\left(x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots\right) \\ &= \cos(x) + i\sin(x) \end{aligned}$$

This is called Euler's formula and allows us to express the Fourier series expansion from above as:

$$f(x) = \sum_{r=-\infty}^{\infty} c_r \exp\left(\frac{2\pi i r x}{L}\right)$$

The coefficients c_r are given by:

$$c_r = \frac{1}{L}\int_{x_0}^{x_0+L} f(x) \exp\left(-\frac{2\pi i r x}{L}\right) dx$$

for a continuously sampled function, and:

$$c_r = \frac{1}{N}\sum_{k=0}^{N-1} f(x_k) \exp\left(-\frac{2\pi i r k}{N}\right)$$

for a discretely sampled function.

Part A – Due at end of the lab period

Your task for Part A is to perform a Fourier transform of a discrete set of evenly spaced samples.

Task A1

Write a function `discreteFT` that uses the above equations to find the Fourier coefficients a_r and b_r. The inputs/outputs should be (in this order):

Inputs	
`f_x`	Sample points

Outputs	
`a`	Row vector of coefficients a_r
`b`	Row vector of coefficients b_r

The length of the vectors `a` and `b` should be `floor(N/2)+1`, where b_0, the first element in the `b` vector, is set to zero. Note that for the case that `N` is even, this means there are `N+1` coefficients but only `N` points. Your function should simply assign a value of zero to the $b_{(N/2)}$ coefficient in this case. Use the function **test_discreteFT** from onQ to test your function.

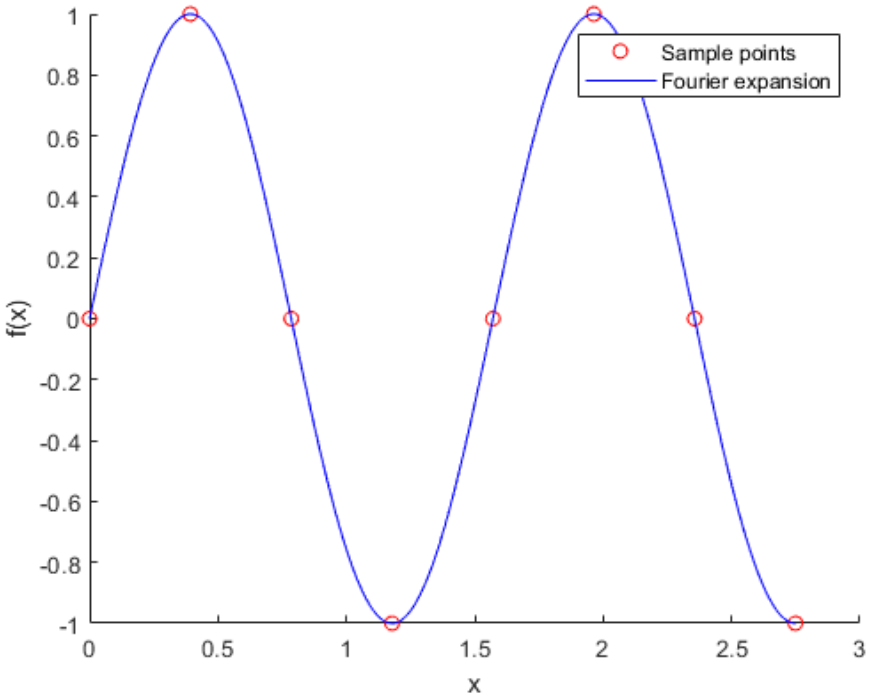
Task A2

Write a plotter function `plotDiscreteFT` that calculates the Fourier coefficient vectors for a set of sample points and plots the Fourier expansion and original points. Use 100 times the number of sample points to plot the Fourier expansion. The inputs/outputs should be (in this order):

Inputs	
`x`	Sample points (independent variable)
`f_x`	Sample points (dependent variable)

Outputs
No output variable. Output figure of recreated function sampled at 100x the data density, and the original x, f_x data points.

