ISCE Tutorial

Paul Rosen, Eric Gurrola, Piyush Agram, Marco Lavalle, Mark Powell

NASA Jet Propulsion Laboratory, California Institute of Technology

November 23, 2015

Contents

CHAPTER 0

License

CHAPTER 1

Introduction

The following chapters are based on a series of cloud-based tutorials for using the Interferometric Synthetic Aperture Radar (InSAR) Scientific Computing Environment (ISCE) to process InSAR data from several different international sensors. The cloud-based tutorials were presented to students at a workshop at UNAVCO in Boulder Colorado in August, 2014 and at a UAVSAR workshop presented at the USGS in Reston Virginia in October, 2014. Each student was given an account that they accessed through a web browser on their own laptop computer. The browser display presented two side-by-side panes to them. On the left pane they read instructions. On the right pane they were presented with a computer terminal emulation that allowed them to enter commands at a command line prompt. Visualization of data was done with a remote desktop application called Guacamole (you will see it referred to in some of the chapters). The browser interface to the tutorial materials and the integrated interface to the cloud virtual machines was developed at the Jet Propulsion Laboratory under a NASA contract. Through these interfaces the students were able to work directly on a pre-configured virtual machine (VM) on the Amazon cloud. The pre-configured VMs were configured with the required software and each had a data volume attached containing the sample raw data sets used in the tutorials. At start up, in other words, the student was ready to begin learning how to use ISCE to process data without having to install software and download data.

The main purpose of the Earthkit tutorials and this document is to teach the student how to work with the software package named ISCE (InSAR Scientific Computing Environment) developed at the NASA Jet Propulsion Laboratory, California Institute of Technology. The ISCE package provides software basic to processing Level0 or Level1 SAR data into interferometric products with options for filtering, unwrapping, and geocoding. Most of the chapters here are on ISCE. A couple of the tutorials, however, (Chapters 8 and 12) show how to use interferometric data created by ISCE as inputs to the Generic InSAR Toolbox (GIAnT-developed by Piyush Agram at the California Institute of Technology) for performing geodetic time series analyses. One tutorial (Chapter 10) describes how to use the PolSARPro package developed at the European Space Agency (ESA) to use InSAR data (such as UAVSAR polarimetric data) to analyze land surface changes.

Earthkit is not available for general use except during the workshops, but the tutorial material may be useful to people first learning how to process InSAR data with ISCE; therefore, we are making the contents of the Earthkit "left pane" (the step by step instructions) available in this document. What is missing here is the pre-installed software and the data that goes with these tutorials. In future versions of this document we will add a chapter on installing the software. For now, there is information in the README.txt file that comes with ISCE on how to install ISCE and its dependencies. There is also a user forum that may be helpful at the website, `http://earthdef.caltech.edu/projects/isce_forum/boards`.

The student may not be able to obtain the exact data referred to in these tutorials, but that should not prevent the student from deriving benefit from the material. If a student is interested in learning how to use ISCE, then they probably already have some data they need to process. There is a high probability that the data type that the student wants to work with is covered in these tutorials. If not, then that data type may still be available in the version of ISCE that the student has (but may have been neglected to be included in the tutorials). If the student can not find information on working with the data type they have, then they might consider asking about it at the user forum mentioned in the previous paragraph.

These tutorials present concepts for using ISCE in a progressive way using data types from different international sensors. Many of the concepts are common to usage of ISCE for data from any sensor. The specific input files described in a given tutorial, however, are usually specific to each data from a particular sensor. Therefore, the student new to ISCE and to processing InSAR data should read through the chapters to get the concepts even if those chapters are discussing

data from different sensors than the one of interest to the student. If possible the student may want to obtain data from the sensor being discussed in a particular tutorial. Alternatively, they should be able to find example ISCE input files for their sensor in the "examples" directory of the ISCE package, and most of what they read in a given tutorial will still make sense to them while working with a different sensor and data type. Also, the tutorials refer to specific directory paths that were created on the cloud instance. The user will have to create their own data paths on their computer and translate between the paths they read in the tutorial and the paths they create on their computer.

This initial version of this tutorial document was created by "pasting" in the Earthkit tutorial pages and adding chapter delimiters between them. The section headings in this version are based on the original tutorial pages and have not been renumbered in a logical way within each chapter. Also, there are dead links that were used to jump to other pages within the tutorial on the cloud version. There are very few of these and with a little patience the context of the words around the link will help the student to find the page, usually within a few pages in the same chapter. Future versions of this document will re-typeset this document in a uniform fashion and will fix these links. We think that it is more useful to put out this document as it is now and fix these minor idiosyncrasies in a future release.

CHAPTER 2

Getting Started With ISCE

1. Introduction to basic ISCE

Welcome to your first session using ISCE. The purpose of this session is to demonstrate the most straightforward application of ISCE for geodetic imaging, the insarApp.py application. insarApp takes two raw data files and optionally a digital elevation model (DEM) and produces a geocoded, and optionally unwrapped, interferogram. insarApp runs automatically from start to finish with no required interaction. The input files have a number of configurable parameters that allow the user to control how the processing is done. For this first session, we have preconfigured all the input parameters in a set of input files.

So let's get started and simply explore what and where the data files are, then run insarApp.

2. Where are we?

Let's see what the files look like on the disk. First let's see where we are. The Unix "pwd" command tells us the "present working directory" in the file system.

```
> pwd
/home/ubuntu/
```

"ubuntu" is your predefined username on this cloud instance virtual machine. (Ubuntu happens to be the name of a version of the linux operating system that is running on this virtual machine.) We have chosen a single username for every instance so everyone sees the same interface.

