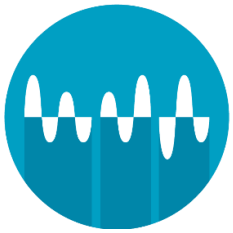


Biomedical Ultrasound Group

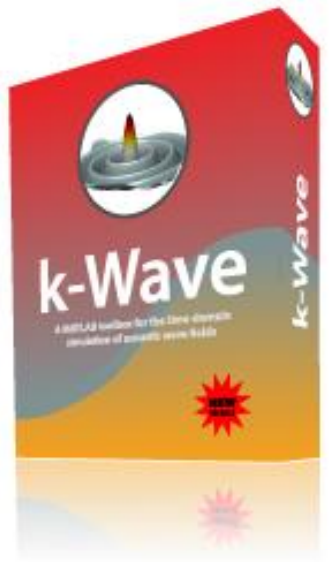
k-Wave short course – Part 3
Introduction to k-Wave

Bradley Treeby and Ben Cox

Biomedical Ultrasound Group (BUG)
Department of Medical Physics and Biomedical Engineering
University College London

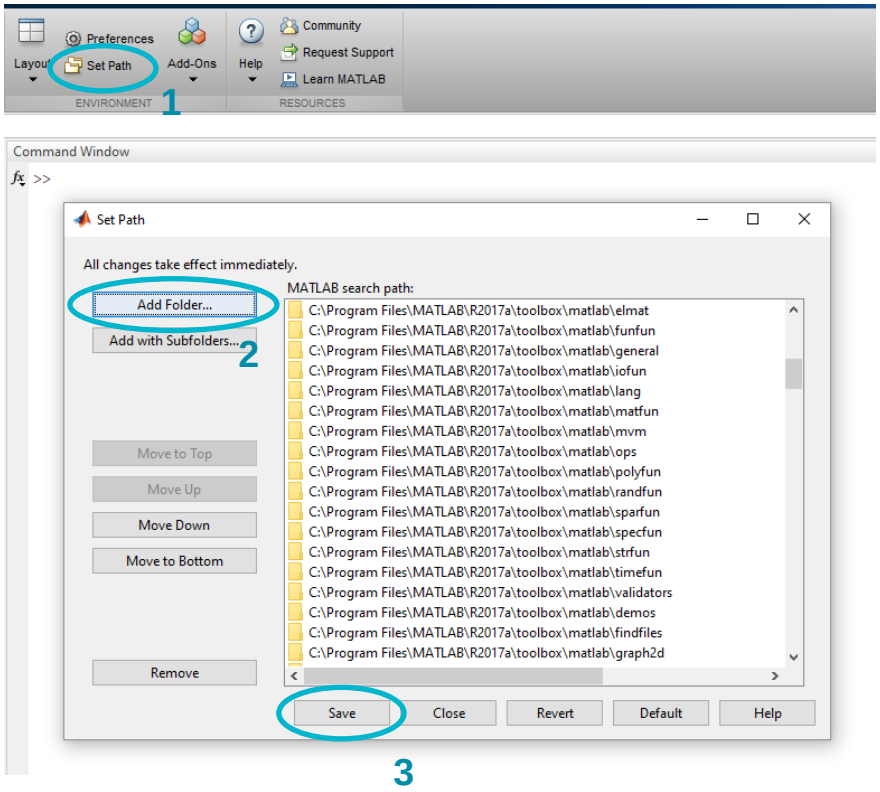
k-Wave toolbox

- Open-source acoustics toolbox written in MATLAB / C++
- Time domain modelling in heterogeneous media
- Two design goals:
  - Models are computationally efficient
  - Models are very easy to use
- Brief history:
  - 2009: First release for photoacoustics
  - 2010: Linear ultrasound simulation
  - 2011: Nonlinear ultrasound simulation
  - 2012: Native C++ code
  - 2013: MPI code optimised for clusters
  - 2014: Elastic simulation
  - 2015: Native GPU code
  - 2016: Multi-GPU code
  - 2017: Thermal code



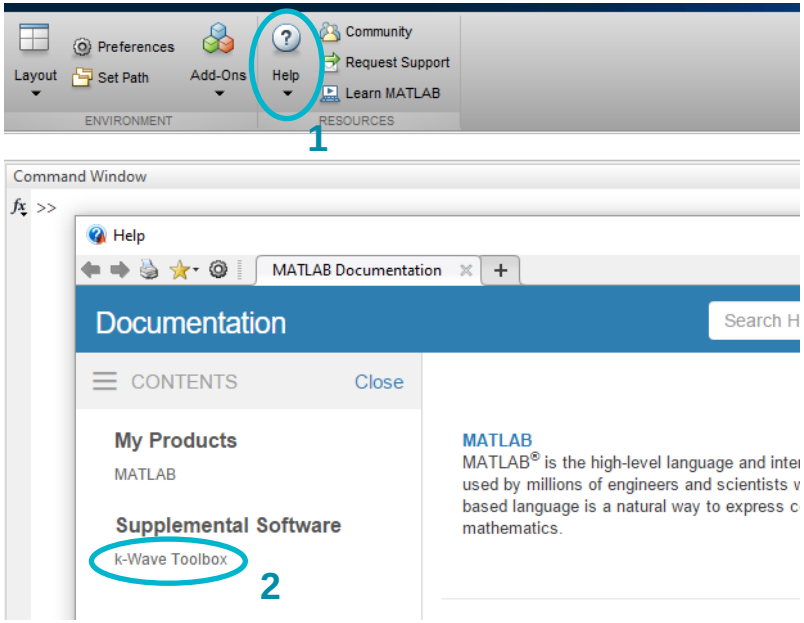
Getting started with k-Wave

- Download toolbox from www.k-wave.org/download.php
- Unzip, and move the k-Wave folder somewhere useful
- Add the root k-Wave folder to the MATLAB path