Use the test function **test_plotDiscreteFT** from onQ to test your function. The output should look something like this:



Task A3

Write a function `discreteCFT` that performs a complex Fourier transform. The inputs/outputs should be (in this order):

Inputs	
`f_x`	Sample points

Outputs	
`c`	Row vector of coefficients c_r

You can either use the expression for c_r given in the background or derive the relations between c_r, c_{-r}, a_r and b_r and modify your function `discreteFT`. Use the function **test_discreteCFT** from onQ to test your function.

Part A: Submission List

Please submit the following files to the **Lab 8: Part A** dropbox on onQ for marking:

- discreteFT.m
- plotDiscreteFT.m
- discreteCFT.m
- Any other functions you wrote that the above files use

Part B – Due Sunday @ 9PM

Task B1: Inverse Fourier Transform

Write a function `discreteICFT` that performs an inverse Fourier transform. The inputs/outputs should be (in this order):

Inputs	
`c`	Row vector of coefficients c_r (as calculated in Task A3)

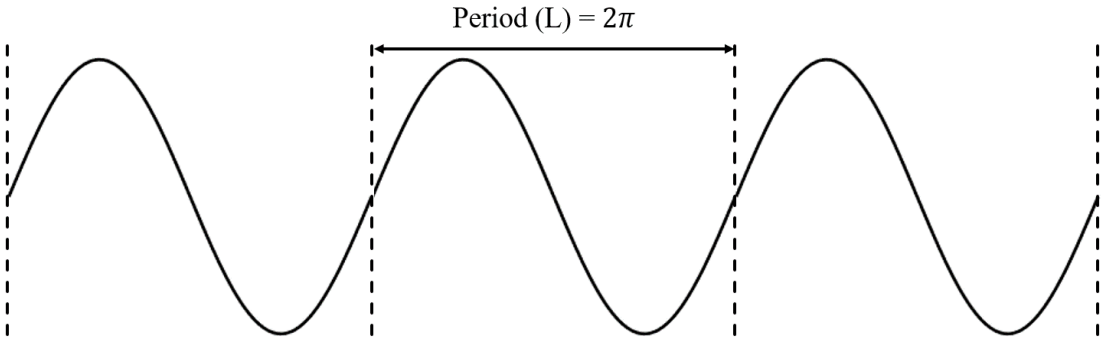
Outputs	
`f_x`	Sample points

Note: There is no test function provided for `discreteICFT`. You will either need to write your own or come up with some other way of checking that your function is behaving as expected.

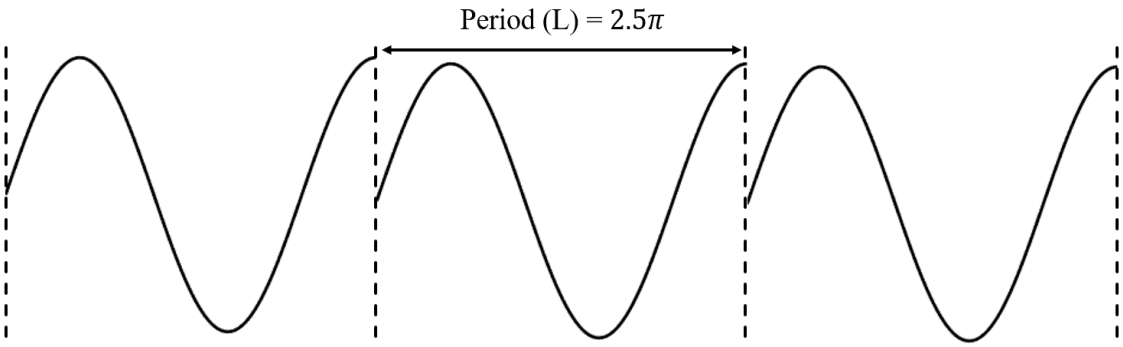
Task B2: Windowing

As you found in Part A, discrete Fourier transforms often differ from the Fourier transform of a function. The two main causes of this are aliasing and leakage. As mentioned above, Nyquist's theorem states that the maximum frequency we can represent with N samples is $f_{Nyquist} = {}^{N}/_{2}$. Any higher frequencies are effectively ignored. This is called aliasing and results from the finite sampling frequency of a discrete signal. You should have noticed this in the under sampled sine function test in Part A.