Now let's take a look at what we've got. The unix "ls" command lists files in the present working directory

```
> cd data
> ls
giant      lab1  lab4      orbits                      sites
instruments  lab3  lost+found  raw_from_roi_pac_tutorial
```

This shows that within the present working directory are a number of other directories that contain the information and descriptive content for each of the ISCE labs. Let's change our working directory to lab1 so we can get started with the first lab:

```
> cd lab1
> pwd
/home/ubuntu/data/lab1
```

3. Setting up to run ISCE.

Now let's see what is in this directory using the "ls -l" command to display more ("-l" = "longer") information about the files:

```
> ls -l
drwxrwxr-x 2 ubuntu ubuntu 4096 Apr 18 17:33 20061231
drwxrwxr-x 2 ubuntu ubuntu 4096 Apr 18 17:34 20070215
drwxrwxr-x 3 ubuntu ubuntu 4096 Oct 17 05:48 20070215_20061231
```

Note there are three subdirectories within the lab1 directory: 20061231 contains the raw data for one radar observation, 20070215 contains the raw data for the other radar observation, and 20070215_20061231 is a directory we have created that will be used to store the output of the insarApp run. This convention of naming data directories by the date is common in the InSAR community. You probably know the dates of your acquisitions from when you decided what data to process. We will describe later how to view the metadata to determine the date of any data set. We can see what is in these subdirectories while staying at this level by issuing the following command:

```
> ls 20061231
IMG-HH-ALPSRP049770670-H1.0__A  LED-ALPSRP049770670-H1.0__A
> ls 20070215
IMG-HH-ALPSRP056480670-H1.0__A  LED-ALPSRP056480670-H1.0__A
> ls 20070215_20061231
insar_20070215_20061231.xml  Master.xml  Slave.xml
```

The data set we have provided for this session is a pair of ALOS PALSAR observations acquired over Los Angeles. We have named the directories containing these data by using the date of acquisition, but the user is free to name these directories arbitrarily (within unix requirements). In this example, we have provided an appropriate DEM that will be applied in the processing. In a later step, we will show how to tell ISCE to find an appropriate DEM, and configure parameters.

To run insarApp, we need to set the present working directory to the directory where we want the output to be. In addition, the input configuration XML files must be in this same working directory. From above, we see that the xml file is in 20070215_20061231, so let's change to that directory:

```
> pwd
> cd 20070215_20061231
> pwd
/home/ubuntu/data/lab1/20070215_20061231
```

```
> ls
insar_20070215_20061231.xml  Master.xml  Slave.xml
```

We will come back later to examine the contents of the xml file, but let’s jump in the deep end and process some data from start to finish.

4. Running ISCE and examining the output

Now that we are at the appropriate working directory to run the application, to process the data we simply issue the following command:

```
> insarApp.py insar_20070215_20061231.xml
```

At this point, the screen should fill with scrolling text describing where you are in the processing. For the scenes provided, the total processing time should take roughly 20 minutes. Go get some coffee and come back then.

Now that insarApp has finished, we can examine the output, and try to make sense of some of the ancillary data that ISCE computes. First, let's see what we've got:

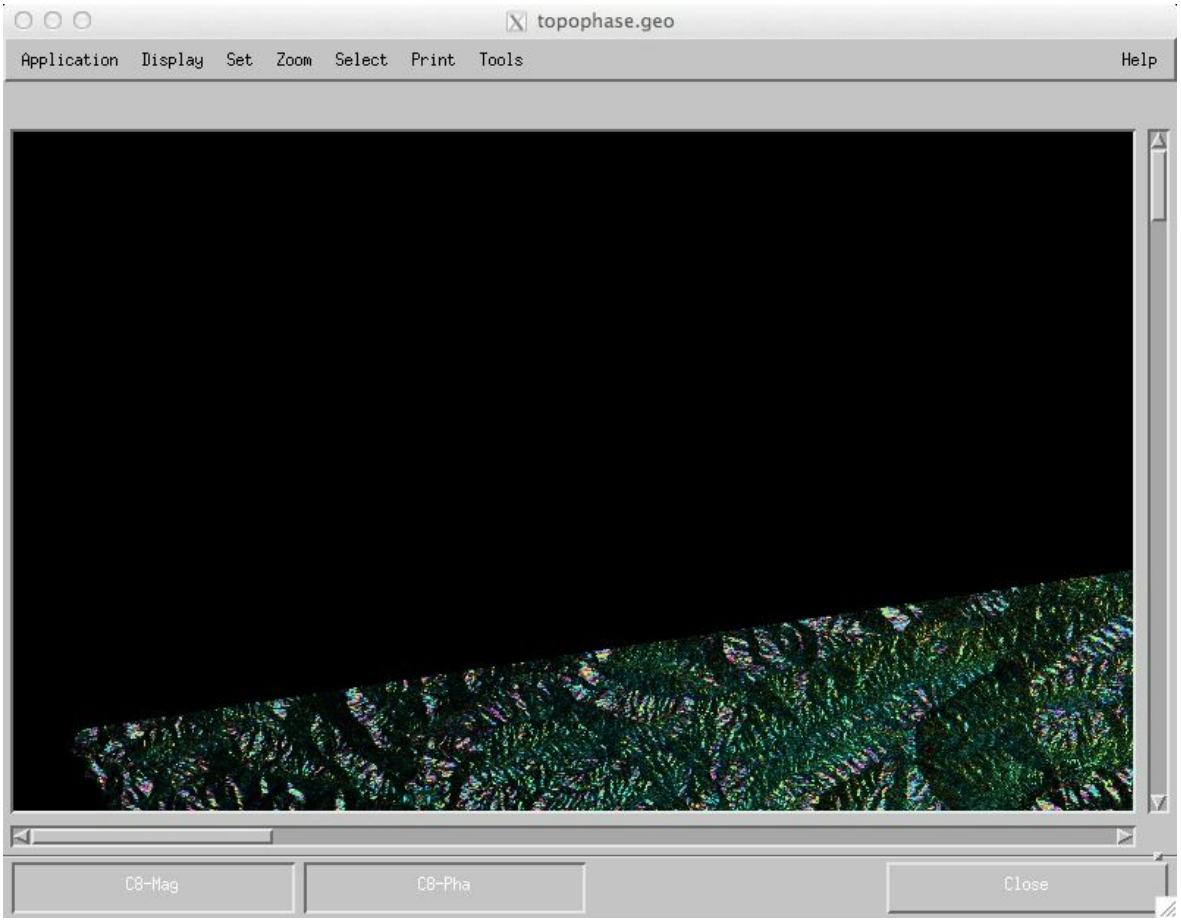
```
> ls
20061231.raw                                iz
20061231.raw.aux                            lat
20061231.raw.xml                            lon
20061231.slc                                Master.xml
20061231.slc.xml                            rangeOffset.mht
20070215.raw                                rangeOffset.mht.xml
20070215.raw.aux                            resampImage.amp
20070215.raw.xml                            resampImage.amp.xml
20070215.slc                                resampImage.int
20070215.slc.xml                            resampImage.int.xml
azimuthOffset.mht                           resampOnlyImage.amp
azimuthOffset.mht.xml                       resampOnlyImage.int
catalog                                     resampOnlyImage.int.xml
dem.crop                                    rgdem
demLat_N33_N35_Lon_W119_W116.dem            Slave.xml
demLat_N33_N35_Lon_W119_W116.dem.wgs84      topophase.cor
demLat_N33_N35_Lon_W119_W116.dem.wgs84.xml  topophase.cor.xml
demLat_N33_N35_Lon_W119_W116.dem.xml        topophase.flat
filt_topophase.cor                          topophase.flat.xml
filt_topophase.flat                         topophase.geo
filt_topophase.flat.xml                     topophase.geo.xml
filt_topophase.unw                          topophase.mph
insar_20070215_20061231.xml                 topophase.mph.xml
insar.log                                   z
insarProc.xml                               zsch
isce.log
```

