Kapok: Python Toolbox for PolInSAR Forest Height Estimation

User's Manual

Version 0.1

Michael Denbina

November 10, 2016

Contents

Chapter 1

Introduction

Kapok is Python library designed to aid in the post-processing, analysis, and use of polarimetric interferometric synthetic aperture radar (PolInSAR) data for ecological study and parameter retrieval. Currently, the main focus of the included modules is on forest height estimation. A couple of basic PolInSAR forest models from the literature are implemented, including inversion routines which allow forest height to be estimated from the PolInSAR data. As well, the toolbox aims to provide a unified framework for easy manipulation and analysis of PolInSAR data, allowing the user to focus on model and algorithm development rather than on basic operations like covariance matrix calculation or file I/O.

Currently, only data from NASA's Uninhabited Aerial Vehicle Synthetic Aperture Radar (UAVSAR) instrument is supported, through a module which imports the UAVSAR data into the data format used by Kapok. Other platforms could be implemented through the creation of new modules which translate other formats. Kapok stores the PolInSAR data on disk using the open source HDF5 data format, allowing easy manipulation, addressing, and slicing of the data, as well as file compression. The various PolInSAR data (e.g., the covariance matrix, the optimized coherences, the parameters of the viewing geometry, etc.) are stored as intuitive multi-dimensional datasets, all within a single HDF5 file for each scene. Both single-baseline and multi-baseline datasets can be imported, though at the moment, many of the inversion and processing functions are designed for the single-baseline case only.

Chapter 2

Installation and Setup

Kapok was developed on Python 3.X, with the following dependencies:

- NumPy
- SciPy
- h5py
- matplotlib
- Cython
- GDAL

The use of a package manager can be helpful, such as conda.

Kapok uses GDAL to perform geocoding of any output data products. Please ensure that the program "gdalwarp" is callable from the command line if you wish to use this functionality. If "gdalwarp" is not callable from the command line, the program can be installed from gdal.org.

To install this library, simply extract the files to a folder of your choice and add the main kapok directory (not the kapok subfolder containing the Python code!) to the PYTHONPATH environment variable. The command "import kapok" in the Python console will then import the core functionality of the library. Kapok contains a number of different modules which are described in detail in Chapter 5. The following chapter gives an overview of the Kapok software in the form of a walkthrough of the standard processing steps, from importing the data to exporting a geocoded map of forest height estimates.

Chapter 3

Getting Started

In this section, we will work through a basic processing chain involving the following steps: data import, some basic data visualization, coherence calculation, coherence optimization, forest height estimation, and geocoding of the estimated forest heights. For reference, see the "basic_processing_example.py" script in the examples directory. This section is a walk through of the same concepts in that example script, though in this document we generally go into more details regarding the different options available.

UAVSAR PolInSAR datasets come in the form of a collection of .slc (single-look complex) files which are flat binary files containing the complex-valued reflectivity for each pixel. There will be one .slc file for each polarization (HH, HV, VH, and VV), for each flight track. A baseline can be formed by pairing up any two different tracks. Along with the .slc files will be .llh files containing the latitude, longitude, and digital elevation model (DEM) values for each pixel. The DEM heights are from the DEM used by the UAVSAR processor, which is generally the SRTM DEM. These parameters, of course, do not depend on the polarization or track number. There will also be a .lkv file containing the look vector between each pixel and the reference track in ENU (East, North, Up) coordinates. There are also .baseline files containing the baseline information for each flight track, for each azimuth index of the image. For each .slc file there will also be an annotation file with the extension .ann containing the metadata information, including a list of all the tracks in the processed stack.

The kapok.uavsar module handles the import of this UAVSAR data into a single HDF5 file, calculating the products (such as the covariance matrix elements, and the k_z and incidence angle values) necessary for further analysis. Before we proceed, note that all of the Kapok functions have function headers describing their purpose, input arguments, and return values. These are all readily accessible through Python's built-in help system (e.g., using "help(functionname)" in a Python shell).