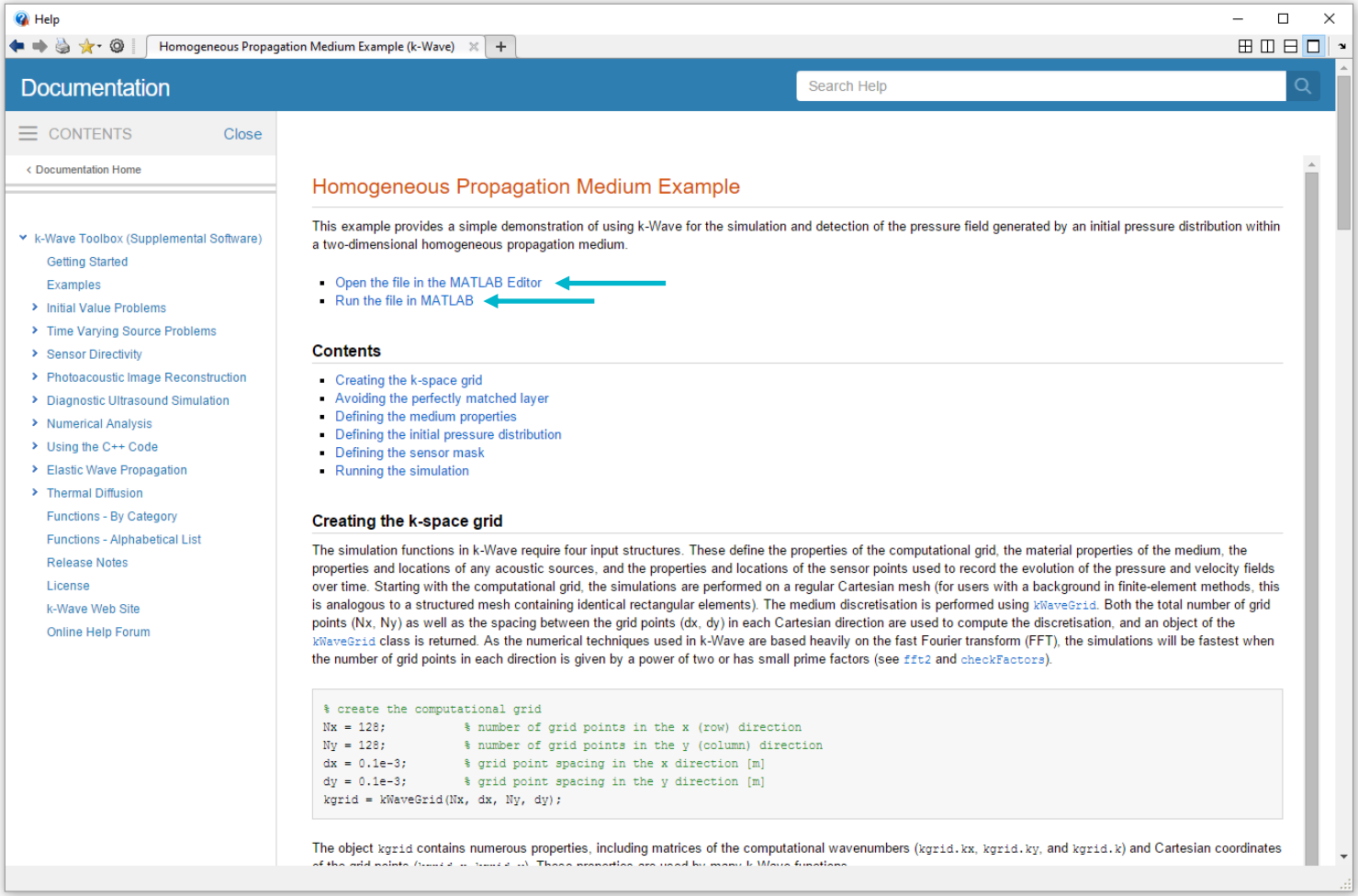
Getting started with k-Wave

- Download toolbox from www.k-wave.org/download.php
- Unzip, and move the k-Wave folder somewhere useful
- Add the root k-Wave folder to the MATLAB path
- Open the help browser and select k-Wave