In later labs we will examine the most important files in this directory in detail, but for now, let's do the most gratifying thing and just look at the final output, which is called

“filt_topophase.flat.geo.” To display InSAR data, we have developed a tool called mdx that is especially well tailored to examining large files with complex number data types such as interferograms. We have an application that looks at the metadata associated with a data file and calling mdx appropriately.

```
> mdx.py filt_topophase.flat.geo
```

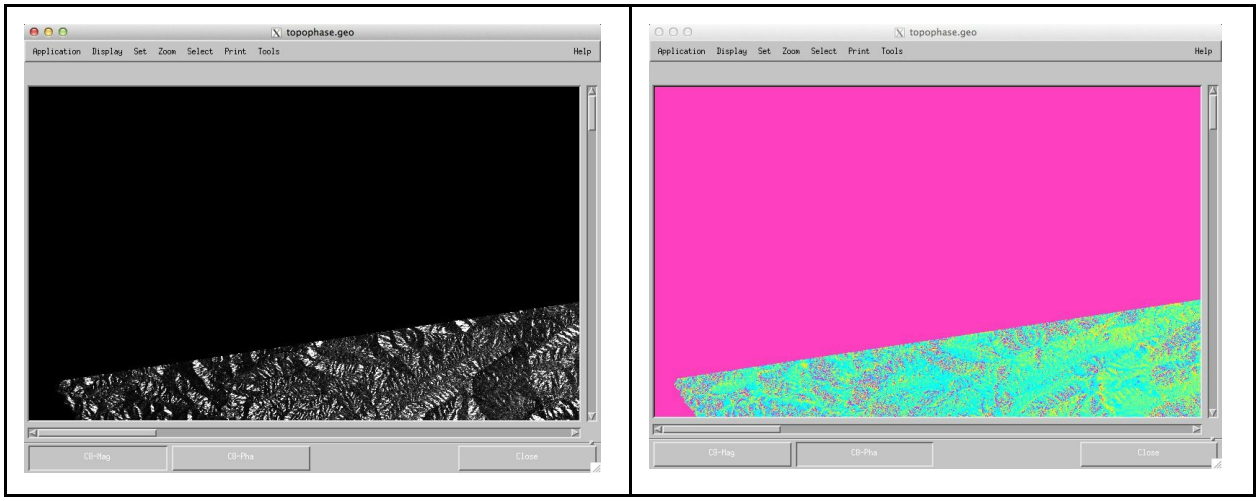
We now have on our screen a new window displaying a portion of the image file.



The data file filt_`topophase.flat.geo` contains a wrapped interferogram that has been flattened and geocoded using a dem (digital elevation model) that was downloaded by ISCE early in the processing. Each pixel in the image is a complex number with an amplitude and phase associated with it. The displayed image has a color mapping that blends the amplitude of the image with the phase which aids the viewer in interpreting the phase values in the context of the local area (e.g. water is always decorrelated in differential InSAR images, so the phase will be random. observing a water body in the amplitude data will assure the viewer that all is well.)

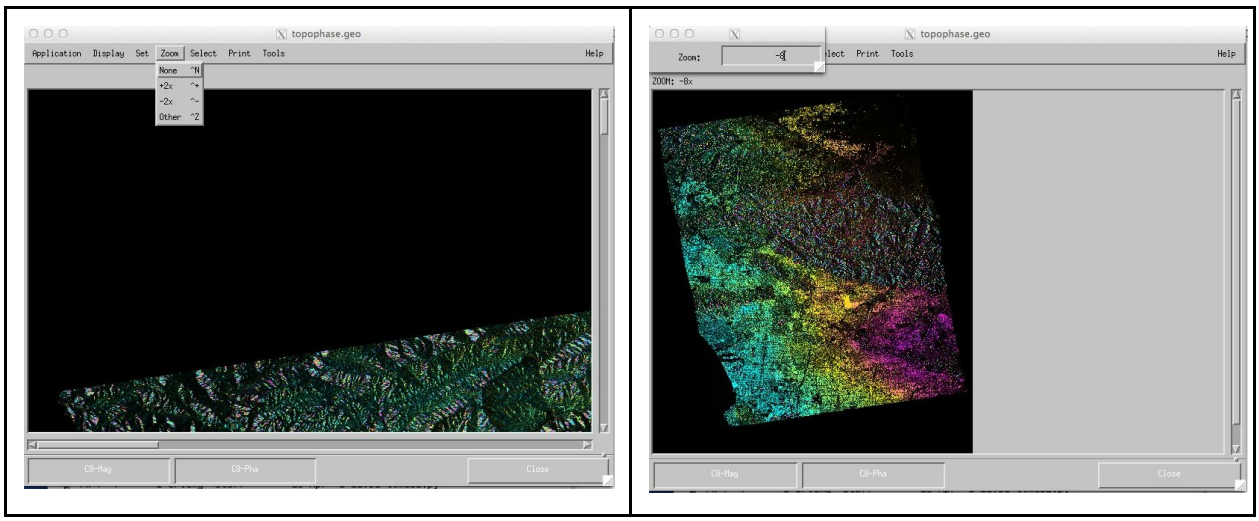
5. Basic MDX controls

The user can toggle the display to see only amplitude by clicking on the “C8-mag” button. To see only phase, click on “C8-pha”. To go back to seeing both, center click again on “C8-mag”.



mdx has a rich set of display features that can be explored at leisure. We should look at one of the most important features though, which is “zoom”. We see that the image as displayed is only a portion of the full image, displayed at a 1:1 zoom scale. We can scroll around the image by clicking the scroll bars, but to get the big picture, we can select the zoom menu item to zoom out.

