Sonar Workbench v3.2

User's Guide

Thomas J Deal, MSc, PhD

Naval Undersea Warfare Center, Newport, RI

Contents

List of Tables

List of Figures

Abstract

Sonar Workbench is a suite of Matlab tools for the design and analysis of sonar systems. Version 1.0 focused on array construction and beam pattern construction and analysis. Version 2.0 added TEAMS interface support. Version 3.0 vectorized position and orientation parameters for increased efficiency. Version 3.1 added full support for calculating beam patterns in the body frame for rotated arrays. Version 3.2 removed Signal Processing Toolbox dependencies and added 2D beam pattern analysis tools. Sonar Workbench includes multiple element types, from which the user constructs an array with arbitrary element position and orientation. The user defines complex weights for each element to create a beam with the desired shape, which can be evaluated at any elevation and azimuthal angles, plotted in 3D space, and measured for beam width and directivity index.

Chapter 1

Introduction

Sonar Workbench is a suite of Matlab tools for the design and analysis of sonar systems. Version 1.0 focused on array construction and beam pattern construction and analysis. Version 2.0 added TEAMS interface support. Version 3.0 vectorized position and orientation parameters for increased efficiency. Version 3.1 added full support for calculating beam patterns in the body frame for rotated arrays. Version 3.2 removed Signal Processing Toolbox dependencies and added 2D beam pattern analysis tools. Sonar Workbench includes multiple element types, from which the user constructs an array with arbitrary element position and orientation. The user defines complex weights for each element to create a beam with the desired shape, which can be evaluated at any elevation and azimuthal angles, plotted in 3D space, and measured for beam width and directivity index.

This user's guide is intended to introduce a new user to Sonar Workbench, the array theory it implements, and its usage in the Matlab environment. Much of the content is adapted from *An Introduction to Sonar Systems Engineering* [1], which is recommended as a companion resource for the user interested in understanding more of the theory. Chapter 2 introduces the coordinate system and reference frames used in Sonar Workbench. Chapter 3 explains how to define elements and lists the built-in element types. Chapter 4 describes the process to build arrays consisting of uniform or mixed element types. Chapter 5 defines beams and demonstrates a method for calculating amplitude and phase weights for conventional beamforming. Chapter 6 demonstrates how Sonar Workbench implements these concepts and conventions and shows the user how to use it for their own analysis.

The example element, array, and beam definitions used in this guide can

be found in the `test` folder, along with a script `CreateSampleBeam.m` the user can execute to demonstrate the features of Sonar Workbench.

Chapter 2

Coordinate System and Reference Frames

Sonar Workbench uses a right-handed, Cartesian coordinate system known as North-East-Down (NED), as shown in Fig. 2.1. In this coordinate system, the first coordinate, x, points north, the second coordinate, y, points east, and the third coordinate, z, points down. Roll, γ, is rotation about the x axis, pitch, θ, is rotation about the y axis, and yaw, ψ, is rotation about the z axis. The NED coordinate system is ideal for underwater applications, because depth is measured downward from the surface, yaw is measured clockwise from north, and pitch is measured relative to the horizontal plane.

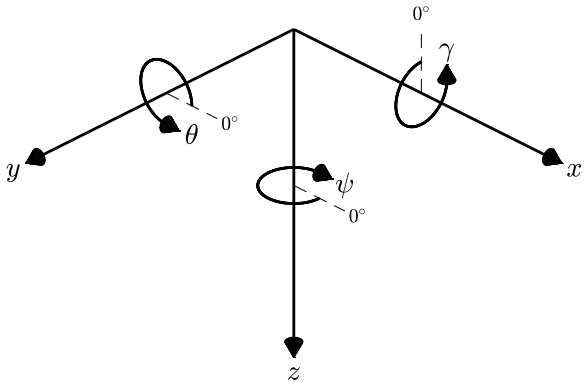

Figure 2.1: NED coordinate system

Sonar Workbench uses three reference frames: the element frame, the

array frame, and the body frame. All frames use the NED coordinate system, and each frame can be located relative to another by a combination of translations[1] and rotations.

The element frame is always located at the center of the element, with the element's maximum response axis aligned with the $+x$ axis. Exceptions to this alignment are the omnidirectional element, which has no maximum response axis, and the linear element, which the user specifies as initially parallel to one of the three axes (x,y,z) in the element frame. For planar piston elements, the element face lies in the element frame y-z plane.

Each element in an array can have arbitrary translation and rotation in the array frame. The array frame origin and orientation is entirely up to the user, but it is typical for planar arrays to be located in center of the array frame's y-z plane and for volumetric arrays' geometric center to be located at the origin of the array frame.

The entire array can also be arbitrarily translated and rotated relative to the body frame. For example, a planar array on the nose of a torpedo might have a simple translation along the body frame x axis, while a flank array might have translations along the body x and y axes plus a rotation ψ about the body frame z axis. Figure 2.2 shows an example of element, array, and body frames for a conformal flank array.

Beam patterns are always computed in the body frame as a function of azimuth angles ψ and elevation angles θ. Azimuth and elevation angles are measured from the body frame $+x$ axis. To calculate beam patterns in the array frame, set the array position and orientation vectors to [0;0;0]. More details about element, array, and body frame alignments will be explained in Chapters 3 and 4.

[1] displacement along the x, y, or z axes

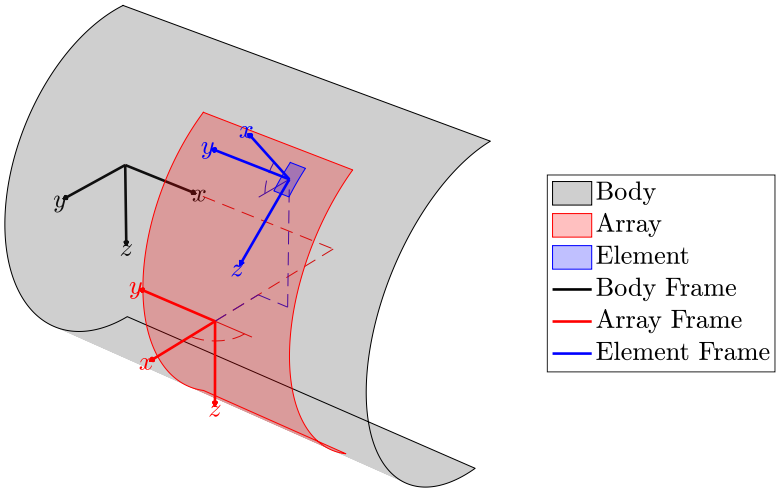

Figure 2.2: Body, array, and element frames for flank array example

Chapter 3

Elements

The transducer element is the fundamental building block for arrays and the starting point for analysis in Sonar Workbench. For the purpose of generating and analyzing beam patterns, specific electromechanical transduction methods do not need to be modeled; instead, the element's acoustic properties can be captured by modeling vibrations of the element's wetted surface. Hereafter, references to an element's geometry refer to the geometry of the element's wetted surface or face. Sonar Workbench treats each element as a uniformly vibrating surface, which is to say that it only models each surface's fundamental mode of vibration. Analyzing element response requires first, defining the element geometry, and second, evaluating that geometry at a specific acoustic wavelength to produce an element pattern.