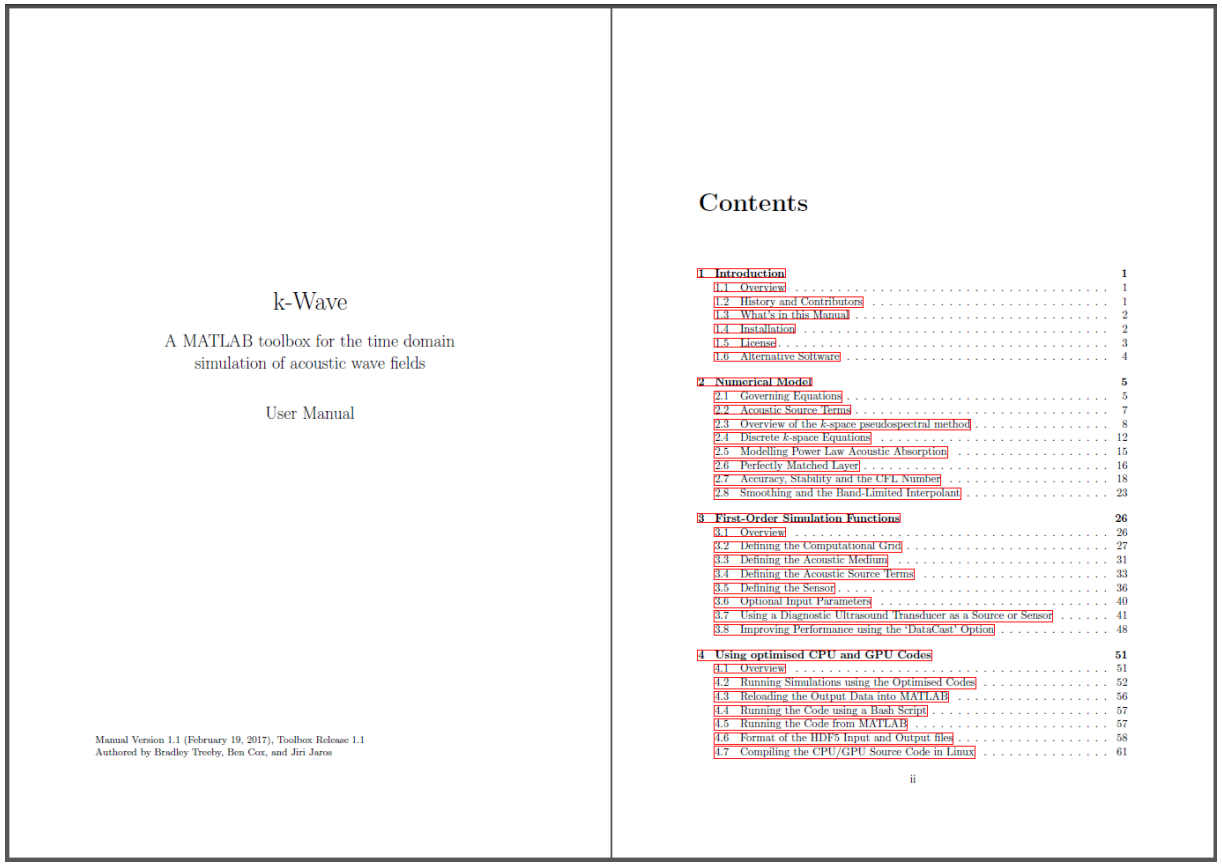
MATLAB help

- Many worked examples included as part of the toolbox



User manual

- Extensive manual covering formulation and usage
- Available from www.k-wave.org/documentation.php



k-Wave

A MATLAB toolbox for the time domain simulation of acoustic wave fields

User Manual

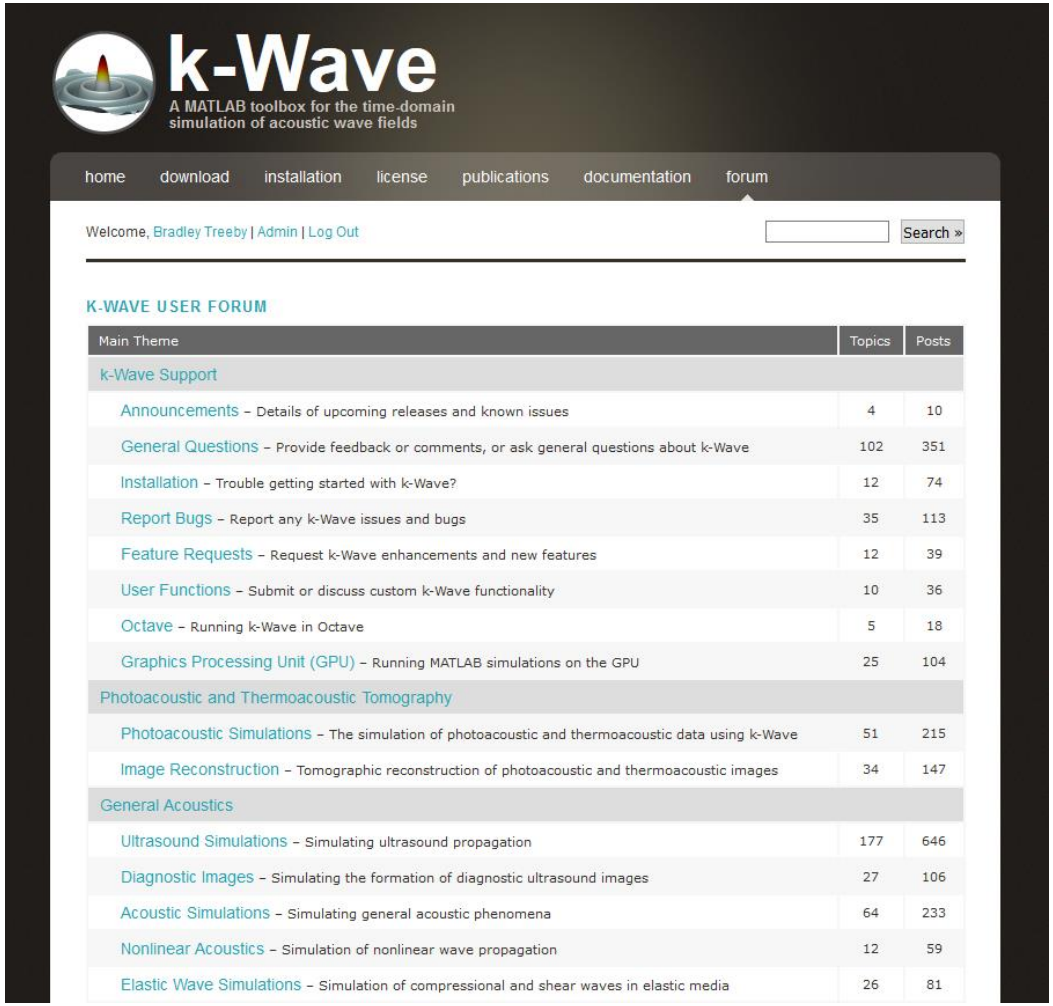
Manual Version 1.1 (February 19, 2017), Toolbox Release 1.1
Authored by Bradley Treeby, Ben Cox, and Jiri Jaros

