Using the Fast Fourier Transform Library (fftlib.tns)

Forest W. Arnold

June 2020

Typeset in LaTeX.

Trademarks

TI-Nspire is a registered trademark of Texas Instruments, Inc.

1 Purpose

The functions in the Fast Fourier Transform library,`fftlib.tns`, implement functions for transforming point (discrete) data using the Cooley-Tukey FFT algorithm and inverting (undoing) the transform, along with several utility functions for working with lists of point data.

The library can be installed and used with TI-Nspire CX CAS, TI-Nspire CAS Student Software, TI-Nspire CAS Teacher Software, and TI-Nspire CAS App for the iPad.

`fftlib.tns` is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or any later version. Visit `https://www.gnu.org/licenses/gpl.htm` to view the License.

2 Restrictions/Limitations

1. The library functions require the CAS version of TI-Nspire.

2. The size of the data that can be processed is limited by the memory and resources available to TI-Nspire. A list containing 2048 points was successfully transformed and inverted with `fft()` and `ifft()`. The desktop version of TI-Nspire CAS (Student Software) ran out of resources when attempting to transform 4096 points.

3. The lengths of the lists of points must be an integer power of 2: 2, 4, 8, 16, 32, ... If the number of points is not an integer power of 2, the lists can be zero-padded to satisfy this restriction.

3 Library History

Author: Forest W. Arnold
Date Created: 13 June 2020
Version: 1.0

4 Function Descriptions

The function `fft(v)` transforms a list of values using the Cooley-Tukey Fast Fourier Transform algorithm. The function `ifft(v)` untransforms a transformed list of values, restoring the values to their original values. The Cooley-Tukey FFT algorithm is described at the web link `https://en.wikipedia.org/wiki/Cooley-Tukey_FFT_algorithm`. The TI-Nspire implementation of the library functions is an in-place, iterative (non-recursive) version of the algorithm based on a C# implementation at the web link `https://rosettacode.org/wiki/Fast_Fourier_transform#C.23`.

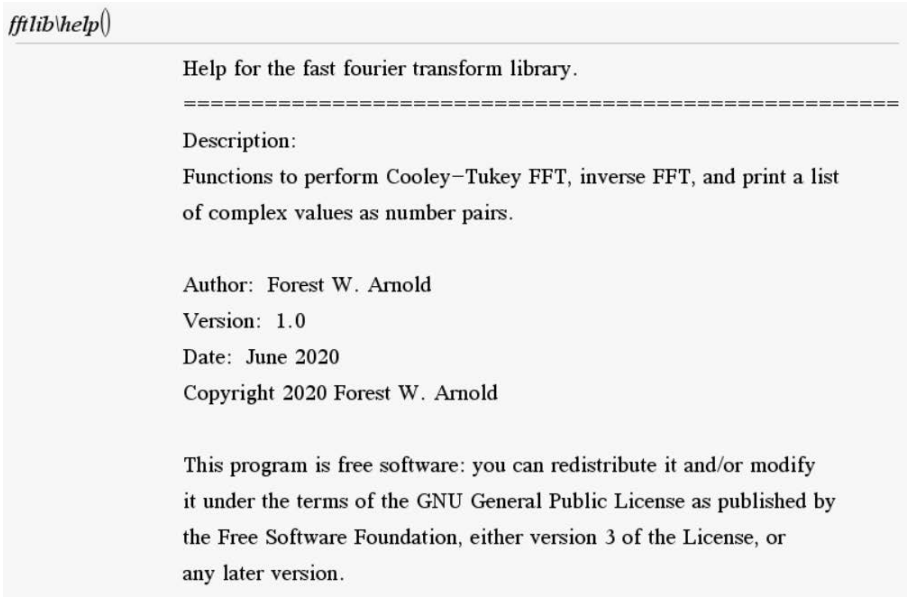
5 Function Usage

The following paragraphs describe each of the public functions in the library.

5.1 help()

Displays usage information for the functions in the library.

Select the help function in the library pane to add it to a calculator page, then execute the function:



Scroll to display information about a library function:

1. fft(v)

Purpose:

Transform a list of point values using Cooley−Tukey FFT algorithm.

Input Arguments:

v − The list of the points to transform.

Returns:

A list of the transformed values.