3.1 Element definition

Sonar Workbench includes support for the element types listed in Table 3.1.

An element structure holds the parameters that define the element geometry. The element structure must contain the `.type` field with an integer corresponding to the elements listed in Table 3.1, which selects a `.m` file that generates the corresponding element pattern. Table 3.1 lists the built-in element type enumerations.

The field `.baffle` dictates whether the element should be baffled by the element frame y-z plane (e.g. arrays mounted to platforms) or unbaffled (e.g. towed arrays, sonobuoys). The omnidirectional, uniform line, and cosine elements can all be used with and without baffling. The piston el-

Table 3.1: Included element types

Element Type	Enumeration
Omnidirectional	0
Cosine	1
Uniform Line	2
Circular Piston	3
Rectangular Piston	4
Hexagonal Piston	5
Annular Piston	6

ements' element patterns are all derived from equations assuming the piston is mounted in an infinite rigid baffle; therefore, it is recommended that the user set these element's `.baffle=1` for best results. Diffraction effects caused by finite baffle dimensions are beyond the scope of Sonar Workbench.

The field `.params_m` is a three-element column vector defining the element geometry. A different number of parameters are required to define the shape of different element types. The components of `.params_m` for each element type are listed in Table 3.2. Note that idealized omnidirectional and cosine elements do not use require shape parameters, so all values can be set to 0 for these element types. As indicated by the field name, dimension units are in meters.

Table 3.2: Element parameter fields

Element Type	Vector Component	Description
Uniform Line	1	length along x-axis (m)
	2	length along y-axis (m)
	3	length along z-axis (m)
Circular Piston	1	radius (m)
Rectangular Piston	1	width (m)
	2	height (m)
Hexagonal Piston	1	inscribed circle radius (m)
Annular Piston	1	outer radius (m)
	2	inner radius (m)

For a line element, only one of the `params_m` components should be

nonzero, corresponding to the axis along which the line array is aligned. Listing 3.1 shows the contents of `SampleElement.m`, which defines a rectangular piston element.

Listing 3.1: `SampleElement.m`

```
%% Element Design
Element.type = 4;                   % Rectangular piston
Element.params_m = [lambda/2; ...   % Element face width, m
                    lambda/4; ...   % Element face height, m
                    0];
Element.baffle = 1;                 % Hard Baffle
```

The user is free to add additional fields to the element structure for their own purposes. These will be ignored by Sonar Workbench.

3.2 Element patterns

Element geometry, coupled with an acoustic wavelength, defines the element pattern. The element pattern is the element's far-field directional response as a function of wavelength, azimuth and elevation, and it can be thought of as a spatial filter. Elements are assumed to be transducers, capable of transmitting and receiving sound, so there is no distinction between transmit and receive element patterns.

Sonar Workbench uses the acoustic wavelength, λ, to calculate element patterns because it combines the frequency and sound speed into a single term. It is related to sound speed c, frequency f in Hz or ω in rad/s, and wavenumber k in m^{-1} by

$$\lambda = \frac{c}{f} = \frac{2\pi c}{\omega} = \frac{2\pi}{k}.$$

At its most general, the element pattern is the three-dimensional Fourier transform of the element's complex aperture function, $A(\lambda, x, y, z)$,

$$E(\lambda, \theta, \psi) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} A(\lambda, x, y, z) e^{j2\pi\left(\frac{\cos\theta\cos\psi}{\lambda}x + \frac{\cos\theta\sin\psi}{\lambda}y + \frac{\sin\theta}{\lambda}z\right)} dx dy dz, \tag{3.1}$$

For the piston elements, the element pattern reduces to a two-dimensional Fourier transform, since the element face lies in the y-z plane,

$$E_{piston}(\lambda, \theta, \psi) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} A(\lambda, y, z) e^{j2\pi\left(\frac{\cos\theta\sin\psi}{\lambda}y + \frac{\sin\theta}{\lambda}z\right)} dy dz, \tag{3.2}$$

and for the uniform line array, it further reduces to a one-dimensional Fourier transform,

$$E_{line}(\lambda, \theta, \psi) = \int_{-\infty}^{\infty} A(\lambda, y) e^{j2\pi \frac{\cos\theta \sin\psi}{\lambda} y} dy, \tag{3.3}$$

for a line array aligned with the y axis. The finite element extents make the integration limits finite. For the simple elements included with Sonar Workbench, the assumption of uniform surface motion means that the aperture function is real-valued and equal to 1 over the entire element surface. This simplifies the integration for certain element geometries. These integrals have analytic solutions, which Sonar Workbench uses instead of evaluating the integrals numerically.

Element patterns are normalized such that they have unity gain along their maximum response axis. Figure 3.1 shows element patterns for each of the included element types listed in Table 3.1 for a wavelength equal to half of the element's maximum dimension. The uniform line element is aligned with the y axis, and the cosine element is aligned with the x axis. Note that the omnidirectional and cosine element patterns do not depend on wavelength.