Contents

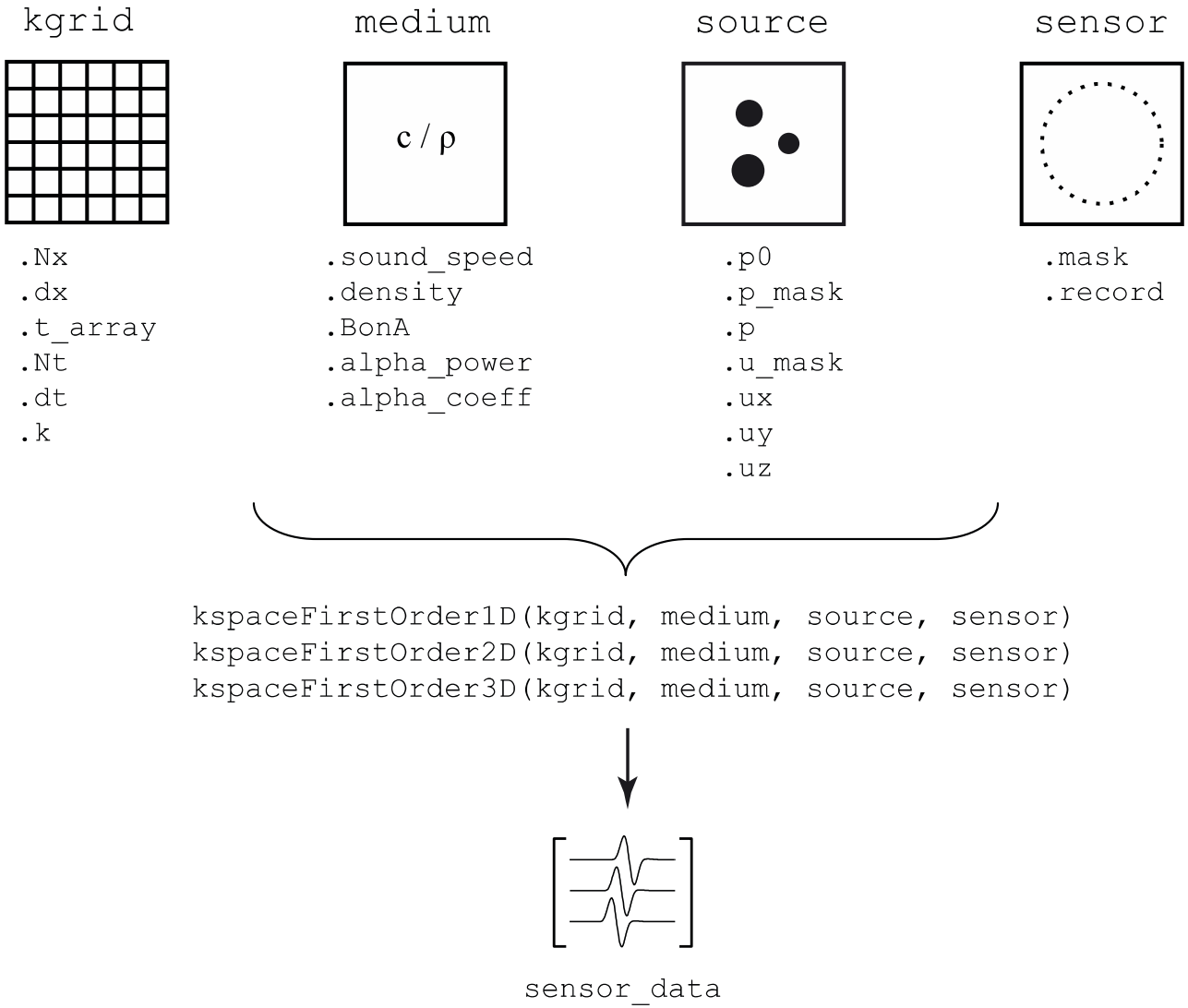
ii

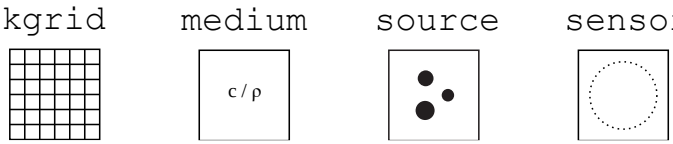
User forum (www.k-wave.org/forum)

- Active online user forum (> 2400 posts)

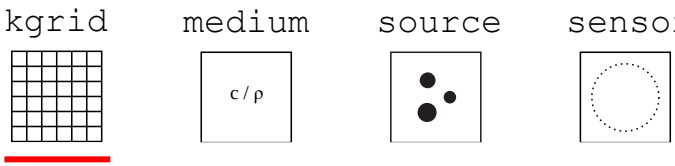


Code architecture

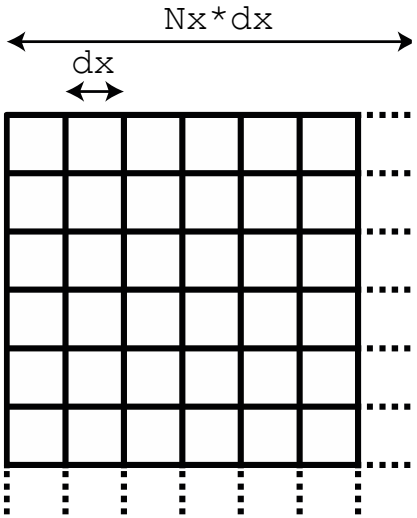




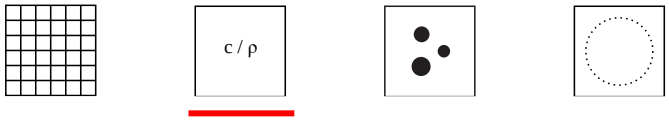

```
% create the computational grid
kgrid = kWaveGrid(Nx, dx, Ny, dy);

% define the time array
kgrid.setTime(Nt, dt);

% define the compressional sound speed [m/s]
medium.sound_speed = 1500*ones(Nx, Ny);
medium.sound_speed(Nx/2:end, :) = 2000;

% define the mass density [kg/m^3]
medium.density = 1000;

% define the absorption coefficient [dB/(MHz^y cm)]
medium.alpha_coeff = 0.75;
medium.alpha_power = 1.5;

% define the source
source.u_mask = makeCircle(Nx, Ny, x_pos, y_pos, radius);
source.ux = toneBurst(sampling, freq, cycles);

% define a circular binary sensor mask
sensor.mask = makeCircle(Nx, Ny, Nx/2, Ny/2, sen_rad);

% run the simulation
sensor_data = kspaceFirstOrder2D(kgrid, medium, source, sensor);
```