Restrictions:

a. The length of the list must be an integer power of 2:
   2, 4, 8, 16, 32, …
   If the number of points is not equal to a power of 2, create a longer zero−filled list whose length is a power of 2 and add the points as the leading entries in the list. For example, for 50 points, create a zero−filled list of length 64 and add the points as the first 50 entries.
b. The maximum number of points is limited by available memory. The code has been tested successfully with 2048 points on the desk−top version of TI−Nspire CAS.

5.2 fft(v)

Transform a list of points using the Cooley-Tukey Fast Fourier Transform algorithm.

Input arguments:

v - the list of point *y* values to transform.

Returns:

A list of transformed point *y* values.

Restrictions:

The length of the list must be an integer power of 2: 2, 4, 8, 16, ...

The length of the list is limited by the amount of memory and resources available.

5.2.1 Example

Create a list of 8 (2^3) y values and transform the values:

$y:=\{1,1,1,1,0,0,0,0\}$ $\{1,1,1,1,0,0,0,0\}$

$yt:=fftlib\backslash fft(y)$ $\{4.,1.-2.41421\cdot i,0.,1.-0.414214\cdot i,0.,1.+0.414214\cdot i,0.,1.+2.41421\cdot i\}$

5.3 ifft(v)

Perform inverse transform of a list of point *y* values using the Cooley-Tukey Fast Fourier Transform Algorithm.

Input arguments:

v - the list of transformed point *y* values.

Returns:

A list of the original, untransformed point *y* values.

Restrictions:

The length of the list must be an integer power of 2: 2, 4, 8, 16, ...

The length of the list is limited by the amount of memory and resources available.

5.3.1 Example

Invert `yt`, the transformed y values:

$iyt := fftlib\backslash ifft(yt)$ $\{1.,1.,1.,1.,0.,0.,0.,0.\}$

5.4 displayclist(v)

Display a list of the real and complex point *y* values as pairs.

Input arguments:

v - the list of point *y* values.

Returns: nothing

Notes:

The values are displayed in the format 'point: (real part, imaginery part)'.

5.4.1 Example

Display `yt`, the transformed y values as number pairs:

```
fft1ib\displayclist(yt)
```

```
1 : (4. , 0. )
2 : (1. , -2.41421 )
3 : (0. , 0. )
4 : (1. , -0.414214 )
5 : (0. , 0. )
6 : (1. , 0.414214 )
7 : (0. , 0. )
8 : (1. , 2.41421 )
```

5.5 padlist(v)

Pad a list of point *y* values with zeros so the length of the list is an integer power of 2.

Input arguments:

v - The list of values to pad.

Returns:

The zero-padded list suitable for input to the function `fft(v)`.

Notes:

This function is used to ensure the length of a list of *y* values meets the length requirement for the `fft(v)` function.

5.5.1 Example

Create a list of 6 values, pad the list, then transform the padded list:

```
y2:={1,1,1,0,0,0}
```

```
{1,1,1,0,0,0}
```

```
y2padded:=fft1ib\padlist(y2)
```

```
{1,1,1,0,0,0,0,0}
```

```
y2tpadded:=fft1ib\fft(y2padded)
{3.,1.70711-1.70711·i,-i,0.292893+0.292893·i,1.,0.292893-0.292893·i,i,1.70711+1.70711·
```

5.6 unpadlist(v,n)

Remove the zero-padding from a list of *y* values to restore the list to its original untransformed length after executing the function `ifft(v)`.

Input arguments:

v - the padded list of values

n - the original, unpadded length of the list.

Returns:

The list of values with the zero-padded entries removed.

Notes:

This function is executed after the function `ifft(v)` untransforms a zero-padded list.

5.6.1 Example

Invert the transformed padded list, then remove the padding to restore the original untransformed list:

```
iy2tpadded:=fft1lib\ifft(y2tpadded)                    {1.,1.,1.,0.,0.,0.,0.,0.}
y2original:=fft1lib\unpadlist(iy2tpadded,6)            {1.,1.,1.,0.,0.,0.}
```

6 An Example Application of the Fast Fourier Transform

The following example shows how to use the library functions to remove noise from a signal.