We begin by navigating to the folder containing the UAVSAR data, and invoking a Python shell (ipython is recommended). We import the kapok module, and the kapok.uavsar module, then use the kapok.uavsar.load() function in order to import the UAVSAR data into a Kapok HDF5 file. After import, this function loads the data into a Kapok Scene object, and returns the result.

```
import kapok
import kapok.uavsar

scene = kapok.uavsar.load('
    pongar_27500_16009_002_160227_L090HH_01_BU.ann', 'pongara.hdf5
    ', compression='gzip', compression_opts=4, mlwin=[20,5])
```

'pongar_27500_16009_002_160227_L090HH_01_BU.ann' is the UAVSAR annotation filename for the first track, which will depend on the UAVSAR dataset we are working with. 'pongara.hdf5' is the output file where the resulting covariance matrix, incidence angle, k_z, and other imported data will be stored. This file will be created by the program. If the file already exists, an error message will be given. The **compression** keyword tells the function that we wish to use gzip to compress the imported datasets. This can be set to None if compression is not desired. The **compression_opts** keyword argument is an integer from 0 to 9, specifying the amount of compression to perform. Higher numbers result in more compression, with lower file size but increased CPU usage. The default value is 4. The **mlwin** keyword specifies the size of the multi-looking window (in azimuth, then range indices). The default value of [20,5] uses a window with 20 pixels in azimuth and 5 pixels in slant range. Using the standard UAVSAR pixel spacing, this will produce multi-looked imagery with pixel spacing of 12 m in azimuth and 8.3 m in slant range. Another common value is [12,3]. Larger windows generally produce smoother phase values, but will also result in larger pixel sizes, worsening the spatial resolution.

For large datasets containing many tracks, this function can take a while to run. If we wish to only import a subset of the data, the **azbounds** and **rngbounds** keyword arguments can be given to the kapok.uavsar.load function. Each keyword argument should be given a two-element list with the minimum and mazimum azimuth and range bounds of the desired subset. Similarly, if one wishes to import certain tracks, but not others, the **tracks** keyword argument can be provided. This should be a list containing all the track indices (starting at zero) that are specified for import. 'tracks=[0,1]' would import the first two tracks, but none of the others, for example, while 'tracks=[0,2]' would import the first and third tracks only. The order of the tracks is the same as in the annotation file, which will generally be ordered by time of acquisition.

For example, if we only wanted to import data for the first two tracks, for SLC rows from 15000–35000, and SLC columns from 1000–6000, we could execute the following command:

```
scene = kapok.uavsar.load('
    pongar_27500_16009_002_160227_L090HH_01_BU.ann', '
    pongara_subset.hdf5', tracks=[0,1], azbounds=[15000,35000],
    rngbounds=[1000,6000])
```

The program will provide progress updates to the Python console as it proceeds. The UAVSAR data is imported in the following order: metadata, covariance matrix, latitude/longitude/DEM, and finally platform and viewing geometry (from which the vertical wavenumber and incidence angle values are calculated). Once the function is complete, a Kapok Scene object will be returned, which here we have called "scene" for simplicity.

After a Kapok Scene HDF5 file has been created, we can quickly and easily load it in without needing to use the kapok.uavsar module. First, we clear the Scene object from memory (which, in the process, closes the HDF5 file):

```
del scene
```

Note that it can be a good idea make a backup copy of the HDF5 file, particularly if we are working with a large dataset that takes a large amount of time to import/process. We can load our newly created HDF5 file as follows:

```
scene = kapok.Scene('pongara.hdf5')
```

The Kapok Scene object contains easy access to all aspects of the PolInSAR dataset, and allows most of the processing functionality through its methods. This object-oriented interface is generally the easiest way to perform the standard processing steps. If more control over the process is desired, the underlying functions can be accessed directly, and we will provide an example of this a bit later in this chapter. For now, we will give an overview of most of the basic methods in the Scene class.

First, let's do some simple visualization of the data. This is performed through the kapok.vis module, which is accessed through the Scene **show** method. Here are a few examples:

```
scene.show('pauli') # Pauli RGB Image
scene.show('coh') # Complex Coherence Image
scene.show('coh',pol='HV') # Same, but for the HV polarization.
```

The first argument to scene.show() is called the image type. 'pauli' shows a Pauli RGB image, 'coh' shows a complex coherence image, 'power' or 'pow' shows an image of the backscattered power in dB, etc. There are a lot more options, some of which we will discuss in the following, and many of which are only relevant for certain image types. See the function header for more details using **help(kapok.Scene.show)**.

Complex coherences are displayed using the HSV color system, with the hue set by the phase of the complex coherence, and the saturation and value set by the magnitude of the complex coherence. This means that pixels with a high coherence magnitude will appear as bright, vibrant colors, with the specific color depending on the phase. Pixels with lower coherence magnitude, where we would expect the phases to be noisier, will appear dark gray and desaturated.

If you wish to view an image of the coherence magnitude or phase by themselves, you can do so using the following syntax:

```
scene.show('coh mag')
scene.show('coh ph')
```

Note that if scene.show('coh') is called without any further arguments, the default polarization to display is HH. This is also true for scene.show('power')—an image of the HH backscattered power is displayed. Similarly, if one is working with multi-baseline data, there is a **bl** keyword argument which specifies the baseline number of interest (starting at zero). By default, the first baseline is displayed. For example, **scene.show('coh',bl=1)** will show the complex coherence for the second baseline, for the HH polarization.