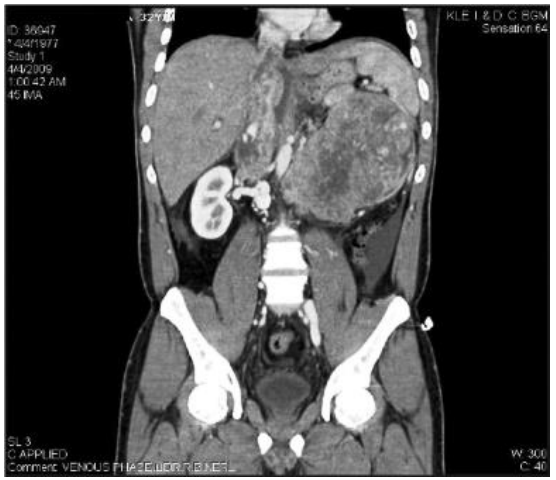

```
% create the computational grid
kgrid = kWaveGrid(Nx, dx, Ny, dy);

% define the time array
kgrid.setTime(Nt, dt);

% define the compressional sound speed [m/s]
medium.sound_speed = 1500*ones(Nx, Ny);
medium.sound_speed(Nx/2:end, :) = 2000;

% define the mass density [kg/m^3]
medium.density = 1000;

% define the absorption coefficient [dB/(MHz^y cm)]
medium.alpha_coeff = 0.75;
medium.alpha_power = 1.5;

% define the source
source.u_mask = makeCircle(Nx, Ny, x_pos, y_pos, radius);
source.ux = toneBurst(sampling, freq, cycles);

% define a circular binary sensor mask
sensor.mask = makeCircle(Nx, Ny, Nx/2, Ny/2, sen_rad);

% run the simulation
sensor_data = kspaceFirstOrder2D(kgrid, medium, source, sensor);
```

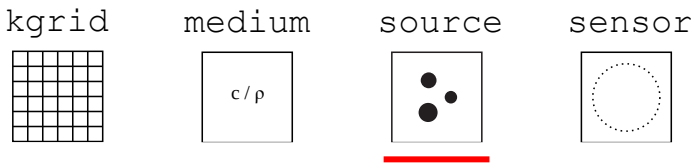


kgrid medium source sensor

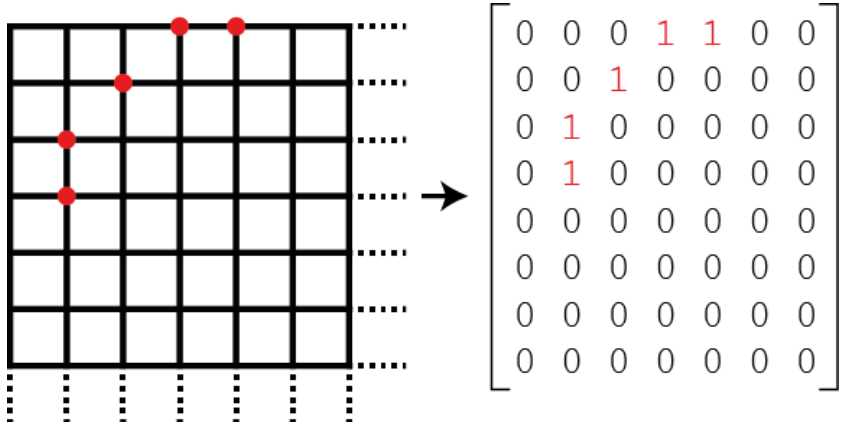



```
% create the computational grid
kgrid = kWaveGrid(Nx, dx, Ny, dy);

% define the time array
kgrid.setTime(Nt, dt);

% define the compressional sound speed [m/s]
medium.sound_speed = 1500*ones(Nx, Ny);
medium.sound_speed(Nx/2:end, :) = 2000;

% define the mass density [kg/m^3]
medium.density = 1000;

% define the absorption coefficient [dB/(MHz^y cm)]
medium.alpha_coeff = 0.75;
medium.alpha_power = 1.5;

% define the source
source.u_mask = makeCircle(Nx, Ny, x_pos, y_pos, radius);
source.ux = toneBurst(sampling, freq, cycles);

% define a circular binary sensor mask
sensor.mask = makeCircle(Nx, Ny, Nx/2, Ny/2, sen_rad);

% run the simulation
sensor_data = kspaceFirstOrder2D(kgrid, medium, source, sensor);
```



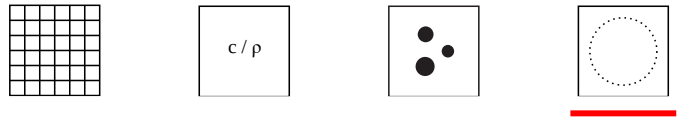



```
% create the computational grid
kgrid = kWaveGrid(Nx, dx, Ny, dy);

% define the time array
kgrid.setTime(Nt, dt);

% define the compressional sound speed [m/
medium.sound_speed = 1500*ones(Nx, Ny);
medium.sound_speed(Nx/2:end, :) = 2000;

% define the mass density [kg/m^3]
medium.density = 1000;

% define the absorption coefficient [dB/(M
medium.alpha_coeff = 0.75;
medium.alpha_power = 1.5;
```