6.1 Example: Filtering a Noisy Signal

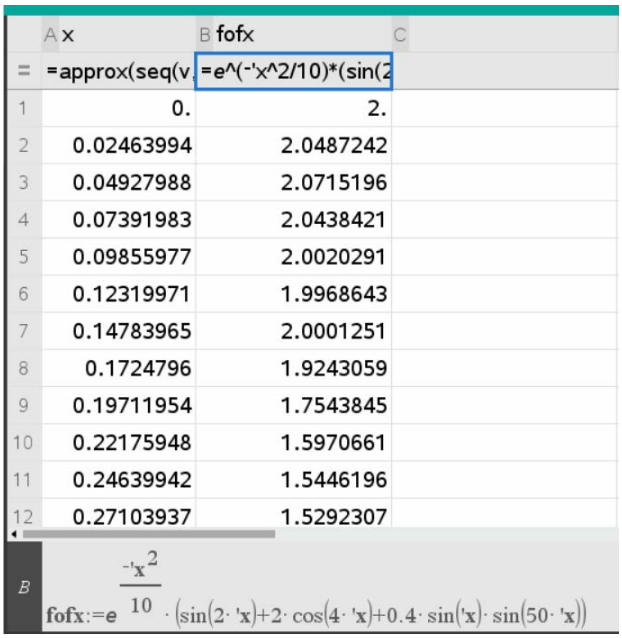
This example demonstrates how to filter a noisy signal using the Fast Fourier Transform. The noisy signal is modeled by the function

$$f(x) = \mathbf{e}^{-\frac{x^2}{10}} \left(\sin(2x) + 2\cos(4x) + 0.4\sin(x)\sin(50x)\right)$$

The signal sample consists of 256 data points collected over the interval $[0, 2\pi]$. The data for the signal is defined in a **Lists and Spreadsheet** page, a portion of which is shown in Figure 1. The x values for the interval are defined in the spreadsheet with `seq()` function:

$$x := approx\left(seq\left(v, v, 0, 2\pi, \frac{2\pi}{255}\right)\right)$$

The signal is plotted in a **Graphs** page as a scatter plot by selecting the **Document Tools - Graph Entry/Edit - Scatter Plot** menu item and entering the names of the lists containing the x and y values. The plotted points are connected with line segments by selecting a point and using the **Attributes** context-sensitive menu item. Figure 2



	A x	B fofx	C
=	=approx(seq(v	=e^(-'x^2/10)*(sin(2	
1	0.	2.	
2	0.02463994	2.0487242	
3	0.04927988	2.0715196	
4	0.07391983	2.0438421	
5	0.09855977	2.0020291	
6	0.12319971	1.9968643	
7	0.14783965	2.0001251	
8	0.1724796	1.9243059	
9	0.19711954	1.7543845	
10	0.22175948	1.5970661	
11	0.24639942	1.5446196	
12	0.27103937	1.5292307	

Figure 1: Spreadsheet of Noisy Signal Data

shows the graph of the signal.

Examining the graph of the signal, it is apparent that the high-frequency components (the noise in the signal) are located in the center of the graph, approximately between $x = 0.5$ and $x = 6.0$. The objective of the filtering process is to replace the high-frequency components with lower-frequency components, resulting in a collection of points that can be interpolated to produce a smooth curve which adequately models the underlying signal. The technique for achieving this consists of the following steps:

- Transform the sampled y values using the Fast Fourier Transform,
- Retain the first m and last m low-frequency values of the transformed data and set the remaining (middle) values to zero,
- Invert the modified transformed data using the inverse Fast Fourier Transform.

The result of these steps is a set of values that generate a smooth curve with low-frequency components when interpolated. The degree to which the high-frequency components are replaced with low-frequency components depends on the value of m. The process is performed three times with $m = 6, m = 12$, and $m = 24$ to illustrate how retaining the first and last m values affects the filtering.