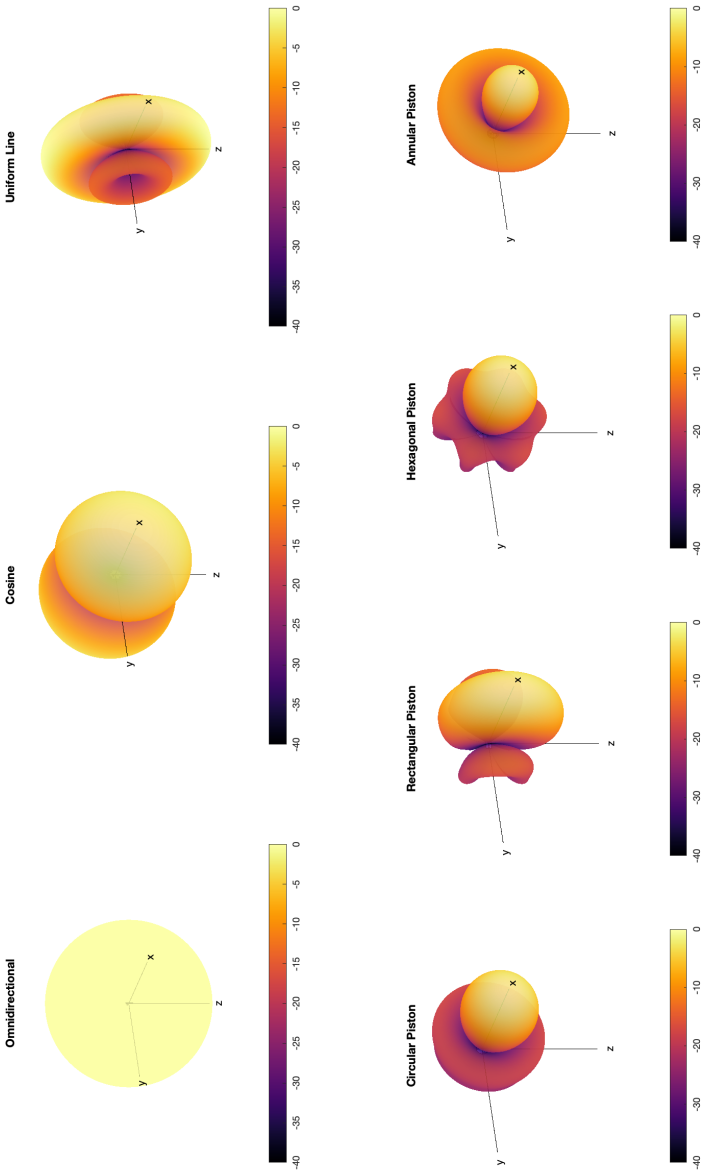

Figure 3.1: Element patterns for included element types (magnitude in dB)

Chapter 4

Arrays

An array is a collection of elements. Each element in an array has a position and orientation in the array frame. The array itself also has a position and orientation in the body frame. Sonar Workbench supports arrays with mixed element types, e.g. vector sensor arrays with a mix of hydrophones (omnidirectional elements) and accelerometers (cosine elements). The array structure has fields which contain this information.

The field `.Ne` holds the integer number of elements in the array. The position, orientation, and element type fields are all column vectors of length `.Ne`. The user is free to choose any convenient element order, as long as that order is consistent across all fields. A (3×`.Ne`) matrix determines the element positions in the array frame, `.ePos_m`, with rows corresponding to x, y, and z, respectively, in units of meters. An additional (3×`.Ne`) matrix determines the element orientations in the array frame, `.eOri_deg`, with rows corresponding to roll, pitch, and yaw, in degrees. Array position and orientation are held in the (3×1) vectors `.aPos_m` and `.aOri_deg`, respectively. Table 4.1 summarizes the fields in the array structure.

4.1 Arrays with uniform element types

When all the array elements are identical, the array structure does not need the `.eindex` field. An example array structure is found in `SampleArray.m`, shown in Listing 4.1.

This planar array contains 50 rectangular elements arranged in a grid 5 elements wide by 10 elements high. The geometry is shown in Fig. 4.1, with

Table 4.1: Array structure fields

Field	Description
`.Ne`	number of elements
`.Net`	number of unique element types
`.Element`	vector of length `.Net` element structures
`.ePos_m`	element position matrix (m)
`.eOri_deg`	element orientation matrix (°)
`.eindex`	element type index (optional)
`.aPos_m`	array position vector (m)
`.aOri_deg`	array orientation vector (°)

Listing 4.1: `SampleArray.m`

```
%% Array Design
Nw = 5;                                 % Number of elements wide
Nh = 10;                                % Number of elements high
dy = Element.params_m(1);               % Horizontal spacing, m
dz = Element.params_m(2);               % Vertical spacing, m
Array.Ne = Nw*Nh;                       % Number of elements
Array.Net = 1;                          % Homogeneous array
Array.Element = Element;
Array.ePos_m = [zeros(1,Array.Ne); ...
                repmat((-(Nw-1)/2:(Nw-1)/2)*dy,1,Nh); ...
                reshape(repmat((-(Nh-1)/2:(Nh-1)/2)*dz,Nw,1),1,Array.Ne)];
                                        % Element position matrix, m
Array.eOri_deg = zeros(3,Array.Ne);     % Element oritnation matrix, deg
Array.eindex = ones(1,Array.Ne);        % Element type index
Array.aPos_m = zeros(3,1);              % Array position matrix, m
Array.aOri_deg = zeros(3,1);            % Array orientation matrix, deg
```

elements numbered according to their order in the array structure.

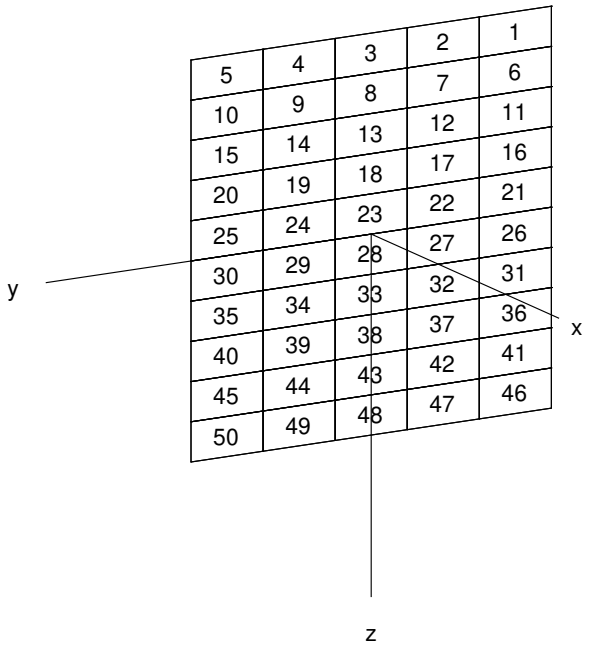

Figure 4.1: Example rectangular planar array

4.2 Arrays with mixed element types

For mixed-element arrays, the `.eindex` field must be a row vector of length `.Ne` integers. The user must first define an element structure array, with one entry for each unique element type. The `.eindex` field for each element must contain the 1-based index into the element structure array corresponding to its element type. The maximum value allowed is `.Net`.

An example is included in `SampleVSCardioid.m` for a simple vector sensor. The relevant portion is shown in Listing 4.2. This vector sensor consists of four elements: a hydrophone and three orthogonal accelerometers. The three accelerometers are identical except for their orientations, so there are a total of two unique elements. The hydrophone is `VS.Element(1)`, and the accelerometer is `VS.Element(2)`. The array structure is named `VS`, and the hydrophone is the first element, followed by the x, y, and z accelerometers. This makes the first entry in `.eindex` equal to 1 and the rest of the entries equal to 2. Rows two and three of `.eOri_deg` rotate elements 3 and 4 from the x axis to the y and z axes, respectively.