Recall that in carrying out a Fourier transform we assumed that the underlying function is periodic. To force this assumption to be true, the discrete Fourier transform algorithm you developed in Part A sums over all the sampled points, setting the "period" to be the entire sample window. If we sampled points from a sine function $f(x) = \sin(x)$ from $x = 0$ to $x = 2\pi$, the function remains as expected:



If instead we sampled points from $x = 0$ to $x = 2.5\pi$, the function now has discontinuities at the beginning and end of each period:



This corrupts our Fourier transform. Try it using your function from Part A, you should find that you get non-zero coefficients for all frequencies.

Leakage can be reduced using a technique called windowing. Windowing multiplies a signal by a window function that tapers to zero at the edges, getting rid of the discontinuities. Choosing a window function can be a complicated task and requires prior knowledge of the frequency content of the signal. However, the Hann window function has been shown to be well-suited to most signal processing tasks and is defined as:

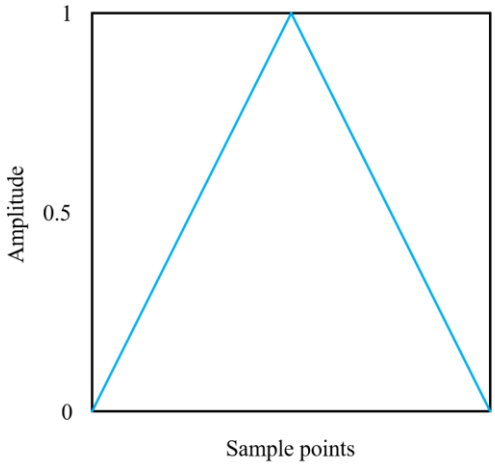
$$Hann(k) = \sin^2\left(\frac{\pi k}{N-1}\right)$$

Write a function `hannWindow` that applies the above Hann window function to a set of sample points. The inputs/outputs should be (in this order):

Inputs	
`f_x`	Sample points

Outputs	
`f_xHann`	Windowed sample points

Write a second windowing function `triangularWindow` that applies a triangular window function to a set of sample points. The window function should be of the form:



The inputs/outputs should be (in this order):

Inputs	
`f_x`	Sample points

Outputs	
`f_xTri`	Windowed sample points

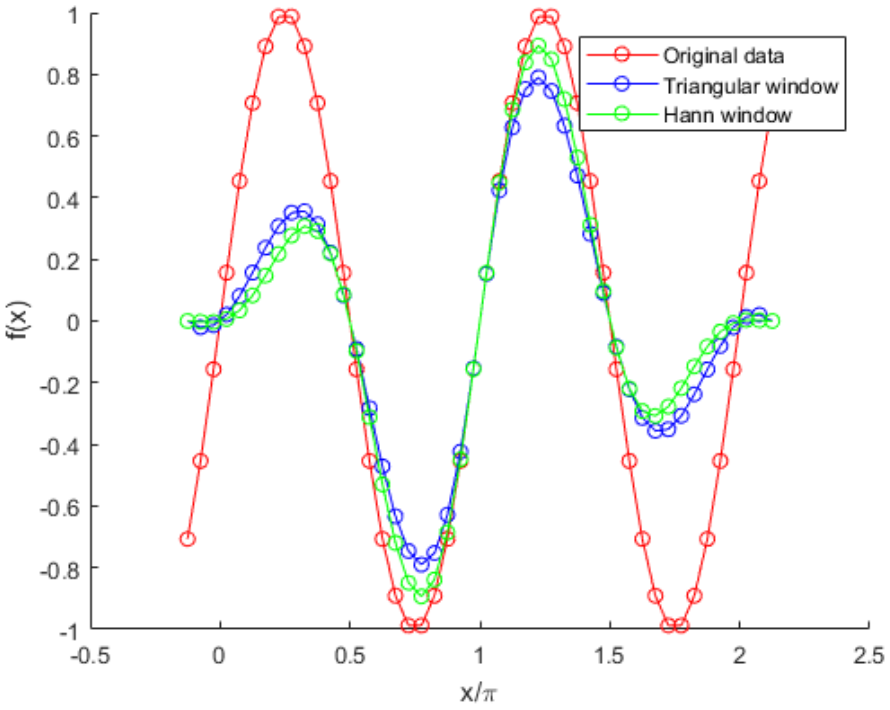
Note: There is no test function provided for `hannWindow` or `triangularWindow`. You will either need to write your own or come up with some other way of checking that your function is behaving as expected.