What if we'd like to show only a subset of the entire PolInSAR scene? This can be accomplished through the **bounds** keyword:

```
scene.show('pauli', bounds=(2000,3500,250,1250))
scene.show('pauli', bounds=(2000,3500))
```

When bounds contains four elements as in the first example, it is interpreted as the (starting azimuth, ending azimuth, starting range, ending range) indices to display. When bounds contains two elements as in the second example, it is interpreted as (starting azimuth, ending azimuth), with the plotted image spanning the full swath in the range direction.

If we'd like to save the plotted image to disk, we can use the **savefile** keyword, which specifies the desired filename:

```
scene.show('pauli', savefile='pauli.png')
scene.show('pauli', bounds=(2000,3500), savefile='pauli_subset.png
    ')
```

We can also display the DEM height, incidence angle, and the k_z values:

```
scene.show('dem') # DEM Heights
scene.show('inc') # Incidence Angle
scene.show('kz') # kz (for the first baseline)
```

If the default colormap bounds are too small or large, the minimum and maximum values can be set using the **vmin** and **vmax** keywords:

```
scene.show('pow', pol='HV')
scene.show('pow', pol='HV', vmin=-30, vmax=-6)
```

The second example will show the HV backscattered power, using a colormap ranging from -30 dB to -6 dB. The first example uses the default values (-25 dB and -3 dB, respectively). By default, power and Pauli RGB images are shown for the first track. The desired track index can be specified using the **tr** keyword:

```
scene.show('pow', pol='HV', tr=0)
scene.show('pow', pol='HV', tr=1)
```

The first example shows the first (master) track, while the second example shows the second (slave) track. Note that if we are opening a lot of figures, especially ones we are saving to disk and do not need to view, we can close all currently open figures using **scene.show('close')**.

Coherence region plotting can be performed using the kapok.region module, accessed using the Scene **region** method. For example:

```
scene.region(2150,615)
```

The first argument is the azimuth index of the pixel to plot, and the second argument is the range index. Note that this method also supports the **savefile** keyword, if one wishes to save the coherence region plot. An example plot is shown in Fig. 3.1. The region itself is shown as the solid blue line. Each of the standard lexicographic and Pauli basis coherences are plotted as a different colored dot. The HV coherence is shown in light green. Note the dark green and brown dots located on the edge of the coherence region. These are the coherences calculated using phase diversity coherence optimization—they are the complex coherences with maximum separation in the complex plane. In theory, the 'high' coherence shown in dark green has the lowest ground contribution of any polarization in the data. The 'low' coherence shown in brown, in contrast, has the highest ground contribution. The dashed green line is the line fitted to these optimized coherences. At the points where this line intersects the unit circle, there are two coherences plotted, one in black, and one in orange. The black dot is the ground coherence chosen by the algorithm, while the orange dot is the other alternate ground solution which was discarded. Note that while the HV coherence is close to the optimized high coherence, they are not equal. While the HV coherence will often (though not always!) contain a small amount of ground backscattering compared to the other polarizations, it is almost never the polarization with the absolute smallest amount of ground backscattering out of all possible polarization states. This is why coherence optimization can be useful.

The region method can also be used to create an interactive coherence region. These plots will have sliders on the bottom with which the user can input the parameters of the Random Volume over Ground (RVoG) model. The modelled coherences will be displayed on the plot, and updated in real-time. This allows the user to estimate the forest height, extinction, and temporal decorrelation parameter values that provide a good fit to the data. This can be called using:

```
scene.region()
scene.region(2150,615,'interactive')
```

In the first example, if no arguments are provided, the software will default to a blank interactive coherence region, since there is no actual data to plot (but the modelled coherences will still be displayed, and can be adjusted through the user interface). In the second example, this produces the same coherence region plot as before, but with the interactive functionality. This window