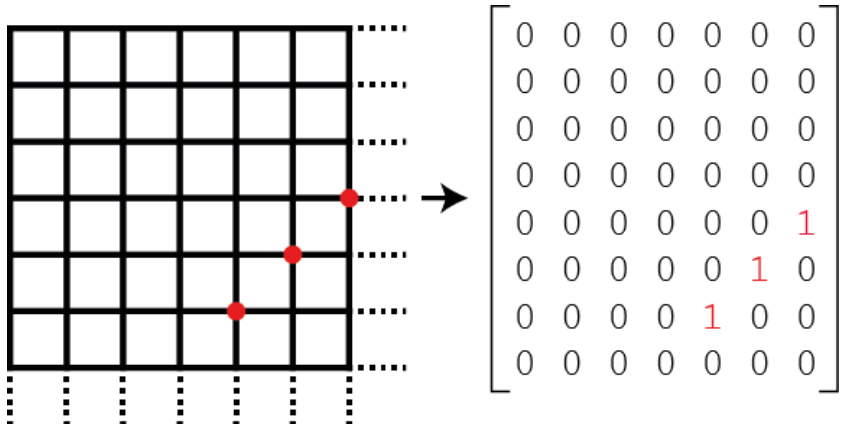



```
% define the source
source.u_mask = makeCircle(Nx, Ny, x_pos, y_pos, radius);
source.ux = toneBurst(sampling, freq, cycles);

% define a circular binary sensor mask
sensor.mask = makeCircle(Nx, Ny, Nx/2, Ny/2, sen_rad);

% run the simulation
sensor_data = kspaceFirstOrder2D(kgrid, medium, source, sensor);
```

kgrid medium source sensor




```
% create the computational grid
kgrid = kWaveGrid(Nx, dx, Ny, dy);

% define the time array
kgrid.setTime(Nt, dt);

% define the compressional sound speed [m/
medium.sound_speed = 1500*ones(Nx, Ny);
medium.sound_speed(Nx/2:end, :) = 2000;

% define the mass density [kg/m^3]
medium.density = 1000;

% define the absorption coefficient [dB/(M
medium.alpha_coeff = 0.75;
medium.alpha_power = 1.5;

% define the source
source.u_mask = makeCircle(Nx, Ny, x_pos, y_pos, radius);
source.ux = toneBurst(sampling, freq, cycles);

% define a circular binary sensor mask
sensor.mask = makeCircle(Nx, Ny, Nx/2, Ny/2, sen_rad);

% run the simulation
sensor_data = kspaceFirstOrder2D(kgrid, medium, source, sensor);
```

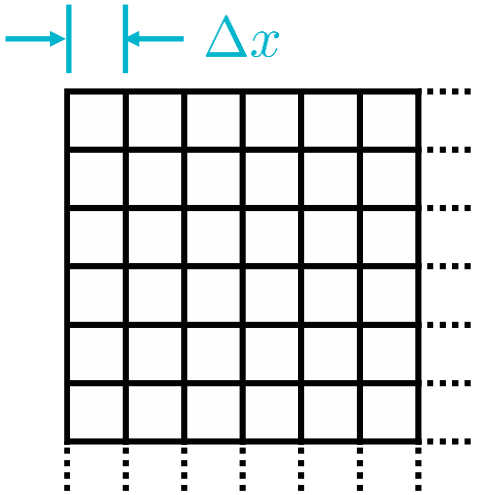


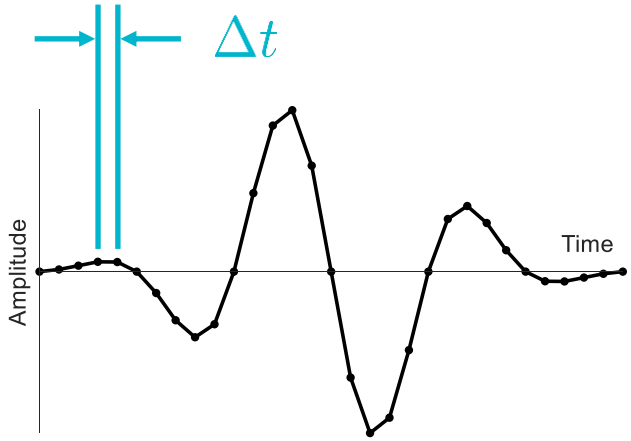
kgrid medium source sensor


```
% create the computational grid
kgrid = kWaveGrid(Nx, dx, Ny, dy);

% define the time array
kgrid.setTime(Nt, dt);

% define the compressional sound speed [m/s]
medium.sound_speed = 1500*ones(Nx, Ny);
medium.sound_speed(Nx/2:end, :) = 2000;

% define the mass density [kg/m^3]
medium.density = 1000;

% define the absorption coefficient [dB/(MHz^y cm)] and exponent
medium.alpha_coeff = 0.75;
medium.alpha_power = 1.5;

% define the source
source.u_mask = makeCircle(Nx, Ny, x_pos, y_pos, radius);
source.ux = toneBurst(sampling, freq, cycles);

% define a circular binary sensor mask
sensor.mask = makeCircle(Nx, Ny, Nx/2, Ny/2, sen_rad);

% run the simulation
sensor_data = kspaceFirstOrder2D(kgrid, medium, source, sensor);
```