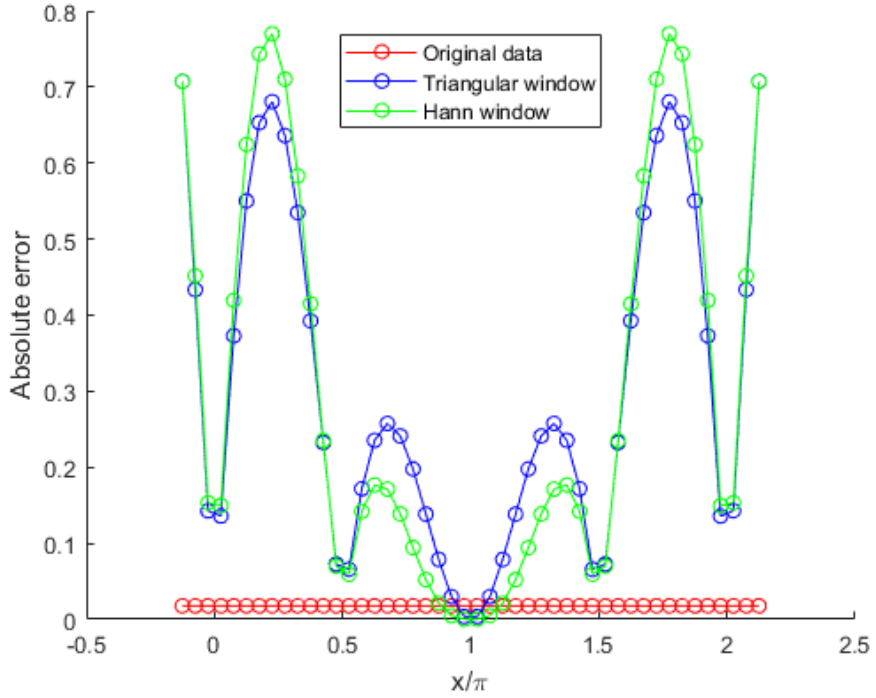
To compare the two windowing functions, write a third function `pulseWindowing` that compares the accuracy of your discrete Fourier transform function from Part A with and without windowing. The inputs/outputs should be (in this order):

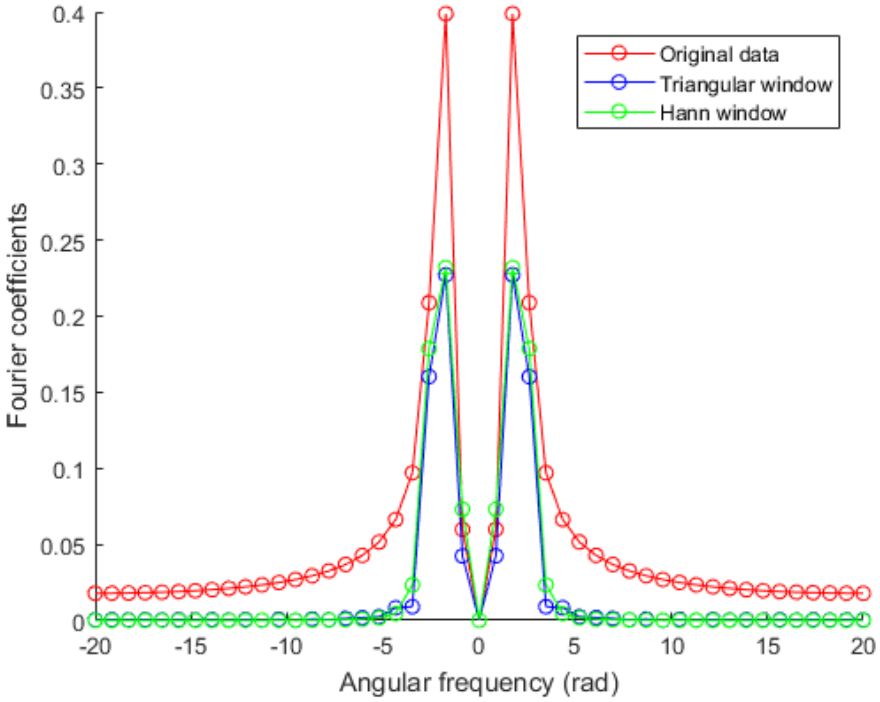
Inputs	
`x`	Sample points (independent variable)
`pulse`	Inline function