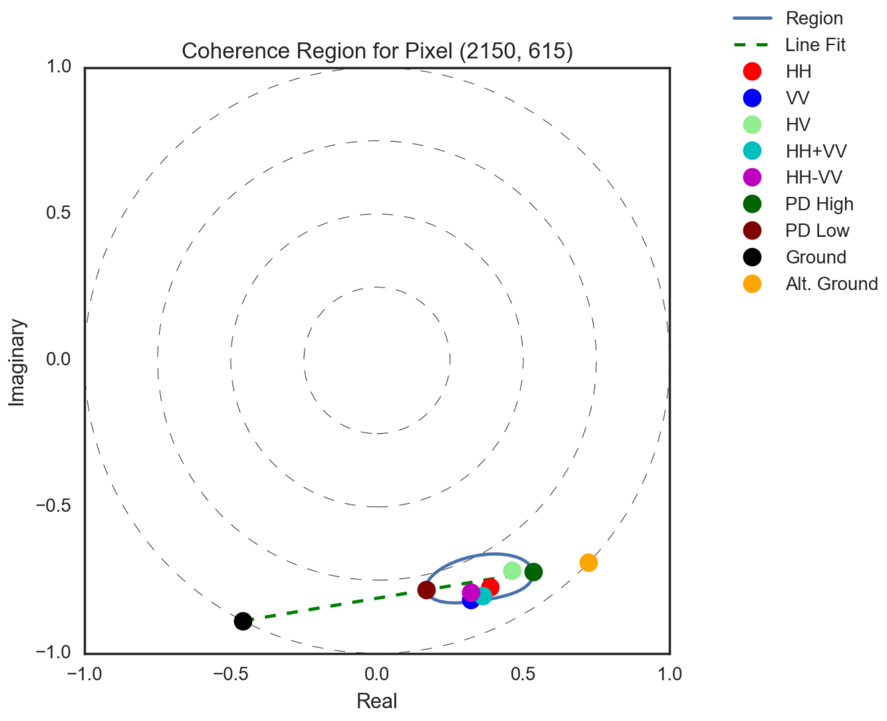

Figure 3.1: An example coherence region plot.

is shown in Fig. 3.2 (note that we've zoomed in to the coherence region). The "hv" slider specifies the forest height, h_v, in meters. The "Ext." slider specifies the extinction parameter, σ_x, in dB/m. The "Mu" slider specifies the ground-to-volume scattering ratio, μ. The "Alpha" slider specifies the temporal decorrelation magnitude of the volume coherence, α_{vt}.

The dashed black line shows the modelled volume coherences for a range of forest height values, starting at 0 m (at the ground coherence) and increasing towards the specified h_v slider value. The volume coherence is shown at the end of this dashed black line with a black X. The solid black line then connects the volume coherence to the modelled coherence with the ground-to-volume ratio specified by the "Mu" slider, which is plotted with another black X. Note that in this plot, we have adjusted the model parameter values to fit the data, assuming that the volume coherence is equal to the optimized high coherence, and neglecting the effects of temporal decorrelation (the "Alpha" slider set to 1.0). In this case, the estimated forest height is approximately 44 m. The assumptions we have made (of zero ground contribution in the high coherence, and negligible temporal decorrelation) are not necessarily true. Note that different forest height values could be found by varying the parameter sliders which still fit the observations. For more information on the RVoG model