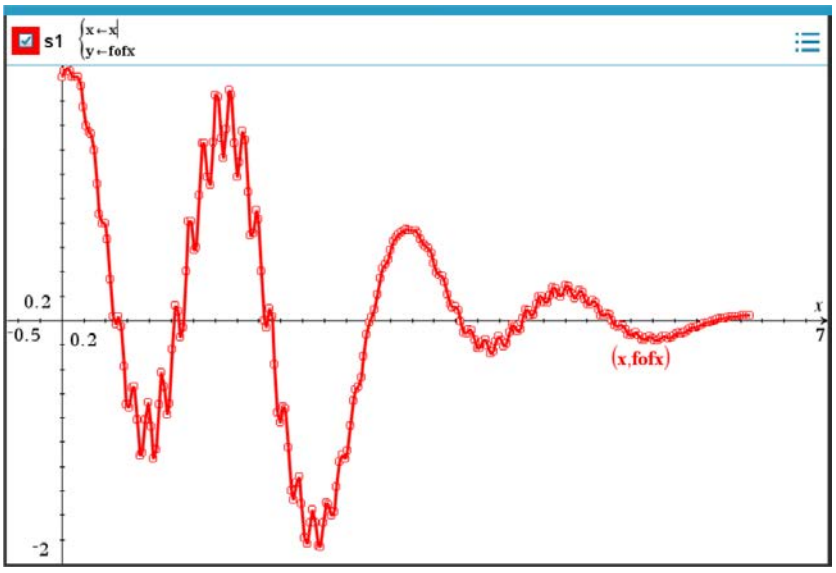


Figure 2: Graph of Noisy Signal

6.2 Filtering a Signal with TI-Nspire

6.2.1 Filtering with $m = 6$

Step 1: Transform the y values by executing `fft()`.

yt:=*fftlib**fft*($fofx$)
{22.146651,33.458709+1.1309843· i,6.6519085−42.147103· i,-12.767881+40.564183· i,106.7}

Step 2: Create a list consisting of the first 6 transformed y values, 244 zeros, and the last 6 transformed y values.

© create a list for m=6 using augment() and mid()

$ytm6$:=augment(augment(mid(yt,1,6),newList(256−12)),mid(yt,251,256))
{22.146651,33.458709+1.1309843· i,6.6519085−42.147103· i,-12.767881+40.564183· i,106.7}

Step 3: Invert new list of values by executing `ifft()`.

© invert the list ytm6 and extract the real part of the values (for plotting)

$invytm6$:=*fftlib**ifft*($ytm6$)
{1.1802988+0.09966934· i,1.2394957+0.09962776· i,1.289245+0.09742955· i,1.328624+0.09}

$reinvytm6$:=real($invytm6$)
{1.1802988,1.2394957,1.289245,1.328624,1.3568294,1.3731926,1.377194,1.3684745,1.34684}

The real part of the inverted values is extracted from the complex values for plotting. Figure 3 shows the graph of the filtered signal with the graph of the unfiltered signal.

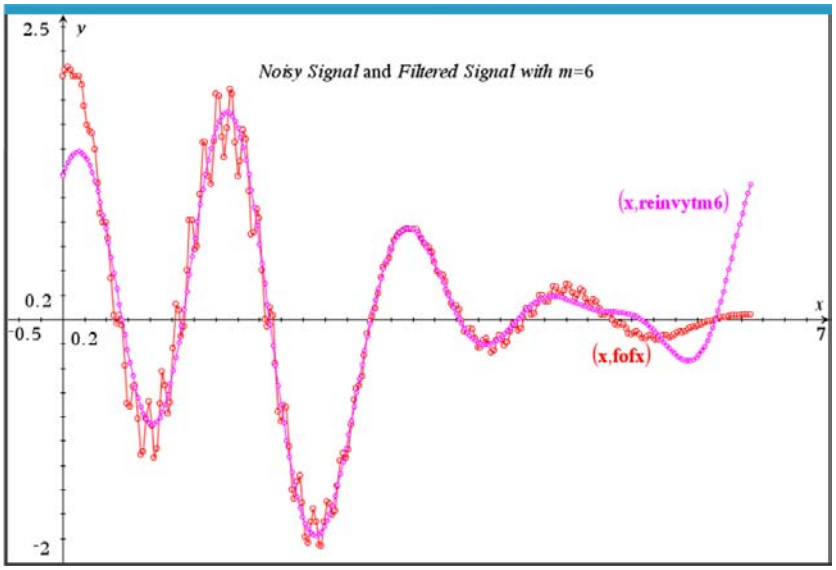


Figure 3: Graph of Noisy Signal and Filtered Signal with $m = 6$

6.2.2 Filtering with $m = 12$

Step 1: Transform the y values by executing `fft()`.