Listing 4.2: `SampleVSCardioid.m`

```
Element(1).type = 0;
Element(1).baffle = 0;
Element(1).params_m = [0;0;0];
Element(2).type = 1;
Element(2).baffle = 0;
Element(2).params_m = [0;0;0];
%% Define Vector Sensor
VS.Ne = 4;
VS.Net = 2;
VS.Element = Element;
VS.ePos_m = zeros(3,VS.Ne,1);
VS.eOri_deg = [0  0  0  0; ...
               0  0  0 90; ...
               0  0 90  0];
VS.eindex = [1 2 2 2];
VS.aPos_m = [0;0;0];
VS.aOri_deg = [0;0;0];
```

Chapter 5

Beams

Once an array and element(s) have been defined, the user can generate any number of beam patterns by defining a set of wavelength-dependent, complex weights, $w(\lambda)$. Multiple beams can be calculated for the same array by changing these weights. The beam structure must contain these weights in a single field called `.ew`, which is a row vector of length `Array.Ne`.

In conventional (plane wave) beamforming, amplitude weights are used to adjust the width of the beam's main lobe and the magnitude of its sidelobes, and phase weights are used to steer the beam in azimuth and elevation. For a source in the array's acoustic near field, phase weights can also be used to focus the beam at a specified range. In adaptive beamforming, a complex set of weights is chosen to simultaneously maximize one criterion (e.g. signal level) and minimize one or more other criteria (e.g. noise level). Iterative methods that calculate complex weights to produce beams with prescribed main lobe dimensions and side lobe levels also exist. Although the details of near-field focusing and adaptive beamforming are beyond the scope of this user's guide, Sonar Workbench can generate beam patterns for any complex weights the user derives.

5.1 Conventional beamforming

Conventional beamforming independently calculates amplitudes, a_i, and phases, ϕ_i to create complex weights $w_i = a_i e^{j\phi_i}$. Listing 5.1 shows an example of one beam that can be formed for the planar array example from Chapter 4. This example contains both amplitude and phase weights.

Listing 5.1: `SampleBeam.m`

```
%% Load Array Definition
SampleArray
%% Amplitude Weights
hweights = [0.3185 0.7683 1.0000 0.7683 0.3185];
vweights = [0.2575 0.4300 0.6692 0.8780 1.0000 1.0000 0.8780 0.6692 0.4300 0.2575];
ea = repmat(hweights,1,Nh) .* ...
     reshape(repmat(vweights,Nw,1),1,Array.Ne);
% Phase Weights
if ~exist('psi0','var')
    psi0 = 0;
end
if ~exist('theta0','var')
    theta0 = 0;
end
ep = exp(-1i*2*pi*cosd(-theta0).*sind(psi0)/lambda*Array.ePos_m(2,:)).* ...
     exp(-1i*2*pi*sind(-theta0)/lambda*Array.ePos_m(3,:));
%% Complex Weights
Beam.ew = ea.*ep;
clear ea ep
```

5.1.1 Amplitude weights

Amplitude shading exploits the Fourier transform relationship between the beam's aperture function and its far field beam pattern. Consequently, sonar designers often use weights, or shading coefficients, taken from standard window functions. Examples include Uniform, Hanning, Hamming, and Chebyshev, as shown in Fig. 5.1. The example beamformer uses two Chebyshev windows, one of length 5 in the horizontal direction, and one of length 10 in the vertical direction. The total weight for each element is the product of the two windows evaluated at its coordinate the horizontal and vertical direction.

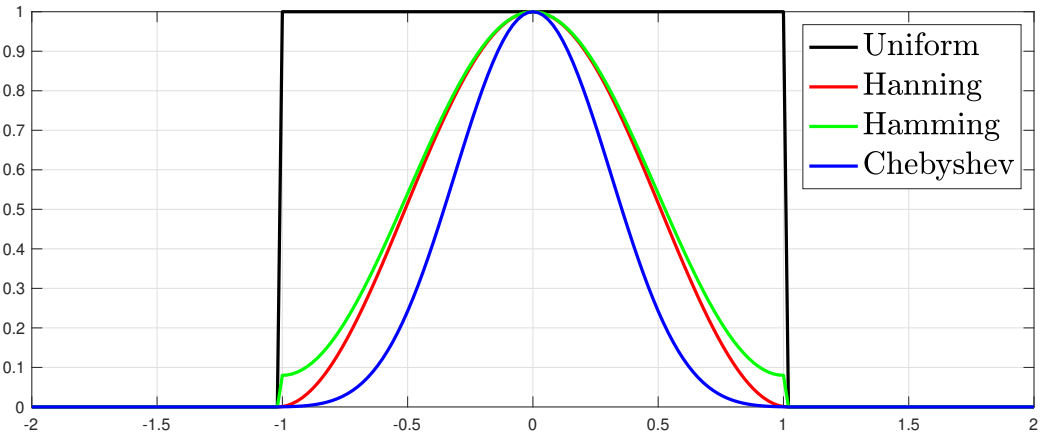

Figure 5.1: Common window functions used for amplitude shading

Since the final beam pattern will be normalized for unity gain along its maximum response axis, the user is not required to normalize the amplitude weights or limit their amplitude. However, it is best to avoid negative amplitude weights and instead represent negative numbers with phase weights.

5.1.2 Phase weights

Without phase weights, i.e. the beam pattern weights are purely real, the beam pattern is unsteered. The user can steer the beam to an arbitrary elevation and azimuth, (θ_0,ψ_0) using the following formula for the i^{th} element's phase,

$$\phi_i(\lambda, \theta_0, \psi_0) = -2\pi \left(\frac{\cos\theta_0 \cos\psi_0}{\lambda} x_i + \frac{\cos\theta_0 \sin\psi_0}{\lambda} y_i + \frac{-\sin\theta_0}{\lambda} z_i \right). \quad (5.1)$$

Note that the negative sign in the third term in Eq. (5.1) accounts for the definition of positive elevation angles in the negative z direction, as shown in Fig. 2.1.

Chapter 6

Analysis

Sonar Workbench includes built-in functions to assist with analysis. Versions 1.0-3.0 concentrated on array building, conventional beam pattern synthesis, and beam pattern analysis. Planned upgrades include support for adaptive beamforming, element and array radiation impedance, and array gain in anisotropic noise.