Outputs
No output variables Output three figures: original and windowed sample data; absolute error between original data and expansion with windowed data; Fourier coefficients with and without windowing

The inline function `pulse` defines the signal shape. The function should output three plots. The first plot should show the original sample data (`pulse` sampled at the sample points `x`), the Hann windowed data and the triangular windowed data. The second plot should show the absolute error between the known signal values and each of the Fourier expansions found using the original sample data, the Hann windowed data and the triangular windowed data. The third and final plot should show the *magnitude* of the Fourier coefficients found using the original sample data, the Hann windowed data and the triangular windowed data. When I sample a simple sine function $f(x) = \sin(2x)$ from $x = -\pi/8$ to $17\pi/8$ in increments of $\pi/20$ I get the following plots:


Original data
Triangular window
Hann window
f(x)
x/π
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-0.5
0
0.5
1
1.5
2
2.5


Original data
Triangular window
Hann window
Absolute error
x/π
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
-0.5
0
0.5
1
1.5
2
2.5



The key feature to note is the reduction in high frequency components for the windowed data. Remember that ideally, only the Fourier coefficients for angular frequencies of 2 and -2 rad would be non-zero (and equal to unity, *i.e.* two delta functions in the third plot above).

Task B3: Fourier filtering

Fourier filtering is a technique commonly employed in signal processing and involves three basic steps:

1. Fourier transform the signal from time space into frequency space
2. Apply a filter to the signal in frequency space
3. Inverse Fourier transform the filtered signal from frequency space back into time space

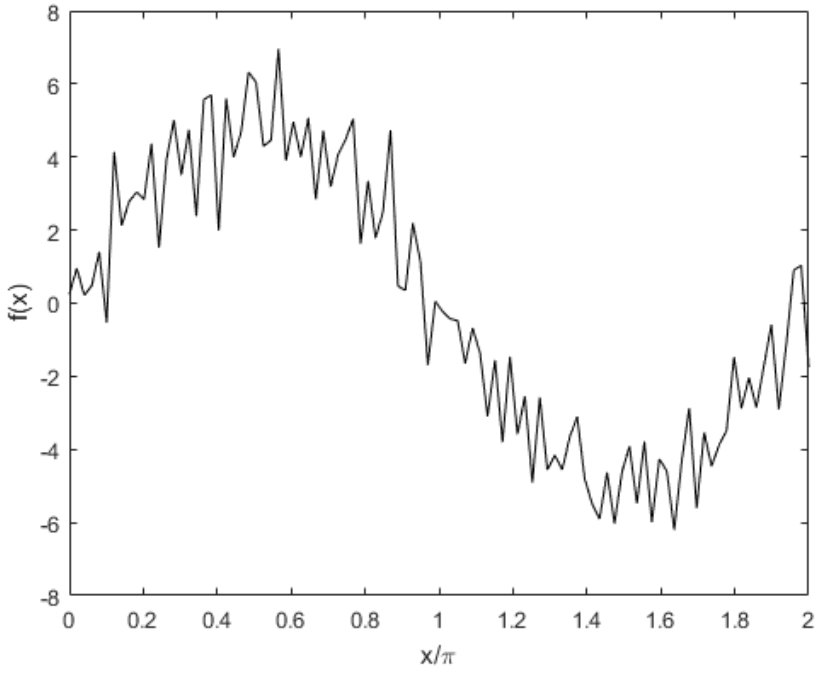
You will use your functions `discreteCFT` and `discreteICFT` to complete steps 1 and 3. To complete step 2 you will need to modify the coefficients c_r to alter the contributions of the different frequency components to the signal. Let's assume we once again have a simple sine wave signal $f(x) = \sin(x)$, but that we add on some noise. "White" noise is the most commonly used but has an even spectral density, *i.e.* it has equal power across all frequencies. This means it's very difficult to filter out, especially using Fourier techniques. Instead we will add what is called "blue noise", which increases in power with increasing frequency. To remove the noise from the signal you will create a low-pass filter.