The zoom menu presents us with several options. “None” means revert to 1:1 scaling. “+2” means zoom in by a factor of 2. “-2” means zoom out by a factor of 2. “Other” allows us to set the zoom factor as we like. In the images below, we selected a factor of -8 which shows us the entire file. Due to the aspect ratio of the window, extra area is displayed as gray. We can resize the windows to make better use of the screen space.



This concludes your first successful ISCE run!

1. Number formats in ISCE data files

ISCE files are almost universally either binary “flat files” containing image data or large arrays of processing relevant information, or ASCII text files containing logging, metadata, and other ancillary information. A “flat file” is just an unformatted sequence of binary numbers that can represent one-, two-, or n-dimensional arrays of data. The binary numbers themselves can be single bytes integers, two-byte integers, four-byte integers, or four-byte floating point numbers, depending on the data file and its application.

To further complicate matters, the numbers can be complex, that is with a real and imaginary part. Radars are coherent instruments, essentially operating at a single radio sinusoidal frequency. As such, the instrument can keep track of the magnitude and the phase of the sinusoidal signal in the received data. It is this phase information that allows us to perform interferometry - interfering the phase of one image against another. (For those rusty on complex numbers: the magnitude of a complex number is calculated from $(R*R+I*I)^{1/2}$ and the phase from $TAN^{-1}(I/R)$).

Complex numbers are typically represented in a file as “sample-interleaved” data. For example, for a 3 x 3 image, if `R` represents the real part of a sample, and `I` represents the imaginary part of a sample, then the image file would look like:

Row 1: `RIRIRI`
Row 2: `RIRIRI`
Row 3: `RIRIRI`

and represented as just a sequential unformatted list of numbers it would look like:

`RIRIRIRIRIRIRIRIRI`

Note that R and I represent different values at each sample position. Note also that the number format for the samples can be a range of possibilities. In ISCE, generally, `R` and `I` are 4-byte float numbers, such that the complex sample is an 8-byte complex quantity. But there is no reason that complex numbers could not be for instance represented as having a 2-byte real and 2-byte imaginary part.

Another aspect of these flat files is that they may contain more than one layer of data. For example typically, ISCE binds a radar image and a derived interferometric image into one file to make it convenient to match an easily recognizable geographic feature with an interferometric quantity. These files are typically line-interleaved. A 3 x 3 line interleaved example would be:

Row 1: `AAA`
Row 2 `PPP`

Row 3: `AAA`
Row 4 `PPP`
Row 5: `AAA`
Row 6: `PPP`

where again A and P represent different values at each sample. This can also be thought of as a double width file of three rows.

Row 1: `AAAPPP`
Row 2: `AAAPPP`
Row 3: `AAAPPP`

ISCE does have some file naming conventions that would aid a user in knowing what format is specified, but it is best not to rely on that. The Output File Formats discussion in the Datasets tutorial has a complete description of the naming conventions used in ISCE for flat files.

Clearly, without a metadata file describing the dimensionality and other attributes of the data file, a user would need separate documentation to interpret it. ISCE metadata files carry all the necessary attributes of the data.

CHAPTER 3

Using MDX

1. MDX interferogram visualization

In a previous lab we saw a preliminary use of the display tool mdx. In this segment, we will explore the features of mdx in greater detail to highlight its utility.

mdx stands for “multi-dataset display using X11” and was written as a special purpose display tool to visualize multiple large data sets associated with radar data processing. mdx takes into account the fact that the data sets are typically very large, often multiple gigabytes, and therefore impractical to read into memory all at once. Therefore, in its design, it keeps track of the coordinates of the display that the user controls with the GUI motif-based user interface, and only reads, scales, maps to color, and displays those data within the window. When the user scrolls the window to another location, mdx keeps track of what is old and what is new in the new position and recalculates and redisplays only the new data. In this way it can be very fast to explore very large data sets. On zooming out, mdx subsamples the data file, so even zooming out over large areas allows for relatively fast panning. This feature, which one would expect to find in many visualization tools, is unique to mdx, and makes it a very powerful tool for working with ISCE data.

2. Command line options

mdx has a large array of command-line arguments to control the data sets that are displayed and how they are displayed. Typing “mdx” at the command prompt with no command arguments returns a help message with a usage message and all the possible options. We will describe the most often used of these options later, but first we will describe a python script called “mdx.py” that allows users quick and easy use of mdx for commonly displayed data types. Typing “mdx.py” at the command prompt with no command arguments returns a help message describing its usage in full.

```
> mdx.py
  Usage:

  mdx.py   filename [-wrap wrap] ... [-z zoom -kml output.kml]

  where

  mdx.py  : displays one or more data files simultaneously by
            specifying their names as input. The maximum number,
            of images that can be displayed depends on the machine
            architecture and  mdx limits. If displayed (no -kml flag)
            the images don't need to have the same extension, but need
            to have same width.

  filename: input file containing the image metadata.
            Metadata files must be of format filename.{xml,rsc}
            and must be  present in the same directory as filename.
            Different formats (xml,rsc) can be mixed.

  -wrap   : sets display scaling to wrap mode with a modules of Pi.
            It must follow the filename to which the wrap is applied.

  ...  : the command can be repeated for different images.

  -z   : zoom factor (+ or -) to apply to all layers. It's optional
            and can appear anywhere in the command sequence and must
            appear only once.

  -kml : only for geocoded images it creates a kml file with all the
            input images overlaid. Each layer can be turn on or off in
            Google Earth. It's optional and can appear anywhere in the
            command sequence and must appear only once. The images don't
            need to be co-registered.

  Examples:
  mdx.py 01_02.int                              # Standard way to run mdx.py
  mdx.py 03_04.int 05_06.int -z -8              # Display two images; zoom out by 8
```

```
  mdx.py 03_04.geo -z 8 05_06.geo -kml fileout.kml # Create a kml file named
fileout.kml with two
                                                   # layers, one per image. Both
images are zoomed in
                                                   # by a factor of 8
  mdx.py 03_04.int 05_06.int -wrap 6.28            # Display two images. Wrap the
second modulo 2Pi


  or

  mdx.py -ext

  to see the supported image extensions.
```

