2 – Frequency Domain Simulation

OBJECTIVES

The purpose of this exercise is to demonstrate the relationship between the time and frequency domains and to show how the latter can be easily used to simulate wave propagation of sound pulses.

SUMMARY

Consider a transmitting transducer that injects a short pulse of sound (a toneburst) into a medium which is then received by a second transducer. In this exercise you will produce animations that show how the received time-domain signal changes as the distance between the transducers increases. One animation will be for the case of non-dispersive propagation (e.g. bulk waves in water) and the second will be for dispersive propagation (e.g. A_0 Lamb waves in a plate).

EXERCISE 2A – NON-DISPERSIVE PROPAGATION

Consider a wavepulse containing 5 cycles at 1 MHz that propagates 150 mm at 3000 ms^{-1}.

1. On paper calculate: (a) the length of time required for the signal to propagate this distance; (b) the duration of the pulse; (c) and the length of time window required to completely enclose the pulse at both the start position and 150 mm later; (d) the time-step required to provide 10 points per cycle in the time-domain; (e) the wavelength at the centre frequency; (f) the size of distance step required to produce 4 points per cycle.

Load the starter script `STARTER_frequency`.

2. Use the data from 1 to enter into the program as inputs.
3. Enter the necessary commands to set up the vectors `time` and `distance` and enter a suitable formula for the variable `time_at_centre_of_pulse` to calculate whereabouts in time the centre of the pulse should be at zero distance such that the start of the pulse is at zero time.
4. Enter a suitable formula to produce the vector `sine_wave` which should have a zero at the centre of the pulse.
5. Call the NDE function `fn_hanning` to produce the vector `window` which is a Hanning window with the appropriate characteristics. Use help to find out what arguments are needed for the function (see Exercise 1).

Run the program until the comment line that says `At this point the input time-domain signal should have been created` (i.e. by typing a line that says `return` (Matlab) or `sys.exit()` (Python) just before it, which will cause the program to end at that point) and confirm by plotting a graph of `input_signal` against `time` that the input signal is as desired (it should have the correct frequency, number of cycles and start at time zero).

6. Work out the number of points, `fft_pts`, to use in subsequent Fourier transforms which should be the next even number greater or equal to the number of points in the vector `time`. Use the appropriate fast Fourier transform function (see Fast Fourier Transforms in Python and Matlab section) with arguments `input_signal` and `fft_pts` to create a vector `spec` that is the frequency spectrum of `input_signal` and which contains a number of points equal to `fft_pts`. Truncate the variable `spectrum` to half its length. (This is why `fft_pts` is forced to be an even number. Only dealing with the first half of the spectrum makes the subsequent program simpler to write as the second half of the spectrum corresponds to negative frequency components but does not contain extra information.)
7. The frequency step between adjacent points in spectrum is given by `freq_step = 1 / (time_step * fft_pts)`. This relationship (i.e. frequency step is reciprocal of duration of time signal that is transformed) is always true when using fast Fourier transform algorithms. Using `freq_step`, create a vector, `frequency`, that contains the frequencies corresponding to the frequency components in `spectrum`, starting at zero.

Run the program until the line that says `At this point the frequency spectrum of the input time signal should have been created` and confirm by plotting a graph of `spectrum` against `frequency` that the frequency spectrum has the correct centre frequency of 1 MHz.

At present skip over the `if…else…` statements which create a new variable, `velocity`. However, note that, because the input variable `nondispersive_propagation` is set to `True` (Python) or one (Matlab), these statements set `velocity` equal to the input variable, `velocity_at_centre_frequency`.

8. Create a vector of wavenumbers from `frequency` and `velocity`.
9. Create a matrix of zeros, `p`, of the appropriate size to hold a time-domain signal of the same length as `input_signal` in each column and with enough columns for each propagation distance that will be considered.

10. The next loop is the main calculation loop where the time-domain signals are simulated for different propagation distances by applying phase delays to the input signal spectrum. Complete the missing statements in the loop. You will need to create a vector, `delayed_spectrum`, that contains the phase-delayed input signal spectrum, subject it to an inverse Fourier transform using the appropriate function ensuring the `fft_points` are used for the transform, truncate the resulting signal to be the same length as the original time-domain input signal and finally slot the result into the appropriate column of `p`.
11. Confirm that the program runs without errors and that the matrix `p` contains finite numerical values (not all zeros or `NaNs`). Finally add a loop at the end to produce an animated display of the contents of `p`, column by column. You may wish to use the `ylim` command to lock the scale of the y-axis.

EXERCISE 2B – DISPERSIVE PROPAGATION

12. Set the input variable `nondispersive_propagation` to `False` (Python) or zero (Matlab) at the start of the program. The dispersive propagation that will be simulated is approximately that experienced by low frequency flexural waves in a plate (the A_0 Lamb wave mode) and results in a phase velocity that is roughly proportional to the square root of frequency.
13. Add the necessary code in the `if…else…` series of statements to create a frequency dependent velocity vector. This should be proportional to the square root of frequency and equal to the value of `velocity_at_centre_frequency` (i.e. 3000 ms^{-1}) when `frequency` is equal to `centre_frequency` (i.e. 1 MHz). There will be a problem with having zero velocity at zero frequency when the wave number is calculated (a zero divided by zero error) so to avoid this, add a line that forces the first element in `velocity` to be equal to any non-zero value, say 1.
14. Execute the program and observe what happens to the pulse as it propagates and its final shape compared to the case of non-dispersive propagation. Estimate its group velocity (i.e. how fast the envelope has propagated).

FAST FOURIER TRANSFORMS MATLAB AND PYTHON

The fast Fourier transforms have almost identical implementations in Matlab and Python.

Matlab – the relevant functions are `fft(x, n)` and `ifft(x,n)`, which perform, respectively, `n`-point forward and inverse fast Fourier transforms of variable `x`. If `x` is a matrix, then the transforms are performed on each column unless an optional third argument is used to specify the dimension to do the transform in. If `n` is omitted, then the transform will use however many points there are in the relevant dimension of `x`, but in general it is better to specify `n` explicitly to avoid unpredictable behaviour. If `n` is greater than the relevant dimension, the input is padded with zeros before being transformed; if it is less then the input is truncated.

Python – the relevant Fourier transforms are in the `fft` module of the `numpy` library. If `numpy` is imported as `np` then the functions are `np.fft.fft(x, n)` and `np.fft.ifft(x,n)`, which perform, respectively, `n`-point forward and inverse fast Fourier transforms of variable `x`. If `x` is an array with 2 or more dimension, then the transforms are performed on the last dimension unless an optional third argument is used to specify the dimension to do the transform in. If `n` is omitted, then the transform will use however many points there are in the relevant dimension of `x`, but in general it is better to specify `n` explicitly to avoid unpredictable behaviour. If `n` is greater than the relevant dimension, the input is padded with zeros before being transformed; if it is less then the input is truncated.