Write a function `lowPassBlueNoise` that filters an input signal corrupted with blue noise. The inputs/outputs should be (in this order):

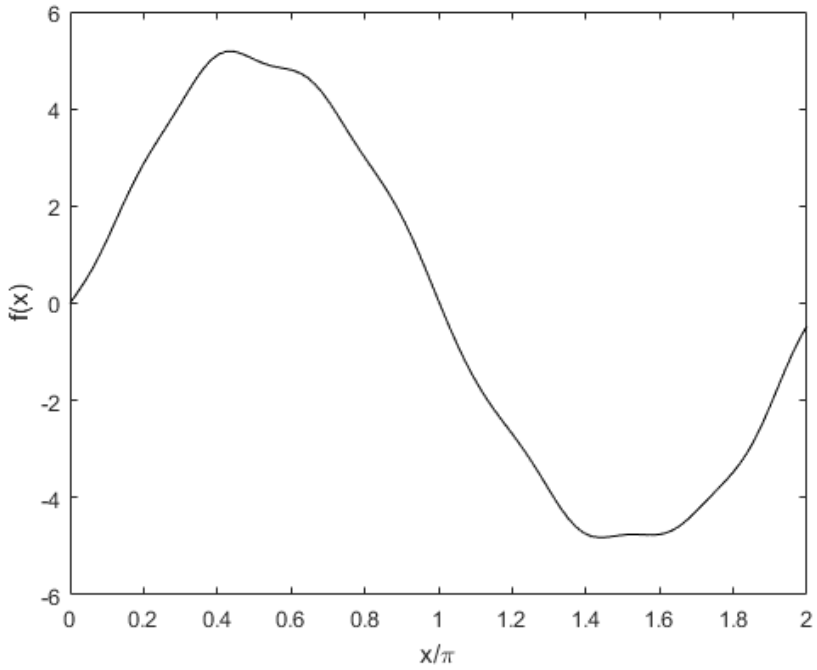
Inputs	
`x`	Sample points (independent variable)
`f_x`	Sample points (dependent variable)
`freq_c`	Cut-off frequency (*in Hz*)

Outputs	
`f_xFiltered`	Filtered sample points.

Use a rectangular window filter, *i.e.* one that sets the Fourier coefficients for frequencies above `freq_c` to zero and leaves those for frequencies below `freq_c` as is. You will find it helpful to write a plotter function to check your work. You can download the sample signal from onQ called **taskB3_sampleDataNoisy** to test your function. It's made up of a true signal $f(x) = 5\sin(x)$ and some blue noise. The data should look like this before filtering:



And something like this after filtering with a cut-off frequency of 1.5 Hz:



Note that since blue noise still has some spectral power below 1.5 Hz you should expect the signal to remain corrupted just much less so and at lower frequencies.

Part B: Submission List

Please submit the following files to the **Lab 8: Part A** dropbox on onQ for marking:

- discreteICFT.m
- hannWindow.m
- triangularWindow.m
- pulseWindowing.m
- lowPassBlueNoise.m
- Any other functions you wrote that the above files use