Spatial and temporal frequencies

- The spatial and temporal grids typically define different Nyquist frequencies





$$k_{x,\max} = \frac{\pi}{\Delta x}$$

$$f_{\max,x} = \frac{\min(c_0)}{2\Delta x}$$

$$\omega_{\max} = \frac{\pi}{\Delta t}$$

$$f_{\max,t} = \frac{1}{2\Delta t}$$

Spatial and temporal frequencies

- The maximum temporal frequency that can be represented on the grid (called the maximum supported frequency) is given by

$$f_{\max,x} = \frac{\min(c_0)}{2\Delta x}$$

- Frequencies higher than $f_{\max,x}$ won't be propagated

- This is typically smaller than the maximum temporal frequency that can be represented in the source

$$f_{\max,t} = \frac{1}{2\Delta t}$$

Defining the grid

- All simulations performed on a regular Cartesian grid
- Grid is specified by the number of grid points in each dimension, and the grid spacing
- Rule of thumb:
  - Start with at least 3 points per wavelength (PPW) *at the highest frequency of interest*, and then calculate `dx`
  - Calculate `Nx` based on the physical domain size

```
% compute dx and Nx based on x_size and f_max
ppw = 3;
dx  = c0_min / (ppw * f_max);
Nx  = round(x_size / dx);
```

Defining the grid

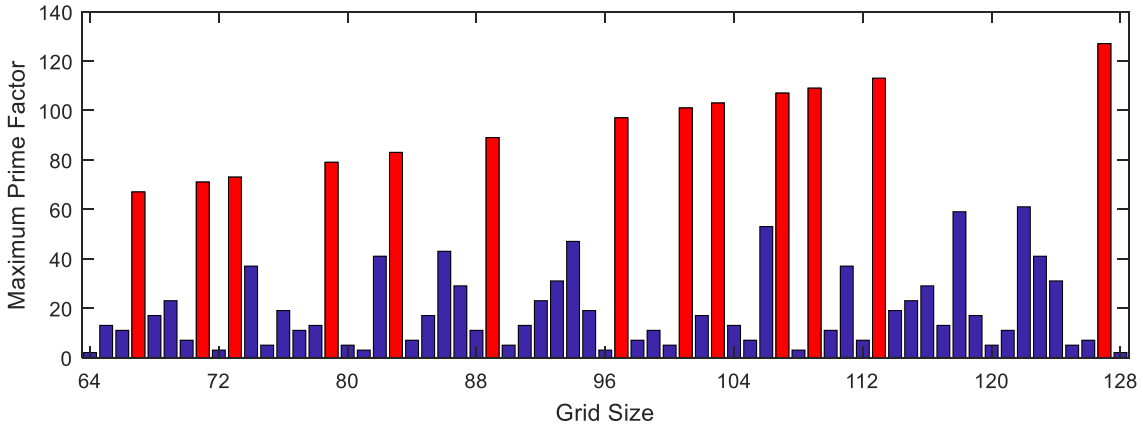
- Grid is specified by creating an object of the `kWaveGrid` class
- Can be given any name, typically we use `kgrid`

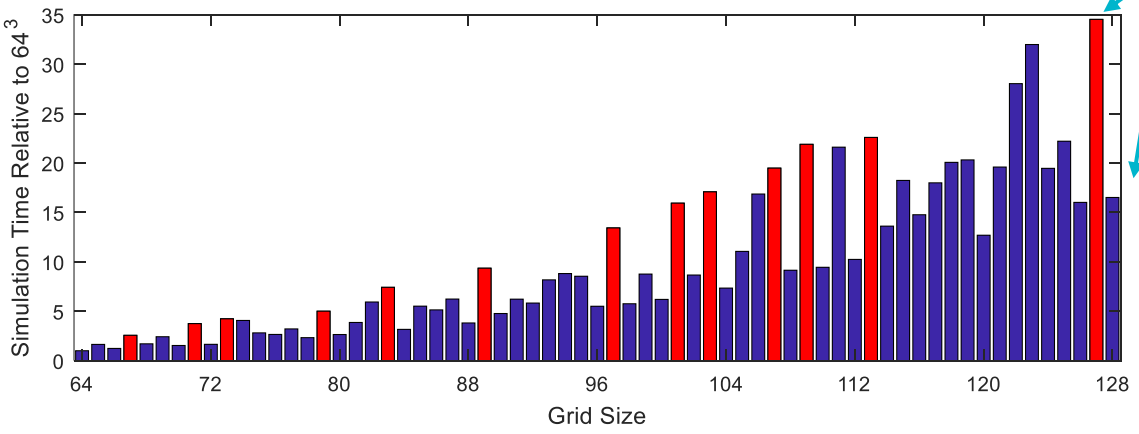
```
% create k-Wave grid object
kgrid = kWaveGrid(Nx, dx, Ny, dy);
```

- The grid object contains:
  - the wavenumber grids expressed as vectors or plaid matrices, e.g., `kx_vec` and `kx`
  - the spatial grid coordinates expressed as vectors or plaid matrices, e.g., `x_vec` and `x`

Defining the grid

- Simulations based heavily on FFT, so will be fastest when grid sizes (including PML) have small prime factors





factor of two performance penalty

Defining the grid

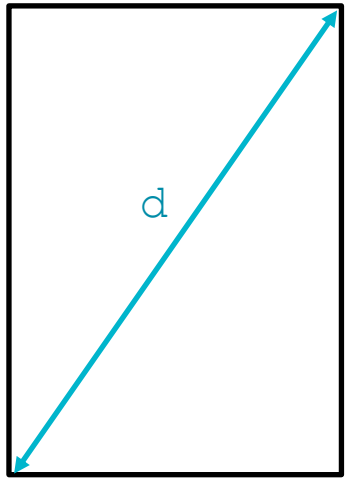
- Simulations based heavily on FFT, so will be fastest when grid sizes (including PML) have small prime factors
- Can use the function `checkFactors`

```
>> checkFactors(200, 300)
Numbers with a maximum prime factor of 2
256
Numbers with a maximum prime factor of 3
216  243  288
Numbers with a maximum prime factor of 5
200  225  240  250  270  300
Numbers with a maximum prime factor of 7
210  224  245  252  280  294
Numbers to avoid (prime numbers)
211  223  227  229  233  239  241  251  257  263  269
271  277  281  283  293
```