3. MDX and file names

The common form of a data filename in ISCE is *prefix.ext* where *ext* is one of a number of standard extensions that describe a particular format or type of data. See Output File Formats discussion in the Datasets tutorial for a complete description of the extensions used in ISCE. Typing “mdx.py -ext” at the command prompt with no other command arguments displays a list of all the extensions that mdx.py understands, including a few heritage extensions not used by ISCE.

```
> mdx.py -ext
version =  1.0.0
Supported extensions:
unw
byt
flg
scor
slc
int
geo
flat
mph
cpx
msk
cor
hgt
hgt_holes
rect
amp
dem
dte
dtm
```

mdx.py takes as arguments a list of filenames, one for each of the images you want to display. Since all ISCE data files are flat files, ie. binary files of data arranged in rows and columns with no headers or other descriptive information, their corresponding xml metadata file is needed to describe the image dimensions. The metadata filename is formed in ISCE by adding “.xml” to the end of the filename. The format of the data is also encoded in the metadata. mdx.py constructs a metadata filename from the data filename provided on the command line, then reads all the descriptive information from the metadata file to determine how to display the data.

4. Layers of data

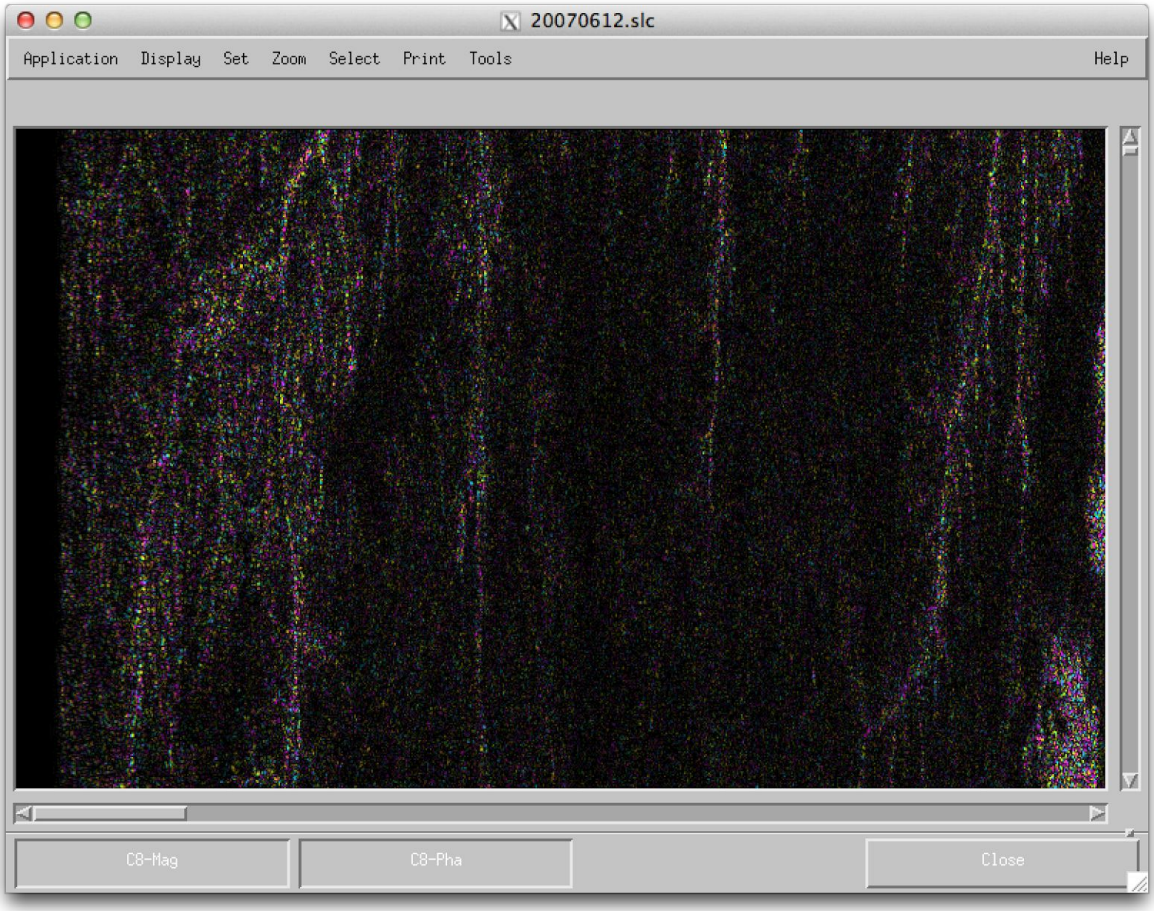
Let's look at a few examples. The output of ISCE after running the insarApp.py application contains a pair of complex images that form the interferogram, typically known as the "master" and "slave," though the name of the file can be arbitrary. In what follows, the master and slave filenames are denoted by the dates of acquisition. You might want to compare these images to see how well they align after motion compensation and image formation.

```
> cd /home/ubuntu/data/sites/lab3_alos/20070612_20090802
```

This site is known as the Afar region in Africa. A599 refers to the Ascending track number and 0230 refers to the frame number. The dates 20070612 and 20090802 are the dates on which acquisitions of this area were made, and we choose to create a directory called `20070612_20090802` as the location to process the data. The input data files are contained in directories `20070612` and `20090802` individually. Let's display one of the processed images: the "single look complex" slc files.

```
> mdx.py 20070612.slc
```