6.1 Plotting arrays

The function `PlotArray.m` creates a 3D plot of an array. Usage instructions are shown in Listing 6.1. The user must pass the array structure as the first parameter to `PlotArray.m`. This will produce a physical plot of the array, as shown in Fig. 4.1. There are three optional inputs to `PlotArray.m`. The user can enter empty brackets `[]` for any optional parameters that are not needed. If the user inputs a beam structure in the second parameter, the elements will be filled with a solid color whose opacity is proportional to the amplitude weights, as shown in Fig. 6.2. To plot the array in an existing figure axis, pass a handle to that axis as the third parameter, `hax`. The Matlab function `gca` can be used to specify the current axis. The fourth parameter lets the user plot the array scaled to dimensions of 1 unit = 1 meter. This is helpful if the array is to be overlaid on a user-generated plot of its host platform. The fifth and final parameter lets the user specify the color to use for each element. This input is a 3-element normalized vector containing red, green, and blue color components.

Listing 6.1: `PlotArray.m`

```
%% function PlotArray(Array)
% function PlotArray(Array,Beam)
% function PlotArray(Array,Beam,hax)
% function PlotArray(Array,Beam,hax,actualsize)
% function PlotArray(Array,Beam,hax,actualsize,color)
%
% Plots the physical layout of an array, including amplitude shading if a
% Beam is defined.
%
% Inputs:
%           Array       - Array structure with the following fields
%           [required ]
%               .Ne         - Number of elements
%               .Net        - Numer of unique element types
%               .Element    - Array of length .Net element structures
%                             with the following fields
%                 .type       - Element pattern generator enumeration
%                 .params_m   - Element shape parameter vector, see element
%                               pattern files for details
%               .ePos_m     - Element position matrix, m
%               .eOri_deg   - Element normal orientation matrix, deg
%               .aPos_m     - Array position vector, m
%               .aOri_deg   - Array orientation vector, deg
%           [optional ]
%               .eindex     - Vector of indices into element structure
%                             vector to support non-uniform element arrays
%
% Optional Inputs:
%           Beam        - Beam structure with the following required fields
%               .ew     - Complex element weight vector
%           hax         - Handle to axis for plotting in an existing figure
%           actualsize  - Plot array elements without scaling
%           color       - RGB triple with elements in [0,1]
%
```

6.2 Calculating beam patterns

`BeamPattern.m` calculates the beam pattern for wavelength λ, elevation angles θ, and azimuthal angles ψ as

$$BP(\lambda,\theta,\psi) = \sum_{i=1}^{N_e} w_i(\lambda,\theta_0,\psi_0) E_i(\lambda,\theta,\psi) e^{j2\pi\left(\frac{\cos\theta\cos\psi}{\lambda}x_i + \frac{\cos\theta\sin\psi}{\lambda}y_i + \frac{\sin\theta}{\lambda}z_i\right)}, \tag{6.1}$$

where, for the i^{th} element in the array, $w_i(\lambda,\theta_0,\psi_0)$ is the complex weight, $E_i(\lambda,\theta,\psi)$ is the element pattern, rotated according to the element orientation (γ_i,θ_i,ψ_i), and (x_i,y_i,z_i) are the element coordinates. This computation is performed in the body frame. To calculate the beam pattern in the array frame, set the array position and orientation to [0;0;0].

The beam pattern is returned in complex linear units. This retains the amplitude and phase information from the beam. By default, the amplitude is normalized by dividing the output by the sum of the beam weight

amplitudes,

$$BP_{norm}(\lambda, \theta, \psi) = \frac{BP(\lambda, \theta, \psi)}{\sum_{i=1}^{N_e} |w_i(\lambda, \theta_0, \psi_0)|}. \tag{6.2}$$

For an unsteered beam, this produces an output with amplitude 1 along the maximum response axis. Steering beams off axis can result in a peak beam amplitude less than 1. The user can also choose to normalize the beam pattern relative to the peak calculated value or skip the normalization step altogether.

Usage instructions for `BeamPattern.m` are shown in Listing 6.2. The first two input parameters are the array and beam structures. The third parameter is the wavelength, λ, in meters. This must be a scalar.

The final required parameters are the elevation and azimuthal angles over which the beam pattern is to be calculated, in degrees. These parameters can be scalars, vectors, or matrices, but they must have compatible dimensions. If both are scalars, the output, `BP`, will be a scalar. If both are column vectors of the same length or matrices with identical dimensions, `BeamPattern.m` treats them as coordinate pairs at which to evaluate the beam pattern. If `theta` is a column vector and `psi` is a row vector, `BeamPattern.m` uses the Matlab function `ndgrid` to calculate a matrix `BP`, with rows corresponding to constant θ and columns corresponding to constant ψ. The user can generate their own matrix inputs using `[Theta,Psi] = ndgrid(theta,psi);` for vectors `theta` and `psi`.

After the required parameters, a single optional parameter, `NormMethod`, selects the method to use for beam pattern normalization, as shown in Table 6.1.

Table 6.1: `NormMethod` options for `BeamPattern.m`

`NormMethod`	**Description**
0	No normalization
1	Normalize by beam pattern weights
2	Normalize by peak calculated beam pattern value

Listing 6.2: `BeamPattern.m`

```
%% function BP = BeamPattern(Array,Beam,lambda,theta,psi)
% function BP = BeamPattern(Array,Beam,lambda,theta,psi,NormMethod)
%
% Computes the beam pattern for an array of elements at coordinates
% (ex_m,ey_m,ez_m) rotated (egamma_deg,etheta_deg,epsi_deg) from the x axis
% with complex element weights ew at wavelength lambda over elevation
% angles theta and azimuthal angles psi. Supported element types are
% omnidirectional, linear, cosine, circular piston, rectangular piston,
% hexagonal piston, and annular piston.
%
% Inputs:
%           Array       - Array structure with the following fields
%           [required ]
%               .Ne         - Number of elements
%               .Net        - Numer of unique element types
%               .Element    - Array of length .Net element structures
%                             with the following fields
%                 .type       - Element pattern generator enumeration
%                 .params_m   - Element shape parameter vector, see element
%                               pattern files for details
%               .ePos_m     - Element position matrix, m
%               .eOri_deg   - Element normal orientation matrix, deg
%           [optional ]
%               .eindex     - Vector of indices into element structure
%                             vector to support non-uniform element arrays
%           Beam        - Beam structure with the following required fields
%               .ew         - Complex element weight vector
%           lambda          - Acoustic wavelength, m
%           theta           - Elevation angle vector or matrix, deg
%           psi             - Azimuthal angle vector or matrix, deg
%
% Optional Inputs:
%           NormMethod  - Normalization method enumeration
%                         0 = No normalization
%                         1 = Weight normalization
%                         2 = Peak normalization
%
% Outputs:
%           BP          - Beam pattern, complex linear units
%
```

6.3 Extracting 2D slices from 3D beam patterns

The function `ExtractBeamSlice.m` extracts a 2D slice from a 3D beam pattern by extracting all values that lie in a plane with user-specified orientation. Common usage includes extracting horizontal and vertical beam patterns for generating 2D plots, but the plane can also be rotated to different angles between horizontal and vertical.