$yt := fft1lib\,fft(fofx)$

$\{22.146651, 33.458709+1.1309843\cdot i, 6.6519085-42.147103\cdot i, -12.767881+40.564183\cdot i, 106.7\}$

Step 2: Create a list consisting of the first 12 transformed y values, 232 zeros, and the last 12 transformed y values.

© create a list for m=12 using augment() and mid()

$ytm12 := augment(augment(mid(yt,1,12), newList(256-24)), mid(yt,245,256))$

$\{22.146651, 33.458709+1.1309843\cdot i, 6.6519085-42.147103\cdot i, -12.767881+40.564183\cdot i, 106.7\}$

Step 3: Invert new list of values by executing `ifft()`.

```
© invert the list ytm12 and extract the real part of the values (for plotting)
invytm12:=fft1lib\ifft(ytm12)
{1.1654037+0.02917639·i,1.3461308+0.02749282·i,1.5118642+0.02344159·i,1.6562204+0.
reinvytm12:=real(invytm12)
{1.1654037,1.3461308,1.5118642,1.6562204,1.773563,1.8592775,1.9099901,1.9237164,1.899
```

The real part of the inverted values is extracted from the complex values for plotting. Figure 4 shows the graph of the filtered signal with the graph of the unfiltered signal.

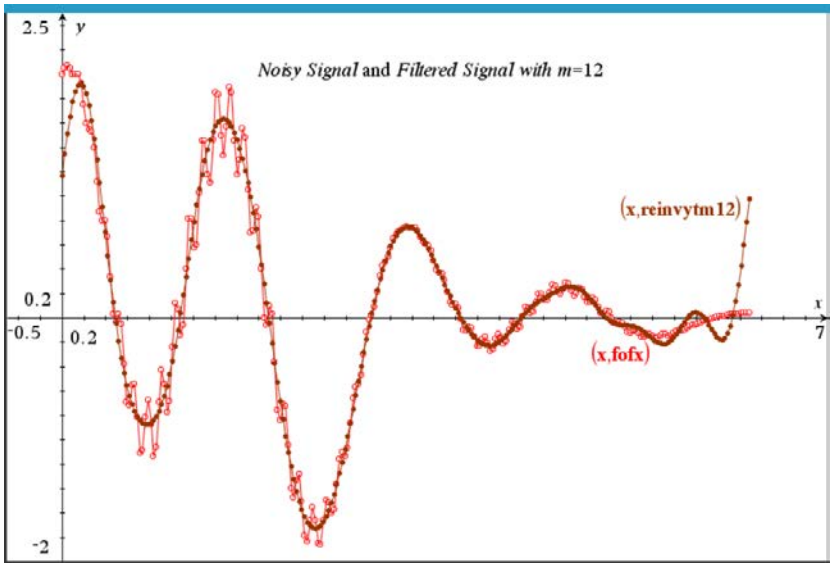


Figure 4: Graph of Noisy Signal and Filtered Signal with $m = 12$

6.2.3 Filtering with $m = 24$

Step 1: Transform the y values by executing `fft()`.

```
yt:=fft1lib\fft(fofx)
{22.146651,33.458709+1.1309843·i,6.6519085−42.147103·i,-12.767881+40.564183·i,106.7
```

Step 2: Create a list consisting of the first 24 transformed y values, 208 zeros, and the last 24 transformed y values.

```
© create a list for m=24 using augment() and mid()
ytm24:=augment(augment(mid(yt,1,24),newList(256−48)),mid(yt,233,256))
{22.146651,33.458709+1.1309843·i,6.6519085−42.147103·i,-12.767881+40.564183·i,106.7
```

Step 3: Invert new list of values by executing `ifft()`.

```
© invert the list ytm24 and extract the real part of the values (for plotting)
invytm24:=fftlib\ifft(ytm24)
{1.2292876+0.01297184· i,1.5887344+0.00898121· i,1.8850383+0.00196337· i,2.0880083−0.
reinvytm24:=real(invytm24)
{1.2292876,1.5887344,1.8850383,2.0880083,2.1840311,2.1774779,2.088044,1.9448751,1.779
```

The real part of the inverted values is extracted from the complex values for plotting. Figure 5 shows the graph of the filtered signal with the graph of the unfiltered signal.