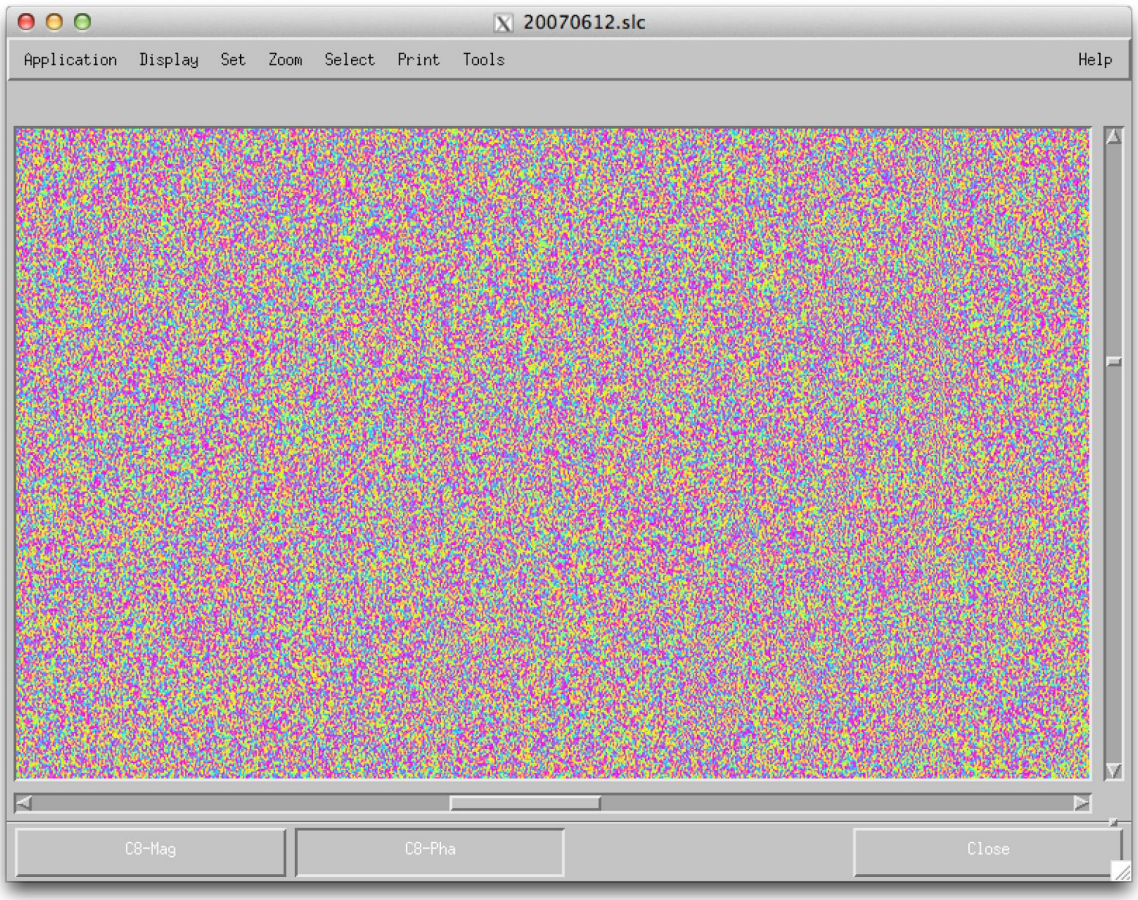
will display this file as 2 layers, each layer with a clickable button to select it.



Note the name associated with these buttons. An `slc` is a complex data type, and the natural way to display each complex pixel is through its magnitude and phase. So for this slc, mdx.py will construct a layer that is the magnitude (`C8-Mag`) and another layer that is the phase (`C8-Pha`). The `C8` designation means that the slc file is stored as a complex 8-byte data type (4-byte real, 4-byte imaginary; mdx understands complex integers as well). For more on complex numbers and data formats, see this document on Number Formats.

Since we specified one slc file as input to mdx.py, there are 2 layers, C8-Mag, C8-Pha. By clicking on either layer, it will select just that layer. Click back and forth on the C8-amp and C8-Pha layers to see that while there is structure in the amp layer associated with the ground reflectivity, the phase layer is completely random. This is because the radar is a coherent system and the phase is representative of the “speckle” pattern created by random collections of scatterers in a resolution element. The phase is a completely random process. The amplitude is also subject to speckle, which is why it is so noisy, but the intrinsic reflectivity of the surface shows through.

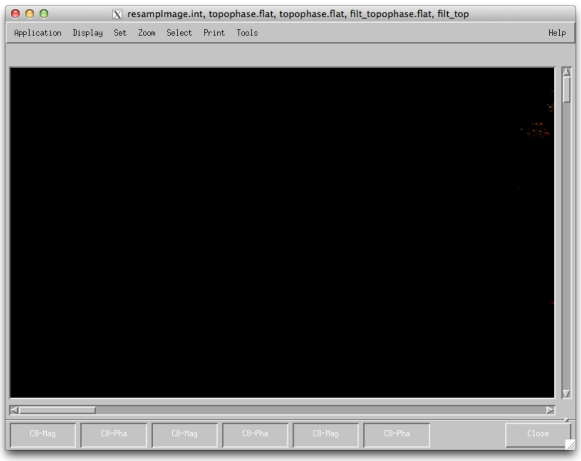
Click on the C8-Pha layer and scroll around using the scroll bars. Try to match the scroll locations as shown below. The phase should still look random!

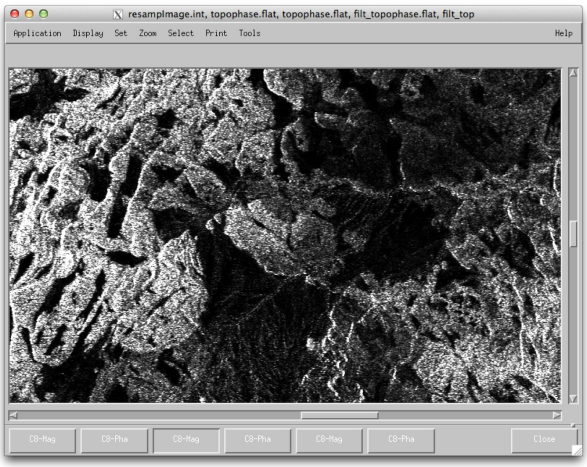


It is hard to imagine looking at this random phase that there is information present in the data. But because the randomness has to do with the spatial variability of the scattering on the ground, if another acquisition is made at a different time from nearly the same vantage point, then the randomness will be essentially the same, so in the difference, i.e. in the interferogram, this randomness cancels and a beautiful interference pattern emerges. Let's examine the various interferograms that ISCE insarApp.py produces.

```
> mdx.py resampImage.int topophase.flat filt_topophase.flat
```

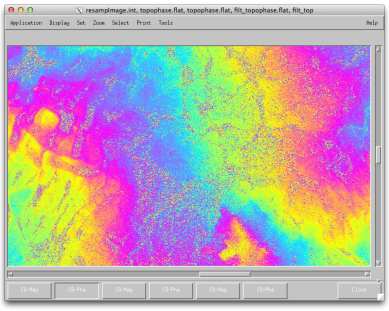
will show the original interferogram, one flattened to remove topography, and another filtered to smooth out the noise. These are also complex data types, so there will be 6 layers - 3 pairs of C8-Mag, C8-Pha layers. Note that the display comes up looking very dark. This is because this corner of the images is indeed low reflectivity water. Also each of the three amplitude layers is multiplied together to merge them for display, so they become even darker.



