1 – Wave Propagation

OBJECTIVES

The purpose of this exercise is to: (a) illustrate the phenomena of continuous wave propagation; (b) demonstrate different programming and plotting techniques for use in later exercises.

SUMMARY

An equation of the form $p(x,t) = \cos(\pm kx \pm \omega t)$ describes the propagation of a continuous sinusoidal wave propagating in the positive or negative x direction. This exercise will produce an animation showing how such a wave propagates and what effect the various parameters have on the wave. It will also help to familiarise you with the programming environment.

EXERCISE 1A – WAVE PROPAGATION

Consider a wave with a frequency of 1 MHz and a velocity of 3000 ms^{-1}.

Load the starter script `STARTER_wave` (see Matlab and Python section below).

1. Modify the program to produce a 100-frame animation of the wave with a time step of 0.05 μs between frames. Each frame should be a snapshot of the wave, `p`, as a function of distance, `x`. The distance axis should show 8 complete cycles (i.e. it should be 8 wavelengths long) and the distance between adjacent points should be less than 1/10 of a wavelength so that it looks smooth. You will need to enter the numeric values given here as input parameters in the programme and suitable equations to calculate angular frequency, wavelength, wavenumber etc. from the input parameters. See the coding tip below on how to create evenly-spaced arrays of values. You will also need to type an appropriate equation for the wave, `p`, within the loop.
2. Find out whether the wave propagates in the positive or negative x direction in the following cases: (a) positive wavenumber and using $p = \cos(kx + \omega t)$; (b) positive wavenumber and using $p = \cos(kx - \omega t)$; (c) negative wavenumber and using $p = \cos(-kx - \omega t)$; (d) negative wavenumber and using $p = \cos(-kx + \omega t)$.
3. Explain why the convention $\cos(kx - \omega t)$ is preferred to $\cos(kx + \omega t)$. From now on the former will be used.
4. Investigate the effect of: (a) doubling the velocity; (b) doubling the frequency.
5. Replace the equation in the loop with an appropriate one using complex exponential notation and verify that the same results are obtained if the real part is plotted.
6. Modify the program so that the real and imaginary parts of the wave are plotted at the same time in blue and green respectively.

EXERCISE 1B – BASIC PROGRAMMING SKILLS

Resave your program with a different name so you can compare results with the original version if necessary.

7. Edit the loop so that the results are not plotted directly to screen as they are calculated but instead stored in the columns of a matrix, `p`.
8. Write a second loop after the first to produce the same animation as before, but this time by plotting the data in the columns of `p`.
9. Plot the contents of `p` on the screen using a suitable 2D or 3D visualisation functions (e.g. in Matlab `imagesc`, `surf`; in Python `imshow`, `plot_surface`) function in a separate figure. Label the axes.

CODING TIP: CREATING ARRAYS OF EVENLY-SPACED VALUES

It is often necessary to create an array of evenly-spaced values, `x`, based on 3 of the following inputs: start value `xs`; end value `xe`; step value `dx`; number of points `n`. The best way to do this depends on what the inputs are, and which ones must be satisfied exactly, as indicated by the examples in the table below.

Inputs	Matlab	Python (import `numpy` as `np`)	Satisfied exactly
`n`, `xs`, `xe`	`x=linspace(xs,xe,n)`	`x=np.linspace(xs,xe,n)`	`n`, `xs`, `xe`
`n`, `xs`, `dx`*	`x=[0:n-1]*dx+xs`	`x=np.arange(n)*dx+xs`	`n`, `xs`, `dx`
`n`, `xe`, `dx`*	`x=[1-n:0]*dx+xe`	`x= np.arange(1-n,1)*dx+xe`	`n`, `xe`, `dx`
`xs`, `dx`, `xe`**	`x=[xs:dx:xe]`	`x=np.arange(xs,xe,step=dx)`	`xs`, `dx`

*In this case, the trick in both languages is to first make an array of integers from 0 to one less than the desired number of points inclusive (`[0:n-1]` in Matlab and `np.arange(n)` in Python). Multiplying this by `dx` gives a vector with exactly the right number of points and step size that starts at zero so adding on `xs` gives the desired result.

**Note that the three requirements in this case cannot be exactly satisfied unless the difference between first and last values is an integer number of steps. The Matlab syntax will produce a vector of points such that the actual last value is less than or equal to the desired last value, and the Python syntax will produce a vector of points such that the actual last value is less than the desired last value. Avoid this syntax unless the number of points and exact last value are not critical.

MATLAB AND PYTHON

These exercises can all be performed in either Matlab or Python. Equivalent resources are provided for both and are accessed as described below:

Matlab **–** from Blackboard, download the starter and example scripts (`STARTER_*.m` and `EXAMPLE_*.m`), the NDE function files (`fn_*.m`), the Matlab data files (`*.mat`), and the encrypted Matlab function files (`*.p`) to whichever folder you want to work in. There is one Matlab function per function file, so provided the function file is in the folder where you are working (or in a folder on the Matlab path), you will be able to use it. Type `help function_name` to get help for any function.

Python **–** from Blackboard, download the starter and example scripts (`STARTER_*.py` and `EXAMPLE_*.py`), the NDE functions module (`NDE_functions.py`), the complied binary dynamically linked libraries (`*.dll`), and the data files (`*.json`). In Python, all the special functions required for the unit are in `NDE_functions.py` so this just needs to be imported at the top of any script where they will be used, e.g. by having the line `import NDE_functions as nde` at the top of the script as in the examples. If you also type `import NDE_functions as nde` in the console window then you can subsequently type `help(nde.function_name)` in the console window to get help about any function in the module.