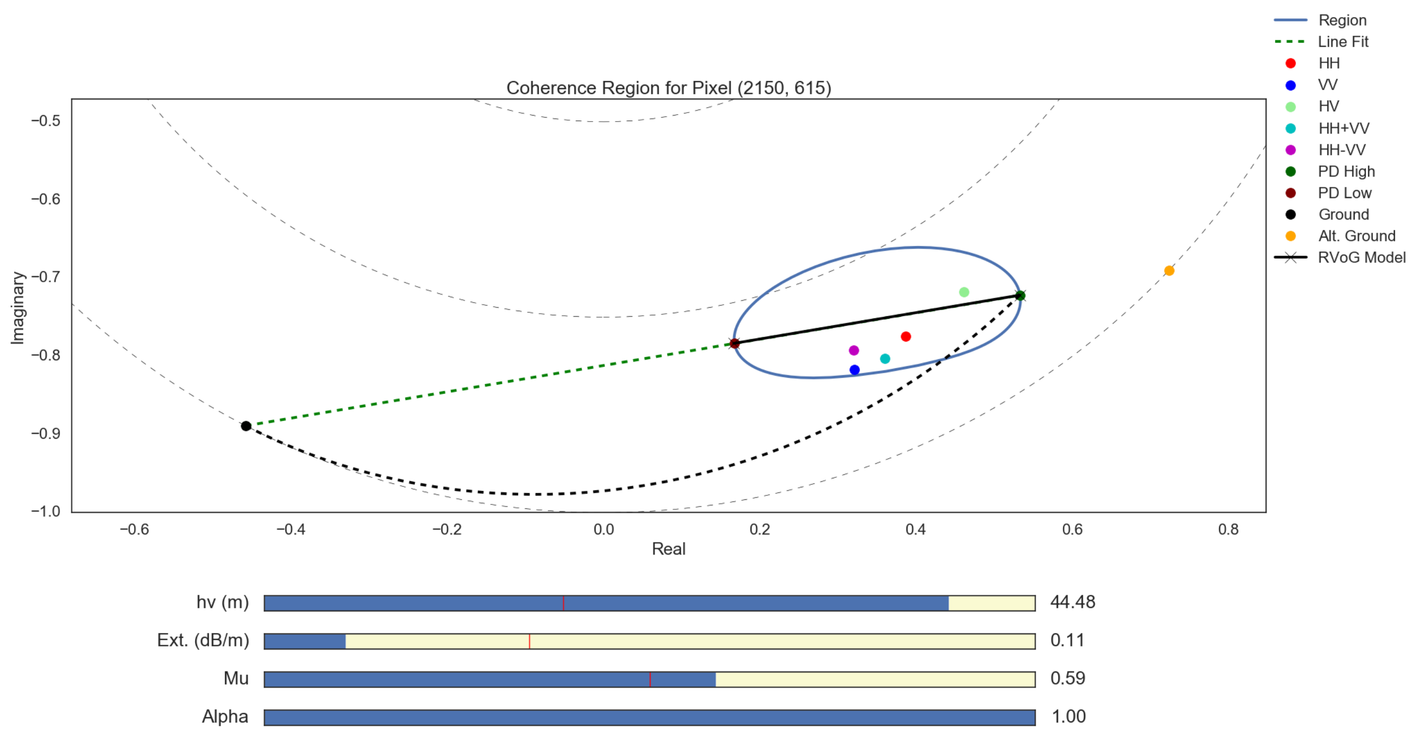

Figure 3.2: An example interactive coherence region plot.

and the line fit ground procedure, the reader may refer to [1]. For more on phase diversity coherence optimization, the reader may refer to [2].

While fitting the RVoG model parameters for a single pixel using the interactive plot can be helpful for illustrating and understanding the model, eventually we are going to want to derive a forest height map for the entire image. To start, we will perform the phase diversity coherence optimization procedure for the whole image using the Scene **opt** method:

```
scene.opt()
```

For large datasets, the coherence optimization procedure can take some time. It will print periodic progress updates to the console. Once it is finished, the optimized coherences will be saved as a dataset within the HDF5 file and can be accessed through the Scene object using the **coh** method.

```
high = scene.coh(pol='high')
low = scene.coh(pol='low')
```

The **coh** method returns the desired coherence as a NumPy array. Note that the **pol** keyword can also be set to 'HH', 'HV', 'HH-VV', etc., or to a three element array containing the complex weights for each polarization channel (in the order HH, HV, VV). These are also valid values for the **pol** keyword when using the **show** method:

```
scene.show('coh', pol='high')
scene.show('coh', pol='low')
```

Now that the coherence optimization is performed, let us proceed to forest height estimation. Model inversion is done using the Scene **inv** method. An example usage for the RVoG model inversion is:

```
rvog = scene.inv(method='rvog', name='rvog', desc='RVoG, hv and
    ext. free parameters, no temporal decorrelation.')
```

The first argument, **method**, specifies the forest height inversion method. The valid values for this argument are 'rvog', 'sinc' (for the sinc coherence model), or 'sincphase' (for the sinc and phase difference model [3]). Note that if no argument is specified, the RVoG model is the default. The next keyword, **name**, specifies a name for the dataset that will be created to contain the estimated model parameter values. By default, this is equal to **method**. The **desc** keyword lets us provide a string describing the model inversion which will be saved as an attribute to the HDF5 group containing the model inversion results.

Note that the HDF5 file is accessible through the Scene object directly, using **scene.f**. However, the Scene object also has a convenient **get** method for accessing HDF5 datasets contained within the file. Model inversion results are stored in the HDF5 file in the 'products' group. Each model inversion is stored as a group, and contains datasets for each of the optimized parameters. The forest heights from the 'rvog' model inversion and the individual optimized parameters can therefore be accessed as follows:

```
rvog_hv = scene.f['products/rvog/hv']
rvog_hv = scene.get('products/rvog/hv')
rvog_hv = scene.get('rvog/hv')

rvog_ext = scene.f['products/rvog/ext']
rvog_ext = scene.get('products/rvog/ext')
rvog_ext = scene.get('rvog/ext')
```

The three methods of getting the estimated forest height and extinction values are equivalent (the **get** method can either be given the full path to the desired dataset, or an abridged path, with the 'products/' group assumed). Note that the extinction values stored in the HDF5 file, and calculated by the RVoG model, are in units of Np/m, not dB/m.