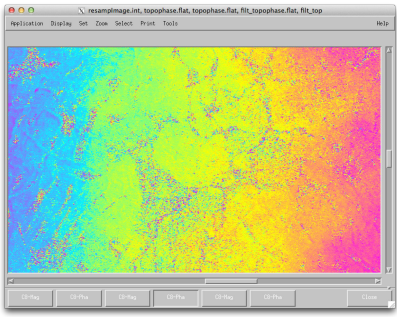


If you click individually on any one amp layer, you will see they are brighter, and if you scroll around to different areas, you'll see much brighter surface features in the image, as shown in the side-by-side comparison above.

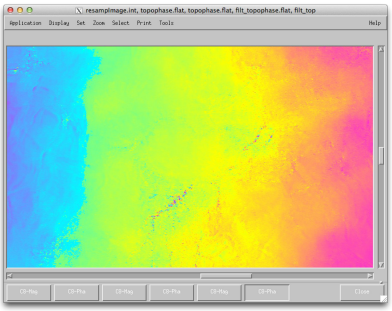
Now click on the three C8-Pha layers to see how they change with different stages of processing. Note that the random phase of the slc file has yielded to a phase difference pattern than is quite well behaved and non-random, through the magic of interferometry.



`resampImage.int - basic interferogram`



`topophase.flat - interferogram with topography removed`

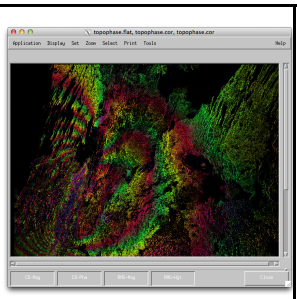
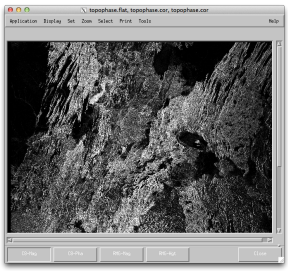
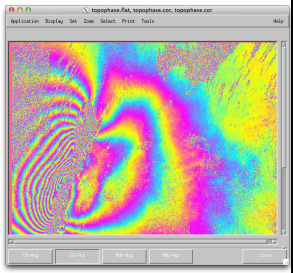
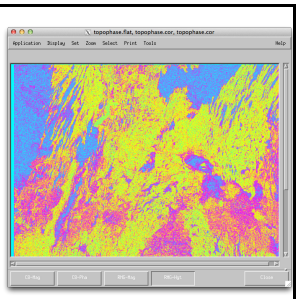


`filt_topophase.flat - filtered version to smooth out the noise`

Note that if you would like to go back to an overlay of data, for example overlaying the amp and pha layers of a particular image, you must click on one layer then "middle click" (for a 3-button mouse) on the other layer. ("middle" click on a 1-button mouse depends on your system; for a mac, often setting the X11 preference to "emulate three button mouse" and holding down the "Option key" while clicking will emulate a middle-click.)

You can also mix data types. You can examine the interferograms and the correlation file to compare the phase noise to the correlation.

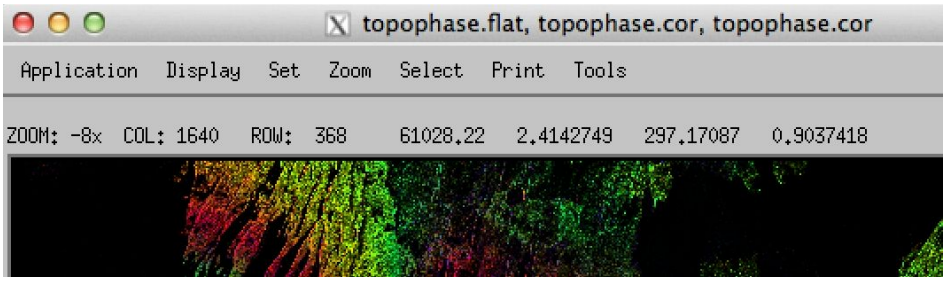
```
> mdx.py -z -8 topophase.flat topophase.cor -wrap 1.2
```

			
merged layers of above command for topophase.flat and topophase.cor	amplitude layer only of topophase.flat	phase layer only of topophase.flat	correlation layer only of topophase.cor, with a color cycle wrap of 0 - 1.2

In this case, mdx.py displays the `C8-Mag` and `C8-Pha` layers of the topophase.flat followed by the `RMG-Mag` displaying again the magnitude kept in the correlation file layer and `RMG-Hgt` displaying the correlation layer. An `RMG` file is a ROI_PAC legacy line-interleaved format file still used in ISCE. Click back and forth on the magnitude layers. They should be completely coregistered. Then click back and forth on the phase and correlation layers. Note the correlation is high when the phase is smoother. The figure above illustrates the various layers. Note also in this example that we specified a zoom factor of 1/8 (“zoom out”) on the command line “`-z -8`”. As shown in the first tutorial, zooming can also be accomplished with the mdx windows.

5. Bonus features

A couple of other features of mdx and mdx.py that you should know about:
1) by clicking on any pixel in the image, the values of all layers at that pixel will be displayed at the top of the display window, above the image but below the menu bars.
2) the menu bars allow the user to interactively control a number of features, including the zoom factor



mdx.py is a convenient way to quickly display files that are produced by ISCE. But the mdx program itself has a considerable amount of command-line flexibility that mdx.py does not try to capture. You can see a glimpse of the syntax expected by mdx by observing the command line for mdx that mdx.py constructs:

```
> mdx.py topophase.flat
Running: mdx topophase.flat -c8 -s 5194
.
.
.
```

By playing with mdx.py and observing the constructed mdx commands, and by reading the usage message for mdx (two pages!), you can get a feel for how to display files with many different characteristics simultaneously and efficiently using the mdx executable command (as opposed to the mdx.py script) directly. But mdx.py should satisfy the needs of most users.