Defining the time step

- The size and number of time steps are controlled by `Nt` and `dt`, which are set to `'auto'` by default
- This sets

```
dt = CFL * dx_min / c0_max

Nt = d / (c0_min * dt)
```



- where the default CFL is 0.3

$$\mathrm{CFL} = \frac{c_0 \Delta t}{\Delta x}$$

- Note, this definition of the CFL is used in all dimensions

Defining the time step

- There are two methods to specify `Nt` and `dt`, `setTime` and `makeTime` (the latter is used if set to `'auto'`)

```
% specify Nt and dt explicity
kgrid.setTime(Nt, dt);
```

```
% calculate Nt and dt (this is the same as 'auto')
kgrid.makeTime(medium.sound_speed);

% calculate Nt and dt specifying the CFL
kgrid.makeTime(medium.sound_speed, CFL);

% calculate Nt and dt specifying the CFL and end time
kgrid.makeTime(medium.sound_speed, CFL, t_end);

% calculate Nt and dt specifying the end time
kgrid.makeTime(medium.sound_speed, [], t_end);
```

Defining the medium properties

- There are 5 medium properties that can be specified

Fieldname	Description
`medium.sound_speed`	sound speed distribution within the medium [m/s]
`medium.density`	ambient density distribution within the medium [kg/m^3]
`medium.BonA`	nonlinearity parameter
`medium.alpha_coeff`	power law absorption prefactor [dB/(MHz^{y}cm)]
`medium.alpha_power`	power law absorption exponent

- These can be specified as scalar values or matrices the same size as the computational grid (except `alpha_power`, which must be scalar)
- If `BonA` is not specified, linear equations are solved
- If `alpha_coeff` or `alpha_power` are not specified, lossless equations are solved

Defining a source

There are three types of source available in k-Wave

1. **Initial pressure source –** `source.p0`
   - Initial value problem assuming the particle velocity is zero (i.e., photoacoustics)

2. **Time varying pressure source –** `source.p`
   - Time rate of the input of mass per unit volume
   - Generates a monopole field

3. **Time varying velocity source –** `source.ux`
   - Input of body forces per unit mass
   - Generates a dipole field

Defining a source mask

- Need to define a source mask, and then either a single time series for all points in the source, or a separate time series for each point in the source mask
- Source mask is ordered column-wise (opposite to C++ which is row-wise), e.g.,

0	1	0
1	0	1
1	0	1
0	1	0

mask

0	3	0
1	0	5
2	0	6
0	4	0

index order

- In this case, the source would have 6 rows containing the time series for each source point

Defining a sensor mask

- The sensor structure defines what is recorded from the simulation (e.g., pressure, velocity), and where it is recorded
- The sensor mask can be defined in three ways:

1. As a binary matrix which specifies the grid points that record the data, where the 1's represent the grid points that form part of the sensor
2. As the grid coordinates of two opposing corners of a line (in 1D), rectangle (in 2D), or cuboid (in 3D) of grid points that record the data
3. As a set of Cartesian coordinates defined as an *N* x *M* matrix, where *N* is the number of dimensions, and *M* the number of sensor points

Defining the sensor record parameters

- By default, the acoustic pressure field is recorded and passed directly to the output `sensor_data`
- Other parameters can be recorded by specifying `sensor.record = {'p', 'u', 'p_max', ...}`

'p' (acoustic pressure)
'p_max' (maximum pressure)
'p_min' (minimum pressure)
'p_rms' (RMS pressure)
'p_final' (final pressure field)
'p_max_all' (maximum pressure at all grid points)
'p_min_all' (minimum pressure at all grid points)
'u' (particle velocity)
'u_max' (maximum particle velocity)
'u_min' (minimum particle velocity)
'u_rms' (RMS particle velocity)
'u_final' (final particle velocity field)
'u_max_all' (maximum velocity at all grid points)
'u_min_all' (minimum velocity at all grid points)
'u_non_staggered' (particle velocity on non-staggered grid points)
'I' (time varying acoustic intensity)
'I_avg' (average acoustic intensity)

Output data

- The output is indexed differently depending on the type of sensor mask used (Cartesian, binary, cuboid-corners)
- For a binary sensor mask, the output data is returned in column-wise order, e.g.,

0	1	0
1	0	1
1	0	1
0	1	0

mask

0	3	0
1	0	5
2	0	6
0	4	0

index order

- The data is indexed as:
  `sensor_data(sensor_index, time_index)`
- Note, the sensor points do not modify the wave field in any way (they act as transparent observers), and the response is omnidirectional

Optional input parameters

- There are many additional optional input parameters to control the behavior of k-Wave
- These are specified as `'Param', value` pairs after the main inputs

```
% save the output as a movie in mp4 format
kspaceFirstOrder2D(kgrid, medium, source, sensor, ...
      'RecordMovie', true, 'MovieProfile', 'MPEG-4');
```

```
% turn off the display, and run in single precision
kspaceFirstOrder2D(kgrid, medium, source, sensor, ...
      'PlotSim', false, 'DataCast', 'single');
```

Other k-Wave functions

- There are many additional functions included in the k-Wave toolbox, including…
  - Time Domain Wave Propagation in Fluid Media (1D, 2D, 3D, AS)
  - Time Domain Wave Propagation in Elastic Media (2D, 3D)
  - Time Domain Heat Diffusion (1D, 2D, 3D)
  - Reference Solutions
  - Geometry and Shape Creation
  - Acoustic Absorption Coefficient Calculation and Conversion
  - Grid and Matrix Utilities
  - Filtering and Spectral Utilities
  - Display and Visualisation
  - Signal Creation and Processing
  - HDF5 Utilities
  - System Parameters and Utilities
  - Photoacoustic Image Reconstruction