Instead of the RVoG model, if we wished to invert the sinc model, we could do so using the following syntax:

```
sinc = scene.inv('sinc', name='sincmodeltest', desc='Sinc
    coherence model.')
```

We can display and save plots of the estimated forest heights for the two models as follows:

```
scene.show('products/rvog/hv', vmin=0, vmax=50, savefile='rvog_hv.
    png')
scene.show('products/sincmodeltest/hv', vmin=0, vmax=50, savefile=
    'sinc_hv.png')
```

If we've run a lot of model inversions with different names, it can be easy to lose track of them. For keeping track of the current contents of the HDF5 file, there is the Scene **query** method. If this method is called with no arguments, it will print a listing of all HDF5 groups and datasets within the file. If the name of a group or dataset is given as the argument to **query**, the attributes of that group or dataset will be printed. If the method is called with an empty string as an argument, the attributes of the main HDF5 file will be printed. In this way, all of the structure and metadata in the Scene object can be easily viewed. Here are some examples:

```
In [11]: scene.query()
kapok.Scene.query | Printing groups and datasets in HDF5 file...
kapok.Scene.query | cov
kapok.Scene.query | dem
kapok.Scene.query | inc
kapok.Scene.query | kz
kapok.Scene.query | lat
kapok.Scene.query | lon
kapok.Scene.query | pdopt
kapok.Scene.query | pdopt/coh
kapok.Scene.query | products
kapok.Scene.query | products/rvog
kapok.Scene.query | products/rvog/hv
kapok.Scene.query | products/rvog/ext

In [12]: scene.query('products/rvog/hv')
kapok.Scene.index | Printing attributes of 'products/rvog/hv'...
kapok.Scene.query | fixed: False
kapok.Scene.query | name: Forest Height
kapok.Scene.query | units: m

In [13]: scene.query('')
kapok.Scene.index | Printing attributes of main HDF5 file...
kapok.Scene.query | stack_name: pongar_27500_01
kapok.Scene.query | site: Pongara, Gabon
kapok.Scene.query | slc_azimuth_pixel_spacing: 0.6
kapok.Scene.query | slc_slant_range_pixel_spacing: 1.66551366
kapok.Scene.query | cov_azimuth_pixel_spacing: 12.0
kapok.Scene.query | cov_slant_range_pixel_spacing: 8.3275683
kapok.Scene.query | num_tracks: 6
kapok.Scene.query | num_baselines: 15
...
```

There are more attributes in the main HDF5 file, but the output shown here has been truncated for brevity.

Note that if we try to run **inv** using a value for the **name** argument which already exists, an error message will be printed and the method will abort. The software will avoid overwriting data unless we force it to. To do this, we can set the **overwrite** keyword:

```
rvog = scene.inv(method='rvog', name='rvog', desc='RVoG, hv and
    alpha free parameters, fixed extinction.', overwrite=True, ext
    =0.04, groundmag=0.95)
```

This time, instead of running the RVoG model using the default options, we have chosen to fix the extinction parameter to a constant value of 0.04 Np/m (approximately 0.35 dB/m). We have also set the magnitude of the ground coherence to 0.95 (default is 1.00). Reference on the various parameter values and options can be found using **help(kapok.Scene.inv)** and **help(kapok.rvog.rvoginv)**. By setting the overwrite keyword to True, the previous results stored under the 'rvog' group will be overwritten by the new model inversion.

In this chapter, we have focused on the use of the object-oriented Kapok interface. Note that all of the functions in the various Kapok modules can be imported and used manually, if greater control over the processing is desired. For example, let's redo the RVoG inversion using the individual functions at each processing step:

```
import kapok.topo
import kapok.rvog

ground, groundalt, volindex = kapok.topo.groundsolver(scene.pdcoh
    [:], kz=scene.kz[:], groundmag=0.95, returnall=True)
coh_high = np.where(volindex,scene.pdcoh[1],scene.pdcoh[0])
hv, ext, converged = kapok.rvog.rvoginv(coh_high, ground, scene.
    inc, scene.kz, ext=0.04, mu=0)
```

The **groundsolver** function finds the ground coherence. It also returns the alternate (unchosen) ground coherence, and a boolean array called volindex which tells us which of the two input coherences is farther from the ground (closer to the volume coherence, e.g., the high coherence). We then use volindex to create an array containing the high coherence image. We then run the RVoG inversion using the **rvoginv** function, giving it the high coherence, the ground coherence, the incidence angle, the k_z values, a fixed extinction, a μ value of zero (for the high coherence). The functions returns three arrays: hv, containing the h_v forest height values in meters; ext, containing the σ_x extinction values in Np/m; and converged, a boolean array specifying for each pixel whether the model was able to find a solution which closely fit the data (ideally, every pixel in converged will be set to True).