Listing 6.3 shows usage instructions for `ExtractBeamSlice.m`. All input parameters are required. The first two are the vertical and horizontal angles over which the beam pattern is evaluated, and the third input is the 3D beam pattern. The third input is a three-element orientation vector containing the roll, pitch, and yaw angles (in degrees) that rotate the default horizontal plane to the desired orientation. The first element of `Ori` is the roll param-

eter. Set this value to 0 for a horizontal slice and set it to -90 for a vertical slice. Other rotational angles can be useful, e.g. 60 for measuring the beam pattern of a haxagonal piston element along one of its axes of symmetry. The final two elements of `Ori` are the elevation and azimuth coordinates of the 2D slice's angular origin. Setting these elements to θ_0 and ψ_0 places the center of the extracted 2D slice at a steered beam's maximum response axis. `ExtractBeamSlice.m` returns a vector of complex beam pattern values, `BPslice`, and a vector of angles, `phi`.

Listing 6.3: `ExtractBeamSlice.m`

```
%% function [BPslice,phi] = ExtractBeamSlice(theta,psi,BP,Ori)
%
% Extracts a 2D slice from a 3D beam pattern along the great circle with
% the specified orientation. Ori(2:3) specifies the center of the  slice in
% the (theta,psi) plane, and Ori(1) sets the angle of the slice within that
% plane, measured clockwise from the positive psi axis. For a vertical
% slice (phi=theta), set Ori(1) = -90, and for a horizontal slice
% (phi=psi), set Ori(1) = 0.
%
% Inputs:
%           theta   - Elevation angle vector, deg
%           psi     - Azimuthal angle vector, deg
%           BP      - Beam pattern matrix with elevation angles in rows,
%                     azimuthal angles in columns, complex linear units
%           Ori     - Slice orientation vector, deg
%
% Outputs:
%           BPslice  - Beam pattern slice, complex linear units
%           phi      - Slice angle vector relative to Ori(2:3), deg
%
```

Since `ExtractBeamSlice.m` outputs values defined over an angular span with fixed 1° spacing, and it uses interpolation to estimate values from coordinates that do not map directly to coordinates in the 3D beam pattern, it is best suited for visualization when a 3D beam pattern has already been calculated. For more precise results, evaluate `BeamPattern.m` at the desired θ and ψ corresponding to the plane of interest directly.

6.4 Plotting 2D beam patterns

The function `Plot2DBP.m` plots the two dimensional beam pattern magnitude with decibel scaling. Listing 6.4 shows usage instructions. The two required inputs are the angle vector, ϕ in degrees, and the beam pattern vector.

The first optional input, `PlotType` selects from two different rectangular plots and two different polar plots, as listed in Table 6.2. The first two plot types are suitable for displaying horizontal beam pattern slices or slices cut through any plane, while the third and fourth plot types are suitable for vertical beam pattern slices, according to the coordinate conventions used

by Sonar Workbench. Figure 6.1 shows examples of each plot type for 2D horizontal and vertical slices through the rectangular planar array's 3D beam pattern.

Table 6.2: `PlotType` options for `Plot2DBP.m`

`PlotType`	**Description**
1	Rectangular plot, 0° up, $+\phi$ right
2	Polar plot, 0° up, $+\phi$ clockwise
3	Rectangular plot, 0° right, $+\phi$ up
4	Polar plot, 0° right, $+\phi$ counterclockwise

The second optional input, `dBScale`, is a two-element vector with the minimum and maximum magnitudes in dB. The default values are -40 and 0 dB. It is common to use 0 dB for the maximum value, since the beam pattern is normalized by default.

The third optional input is the axis handle, `hax`, in which to plot the beam. This input has the same functionality as it does in `PlotArray.m`, and it allows the beam to be plotted with the array on the same axis. Matlab function `gca` can be used to specify the current axis.

`Plot2DBP.m` returns a single optional output, which is a handle to the plotted data. This allows the user to change the plot's appearance, such as line color and weight or marker type and size.

Listing 6.4: `Plot2DBP.m`

```
%% function h = Plot2DBP(phi,BP)
% function h = Plot2DBP(phi,BP,PlotType)
% function h = Plot2DBP(phi,BP,PlotType,dBScale)
% function h = Plot2DBP(phi,BP,PlotType,dBScale,hax)
%
% Plots a 2D beam. Beam pattern is defined over angles phi. User can choose
% to plot in a new figure or in an existing axis. Optionally returns a
% handle to the plot to allow the user to change line properties such as
% color, width, and marker.
%
% Inputs:
%           phi      - Angle vector, deg
%           BP       - Beam pattern vector, complex linear units
%
% Optional Inputs:
%           PlotType - Plot type enumeration
%                      1 = Rectangular plot, phi = x axis
%                      2 = Polar plot, phi = 0 north, CW+
%                      3 = Rectangular plot, phi = y axis
%                      4 = Polar plot, phi = 0 east, CCW+
%           dBScale  - Magnitude range for plot, [dBmin, dBmax], dB
%           hax      - Handle to axis for plotting in an existing figure
%
% Optional Outputs:
%           h        - Handle to plot
%
```