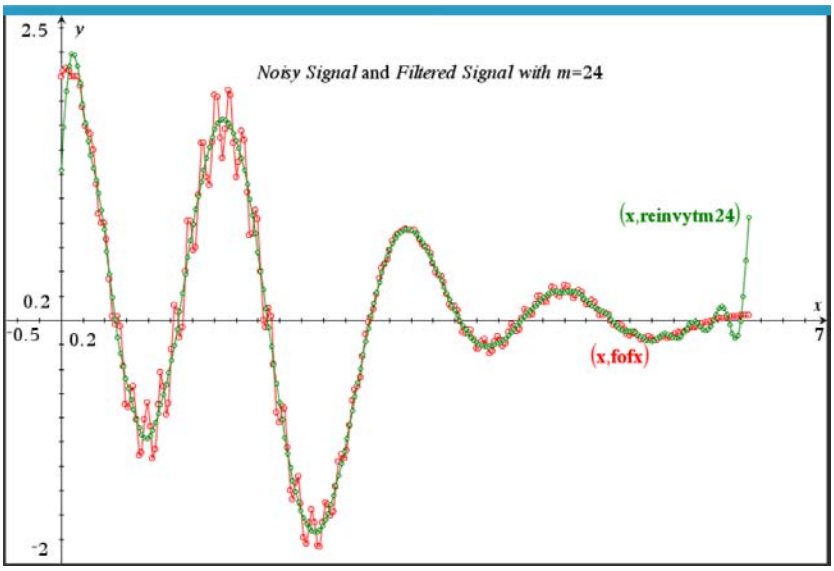


Figure 5: Graph of Noisy Signal and Filtered Signal with $m = 24$

6.2.4 Comparing the Results

Figure 6 is a graph of the noisy signal along with the three filtered signals. As the graph shows, the three filtered signals model the central portion (from about $x = 0.32$ to about $x = 5.5$) of the original signal very well. However, all three deviate from the original signal at the beginning and ending points of the sample, with the deviation being greatest at the ending points. The filtered signal with $m = 24$ results in the best approximation to the original signal.

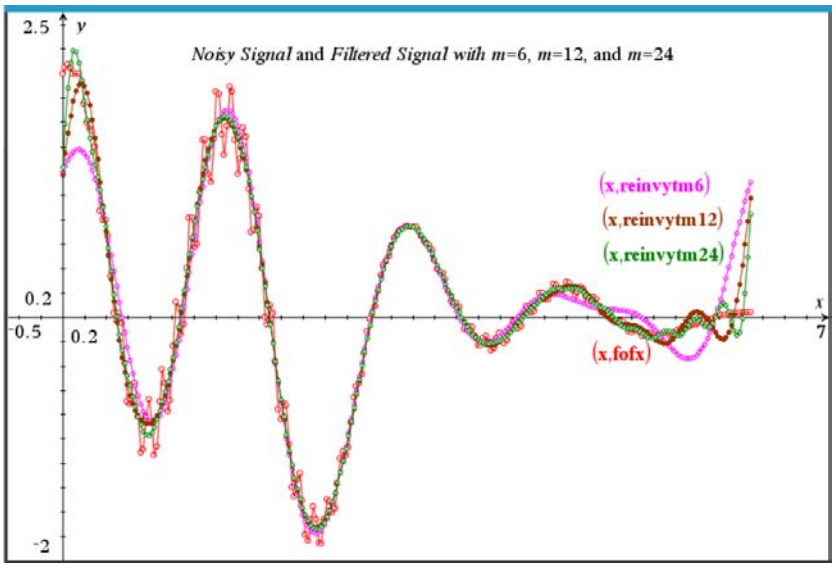


Figure 6: Comparison Graph of Noisy Signal and Filtered Signals

6.3 Some Other Applications of the Fast Fourier Transform

A few areas where the Fast Fourier Transform is used in mathematics and engineering are

- Solutions of partial differential equations.
- Data compression and decompression.
- Determining dominant frequencies of vibrating signals by converting signals from a time spectrum to a frequency spectrum.
- Image processing.
- Solving difference equations.
- Fast large integer and polynomial multiplication.

An online textbook that thoroughly discusses both theory of and applications of the Discrete Fourier Transform and Fast Fourier Transform is available at the link

`https://ccrma.stanford.edu/~jos/mdft/mdft.html`

This textbook also has examples (with graphs) of various applications of Fourier transforms.