So far, the model inversions have been performed for every pixel in the PolInSAR data. In the case of water areas, the model will produce strange results. The low coherence magnitude of water areas is interpreted by the models as volumetric decorrelation. The estimated forest heights for the water areas will therefore tend to be very large. If we want to skip model inversion for these water areas, we can provide the **mask** keyword to the **inv** method. Mask should be a boolean array, with the same size as the scene. Where the mask is True, the inversion will be performed. The inversion will skip over pixels where mask is False, and the parameter values will be set to -1 for these pixels. This is particularly helpful for the RVoG model, where reducing the number of inverted pixels will also reduce the processing time.

Now we derive a mask from the HV backscattered power, excluding pixels with HV backscatter values below -22 dB:

```
import numpy as np

mask = scene.power('HV') # Get the HV backscattered power (in
    linear units).
mask[mask <= 0] = 1e-10 # Get rid of void data (set to -100 dB).
mask = (10*np.log10(mask)) > -22 # Values below -22 dB will not be
     inverted.
```

Note that the **power** method returns the backscattered power (in linear units) for the given polarization. Now we rerun the inversion using this mask:

```
rvog = scene.inv(method='rvog', name='rvog', desc='RVoG, hv and
    alpha free parameters, fixed extinction.', overwrite=True, ext
    =0.04, groundmag=0.95, mask=mask)
```

If we view these new results, we should see that the forest height values for the low backscatter areas should be set to -1. Finally, we would like to geocode our results. This is accomplished with the **geo** method:

```
scene.geo('rvog/hv', 'rvog.grd')
```

If we would like the masked out forest heights to be set to 0 instead of -1, we can multiply by the mask when geocoding:

```
scene.geo(scene.get('rvog/hv')*mask, 'rvog.grd')
```

Note that we can pass either a string (identifying the path to a HDF5 dataset in the file) or an array (containing the data to geocode) to the **geo** method. The **geo** method can also be used to create geocoded output maps for other inverted model parameters, or for the backscattered power, etc. When run, the above example will save the RVoG forest heights to an ENVI file called 'rvog.grd', in WGS84 Geographic (latitude, longitude) projection. The forest height map can then be opened in GIS software for further analysis.

Chapter 4

Data Organization

In this chapter, we will briefly discuss the internal data structure used to store the data inside the HDF5 file. The following is a list of the HDF5 datasets in the file after the initial data import process and the coherence optimization is performed:

- 'cov' (Covariance Matrix): [azimuth, range, row, column] (8-byte complex)
- 'kz' (k_z): [baseline, azimuth, range] (4-byte float)
- 'dem' (DEM): [azimuth, range] (4-byte float)
- 'inc' (Incidence Angle): [azimuth, range] (4-byte float)
- 'pdopt/coh' (Phase Diversity Optimized Coherences): [baseline, 2, azimuth, range] (8-byte complex)

The square brackets denote the dimensions of the datasets along each axis. The data types of each dataset are given in the parentheses.

The covariance matrix can be accessed directly through **Scene.f['cov']**, but can also be accessed through **Scene.cov**. There are similar shortcuts available for the other datasets in the list above. The optimized coherence dataset can be accessed using **Scene.pdcoh**. The text in square brackets are the dimensions of each of the datasets. Note that for 2D datasets, the azimuth index is the first dimension, and the range index is the second dimension. The covariance matrix has two additional dimensions, the row and column indices of the covariance matrix for each pixel. Note that in order to save disk space, any covariance matrix elements below the main diagonal of the matrix will be set to zero. Only elements above the main diagonal should be used. The Scene object methods automatically fill in the zero-valued elements with the appropriate values when necessary. If we wish to work with the covariance matrix directly, and require all the elements to be filled in with their actual values, we can calculate the zero-valued elements using the **kapok.lib.makehermitian** function.

For multi-baseline data, both the k_z and optimized coherences will have a baseline dimension. Baseline indices start at zero (for the first baseline), and increase from there. Baselines are numbered such that adding additional tracks will not change the numbering of existing baselines (see the header of the **kapok/lib/mb.py** file, containing functions which handle the baseline indexing, for more details). Note that for single-baseline data, k_z will have dimensions of [azimuth, range], while the phase diversity coherences will have dimensions of [2, azimuth, range]. The singleton dimensions are removed in the single-baseline case.

The 2 length dimension of the optimized coherences represents the two coherences calculated by the algorithm. The Scene **opt** method will attempt to sort the coherences such that the coherences located at [0] represent the high coherence, and the coherences in [1] represent the low coherence. But this may not always be correct depending on the desired ground solver parameters.