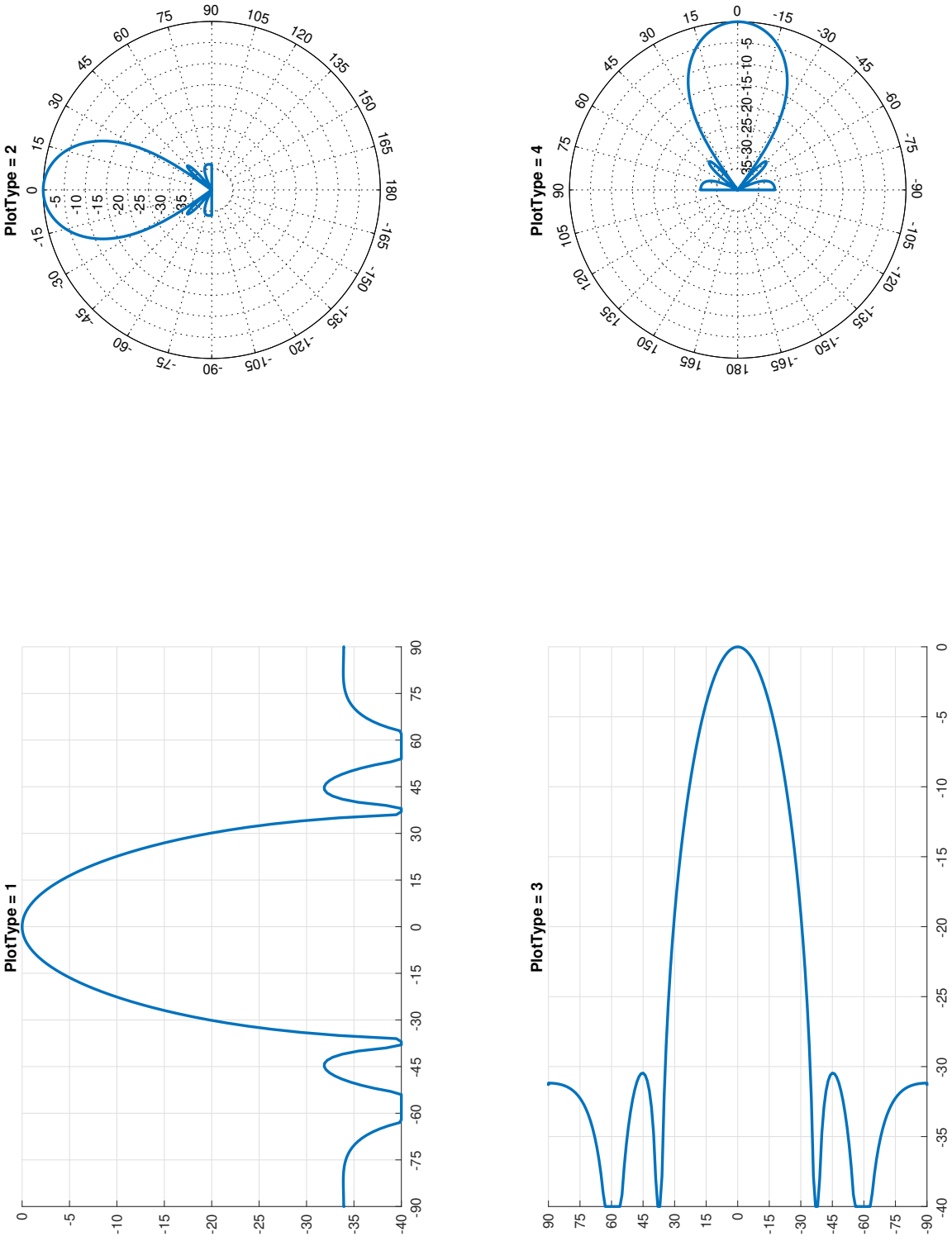

Figure 6.1: Example slices through rectangular planar array beam pattern

6.5 Plotting 3D beam patterns

The function `Plot3DBP.m` plots the three dimensional beam magnitude with decibel scaling. Usage instructions are shown in Listing 6.5. The three required inputs are elevation and azimuthal angle vectors, θ and ψ in degrees, and the beam pattern matrix.

The first optional input, `PlotType`, selects a plot with the beam pattern surface and color proportional to magnitude when its value is 1, and it selects a pot with the beam pattern color proportional to magnitude but drawn on a surface of constant radius when its value is 2. Examples of these two options are shown in the left and right panels of Fig. 6.2 Those plots are for the example element, array, and beam defined in Listings 3.1, 4.1, and 5.1.

The second optional input, `dBScale`, is a two-element vector with the minimum and maximum magnitudes in dB. The default values are -40 and 0 dB. It is common to use 0 dB for the maximum value, since the beam pattern is normalized by default.

The third optional input is the axis handle, `hax`, in which to plot the beam. This input has the same functionality as it does in `PlotArray.m`, and it allows the beam to be plotted with the array on the same axis. Matlab function `gca` can be used to specify the current axis.

The final optional inputs allow the beam to be offset according to the array position in the body frame.

Listing 6.5: `Plot3DBP.m`

```
%% function Plot3DBP(theta,psi,BP)
% function Plot3DBP(theta,psi,BP,PlotType)
% function Plot3DBP(theta,psi,BP,PlotType,dBScale)
% function Plot3DBP(theta,psi,BP,PlotType,dBScale,hax)
% function Plot3DBP(theta,psi,BP,PlotType,dBScale,hax,aPos_m)
%
% Plots a beam pattern in 3D space. Beam pattern is defined over azimuthal
% angles psi and elevation angles theta. User can choose to plot in a new
% figure or in an existing axis.
%
% Inputs:
%           theta   - Elevation angle vector, deg
%           psi     - Azimuthal angle vector, deg
%           BP      - Beam pattern matrix with elevation angles in rows,
%                     azimuthal angles in columns, complex linear units
%
% Optional Inputs:
%           PlotType - Plot type enumeration
%                      1 = Plot beam dimensions in 3D
%                      2 = Plot beam magnitude on constant radius surface
%           dBScale  - Magnitude range for plot, [dBmin, dBmax], dB
%           hax      - Handle to axis for plotting in an existing figure
%           aPos_m   - Array position vector, m
%
```