As discussed in the previous section, model inversion results are always stored in a 'products/' group within the HDF5 file, separated from the other datasets. Each inversion will be a subgroup within 'product', e.g., 'products/rvog' or 'products/sinc', etc. Within these subgroups, there will be one dataset for each solved parameter, e.g., 'products/rvog/hv' or 'products/rvog/ext' or 'products/sinc/hv', etc. All of the parameters are stored as 2D datasets with the same [azimuth, range] dimensions as the rest of the data.

Chapter 5

Module Reference

What follows is a list of the Kapok modules, as well as a brief description of each of their functions. For more details on a particular function, the function headers can be viewed using Python's built-in help system, using **help(kapok.module.function)**.

5.1 kapok (Core Functionality)

This module contains the Scene class definition, which has the following methods:

- get — Get a requested HDF5 dataset.
- query — Get a list of groups and datasets in the HDF5 file, and read the attribute values.
- inv — Model inversion and parameter estimation.
- opt — Coherence optimization.
- show — Display images.
- power — Return the backscattered power for a given polarization.
- coh — Return the complex coherence for a given polarization.
- region — Coherence region plotting.
- geo — Output geocoded data for further analysis.
- ingest — Load external ENVI raster data (in WGS84 geographic coordinates), reproject them into the UAVSAR radar coordinates, and then save them into the Kapok Scene HDF5 file.

Help for the entirety of the Scene object can be accessed using **help(kapok.Scene)**.

5.2 uavsar (UAVSAR Data Import)

This module handles the loading of UAVSAR data, and has the following functions:

- load — Load in the specified UAVSAR data stack, and import/calculate the covariance matrix, incidence angle, k_z, and other data.
- Ann — This is a simple class which allows loading and querying of UAVSAR annotation files.

5.3 vis (Data Visualization)

This module handles display of images for a variety of image types, and has the following functions:

- show_linear — Display linear data.
- show_power — Convert power data to dB and display.
- show_complex — Show the magnitude and phase of complex data using the HSV colormap.
- show_paulirgb — Show a Pauli basis RGB color composite image.

5.4 cohopt (Coherence Optimization)

This module handles coherence optimization, and has the following functions:

- pdopt — Perform phase diversity coherence optimization over an entire PolInSAR image.
- pdopt_pixel — Perform phase diversity coherence optimization for a single pixel. This function is called when plotting coherence regions.

5.5 region (Coherence Region Plotting)

This module plots coherence regions, and creates the interactive coherence region to test the model parameters. It has the following functions:

- cohregion — Plot the coherence region for a chosen pixel.
- rvogregion — Plot an interactive coherence region with sliders for the RVoG model parameters.

5.6 topo (Topography Estimation)

This module handles estimation of the topographic InSAR phase, and has the following functions:

- groundsolver — Estimate the ground coherence from two optimized coherence observations.
- linefit — Calculates the two possible ground coherence intersections between the line and a circle with radius equal to the ground coherence magnitude.

5.7 sinc (Sinc Forest Model)

This module contains the sinc coherence forest model, and the sinc and phase difference models, basic methods for forest height estimation [3]. It has the following functions:

- sincinv — Calculate forest height using the sinc coherence model.
- sincfwd — Sinc coherence forward model.
- sincphaseinv — Calculate forest height using the sinc and phase difference model.

5.8 rvog (Random Volume over Ground Forest Model)

This module handles inversion of the random volume over ground forest model [1], and has the following functions:

- rvogfwdvol — Calculate the RVoG forward model volume coherence.
- rvoginv — Invert the RVoG model, estimating the forest height and either the extinction parameter or the temporal decorrelation, depending on the input options.

5.9 geo (Geocoding)

This module handles geocoding of output data products, and has the function:

- radar2ll — Resample an array in radar (azimuth, slant range) coordinates to Geographic coordinates using gdalwarp.

Bibliography

[1] S. Cloude and K. Papathanassiou, “Three-stage inversion process for polarimetric SAR interferometry,” *Radar, Sonar and Navigation, IEE Proceedings* -, vol. 150, no. 3, pp. 125–134, June 2003.

[2] M. Tabb, J. Orrey, T. Flynn, and R. Carande, “Phase diversity: A decomposition for vegetation parameter estimation using polarimetric SAR interferometry,” in *Proceedings of EUSAR 2002*, 2002, pp. 721–724.

[3] S. R. Cloude, “Polarization coherence tomography,” *Radio Science*, vol. 41, no. 4, 2006. [Online]. Available: http://dx.doi.org/10.1029/2005RS003436