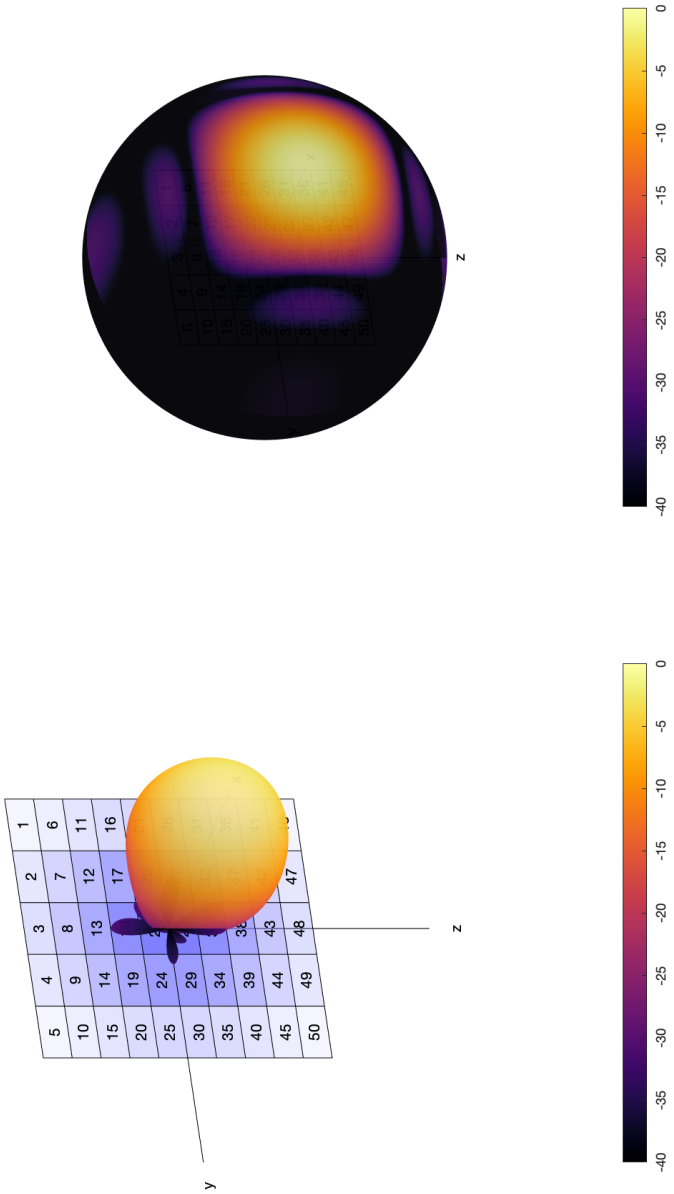

Figure 6.2: Example beam pattern for rectangular planar array

6.6 Calculating beam width

Beam width is the angular span over which the beam magnitude is within 3 dB of its peak value. Sonar Workbench contains two functions to calculate beam width: `BeamWidth.m` for 2D beams, which computes beam width in a single plane, and `BeamWidth3D.m` for 3D beams, which computes vertical and horizontal beam widths. Usage is shown in Listings 6.6 and 6.7.

Listing 6.6: `BeamWidth.m`

```
%% function BW = BeamWidth(ang,bp)
% function BW = BeamWidth(ang,bp,ang0)
% function BW = BeamWidth(ang,bp,ang0,renorm)
%
% Calculates the -3 dB beam width of a single beam slice. Peak location can
% be aided with optional parameter ang0. Beam width can be calculated using
% beam pattern amplitude as entered or by re-normalizing the beam pattern
% about the desired peak value.
%
% Inputs:
%           ang     - Angle vector, deg
%           bp      - Beam pattern slice vector, complex linear units
%
% Optional Inputs:
%           ang0    - Initial estimate for peak search, deg
%           renorm  - Re-normalize beam pattern about peak value
%
% Outputs:
%           BW      - Beam width, deg
%
```

Listing 6.7: `BeamWidth3D.m`

```
%% function [BWV, BWH] = BeamWidth3D(theta,psi,BP)
% function [BWV, BWH] = BeamWidth3D(theta,psi,BP,theta0,psi0)
% function [BWV, BWH] = BeamWidth3D(theta,psi,BP,theta0,psi0,renorm)
%
% Calculates the horizontal and vertical -3 dB beam widths of a 3D beam
% pattern. Peak location can be aided with optional parameters psi0 and
% theta0. Beam width can be calculated using beam pattern amplitude as
% entered or by re-normalizing the beam pattern about the desired peak
% value.
%
% Inputs:
%           theta   - Elevation angle vector, deg
%           psi     - Azimuthal angle vector, deg
%           BP      - Beam pattern matrix with elevation angles in rows,
%                     azimuthal angles in columns, complex linear units
%
% Optional Inputs:
%           theta0  - Initial elevation estimate for peak search, deg
%           psi0    - Initial azimuth estimate for peak search, deg
%           renorm  - Re-normalize beam pattern about peak value
%
% Outputs:
%           BWV      - Vertical beam width, deg
%           BWH      - Horizontal beam width, deg
%
```

There are two required parameters for `BeamWidth.m`. The first, `ang`, is a vector of angles in degrees over which the beam pattern slice is defined, and the second, `bp`, is the beam pattern at those angles, in linear units. The user can assist the function in locating the correct peak value to search around by

specifying an approximate peak angle in the first optional parameter, `ang0`. The final optional parameter, `renorm`, instructs the function to re-normalize the beam pattern to have a peak magnitude of 0 dB before searching for the -3 dB angles. This can be useful for a steered beam whose peak magnitude is less than 0 dB.

The output, `BW`, is the measured beam width in degrees. The function attempts to interpolate between input grid angles using spline interpolation of the beam magnitude in dB. The user can get a more precise measurement by using finer angle sampling. If `BeamWidth.m` can not calculate a valid beam width, it returns `NaN`.

`BeamWidth3D` has similar usage but with expanded support for an additional dimension. Its first three parameters are required, and correspond to angle vectors θ and ψ in degrees and beam pattern matrix `BP` in linear units. The first two optional parameters, `theta0` and `psi0` serve the same purpose as `ang0` in `BeamWidth.m` for the elevation and azimuthal angles, respectively. The final optional parameter, `renorm`, is identical to that parameter in `BeamWidth.m`. There are two outputs, `BWV` and `BWH`, which are the vertical and horizontal beam widths in degrees, respectively. For the example beam in Fig. 6.2, the vertical and horizontal beam widths are calculated to be 26° and 25.6°.

6.7 Calculating directivity index

Directivity Index (DI) is a measure of a beam's ability to filter spatially isotropic noise. It is the ratio of the noise level received by an omnidirectional sensor to the noise level received by the beam, expressed in dB. It is calculated from the Directivity (D) according to

$$DI(\lambda) = 10 \log_{10} D(\lambda), \tag{6.3}$$

where

$$D(\lambda) = \frac{4\pi}{\int_0^{\pi} \int_0^{2\pi} |BP(\lambda, \theta, \psi)|^2 d\psi d\theta}. \tag{6.4}$$

`CalculateDI.m` implements Eqs. (6.3) and (6.4) using numerical integration over a realized beam pattern. Usage is shown in Listing 6.8.

There are three required parameters. The first two are angle vectors θ and ψ in degrees, and the third is the beam pattern matrix `BP` in linear units.

For best results, θ should span 180° vertically, and ψ should span 360° horizontally. If instead, the beam pattern is defined as a subset of these angles, `CalculateDI.m` assumes the beam pattern value is exactly zero outside the defined region. There is a single output, `DI`, which is the directivity index in dB.

Listing 6.8: `CalculateDI.m`

```
%% function DI = CalculateDI(theta,psi,BP)
%
% Calculates the directivity index of beam pattern BP defined over
% elevation angles theta and azimuthal angles psi. (Directivity Index is a
% special case of array gain for isotropic noise.) It is assumed that theta
% spans +/-90 degrees. Any azimuths for which the beam pattern is not
% defined are assumed to contribute zero energy to the beam pattern.
%
% Inputs:
%           theta   - Elevation angle vector, deg
%           psi     - Azimuthal angle vector, deg
%           BP      - Beam pattern matrix with elevation angles in rows,
%                     azimuthal angles in columns, complex linear units
%
% Outputs:
%           DI      - Directivity Index, dB
%
```

Bibliography

[1] L. J. Ziomek, *An introduction to sonar systems engineering*. Boca Raton, FL: Taylor & Francis/CRC, 1st ed., 2017.