CHAPTER 4

Processing Interferometric Data Sets Using insarApp.py

Understanding what data sets are possible to use with ISCE

ISCE is a package designed to work with data from most of the available international SAR sensors operating in a standard “stripmap” mode. In stripmap imaging, the sensor generates a regular stream of radar pulses and the pointing of the radar beam is fixed to be roughly broadside to the direction of flight of the spacecraft. In this way a continuous swath is acquired for as long as the radar sensor is on. The ISCE package understands the formats of the following sensors:

Satellite	Years of Operation	Repeat cycle (days)	Wavelength (band/cm)	Stripmap Modes	Product level that can be ingested into ISCE
European ERS-1/ERS-2	1992-2001(-2011)	35 (1,3,183)	C / 6	1 strip map	L0
European Envisat	2003-Sep.2010, Oct. 2010-Apr. 2012	35 (30)	C / 6	7 standard modes, including dual-pol	L0
Japanese ALOS	Jan. 2006–Apr. 2011	46	L / 24	SIngle, dual and quad-pol modes	L0 (called L1.0)
German TerraSAR-X TanDEM-X	2007 - present 2010 - present	11	X / 3	Variable resolution and beam pointing	L1 only
Italian COSMO-SkyMed 4 Satellites	2007 - present	16 (1,4,7,8)	X / 3	Variable resolution and beam pointing	L0 L1
Canadian Radarsat-2	2007 - present	24	C / 6	Variable resolution and beam pointing	L1 only

Other sensors, including Japan’s JERS-1, and Canada’s Radarsat-1, have orbit control and knowledge factors that ISCE cannot currently handle for interferometry, so these are not fully supported. In the future, we will be adding them to the available data sets. ISCE understands the individual formats of the Level 0 or Level 1 data and converts them to a standard internal format that is uniform across sensors within ISCE. In this way, the stripmap and interferometric processing can proceed identically for all sensors.

Level 0 (L0; for ALOS, called Level 1.0) data are raw radar pulses that have not yet been processed to imagery but have been conditioned to remove downlink telemetry and fix data transmission errors such as timing glitches and data dropouts. Level 1 (L1; for ALOS, called Level 1.1) data are processed to form complex radar images, often called “single-look complex” images (SLC images). A complex image is a two-dimensional pixel array of complex numbers (real and imaginary parts) which represent the backscattered amplitude and phase of each pixel.

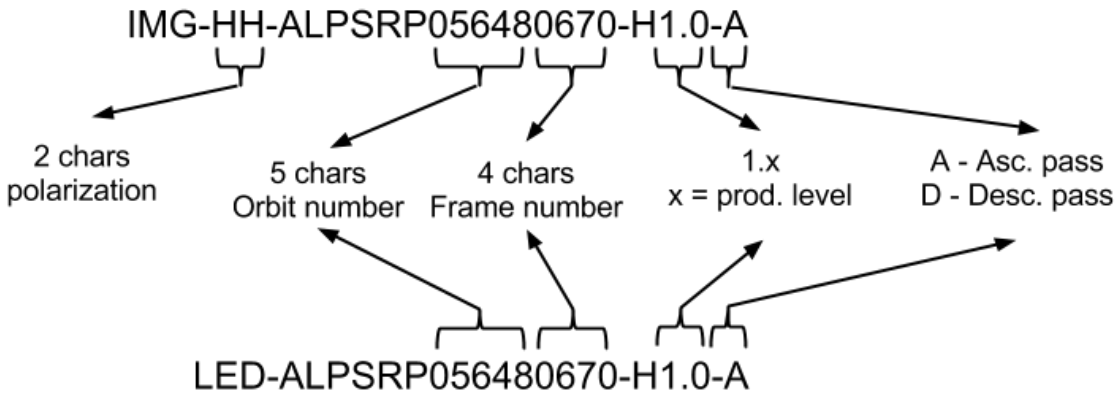
Other modes that many of these sensors potentially can deliver are spotlight mode and ScanSAR mode, the former for focused areas at fine resolution, the latter for broad areas at coarse resolution. Since interferometry is more challenging in both these modes, and data availability is limited, ISCE does not support processing raw data from these modes currently. Spotlight data already processed to Level 1 (SLC) images, including COSMO-SkyMed and TerraSAR-X, can be processed in ISCE if it was acquired with interferometric compatibility.

This lab contains modules 3.1, 3.2, 3.3, etc., starting with a processing example from ALOS PALSAR in module 3.1, and then some discussions about the ISCE outputs. Later labs 4-7 cover other sensors. To a large extent, these modules are independent, so students can run only those sensor modules in which they have interest.

1.Understanding ALOS PALSAR Data Set Names

We'll start with a popular data type: The PALSAR L-band data from Japan's ALOS satellite, which operated between 2006 and 2011. We will describe the names these files are given by the data provider, and how these can be placed in the ISCE input files for processing.

Were you to download PALSAR data from a data provider, each frame comprises an image data file and a image leader file, as well as possibly some other ancillary files that are not used by ISCE. The leader file contains parameters of the sensor that are relevant to the imaging mode, all the information necessary to process the data. The data file contains the raw data samples if Level 1.0 raw data (this is just a different name from what other satellites call Level 0) and processed imagery if Level 1.1 or 1.5 image data. The naming convention for these files is standardized across data archives, and has the following taxonomy:



Files with IMG as prefix are images. Files with LED as prefix are leaders. We will describe how to find and download these data shortly. But first let's see how these filenames are specified in the inputs to ISCE. ISCE at present only supports processing the raw or Level 1.0 PALSAR data.

2. Inserting ALOS PALSAR filenames into the ISCE xml input files

Now it is time to take a look at the input file that we used in lab1 when we ran insarApp.py. First you should change your working directory to the lab1 directory. Recall from lab1 that you use the command `pwd` to see the location of the directory you are currently in and you use the command `cd` to change directories. For your current directory you should see,

```
> pwd
/home/ubuntu
```

Now cd into the lab1 processing directory where you ran the ISCE application insarApp.py.

```
> cd data/lab1/20070215_20061231
```

You can use the `ls` command discussed in lab1 to see the files in this directory. For now we will be concerned only with the three files, `insar_20070215_20061231.xml`, `Master.xml`, and `Slave.xml`.

```
> ls -l insar_20070215_20061231.xml Master.xml Slave.xml
insar_20070215_20061231.xml
Master.xml
Slave.xml
```

Use the Unix command `cat` (for catenate), to see the contents of the input file:

```
> cat insar_20070215_20061231.xml
<insarApp>
<component name="insarApp">
    <property name="sensor name">
        <value>ALOS</value>
    </property>
    <component name="Master">
        <catalog>Master.xml</catalog>
    </component>
    <component name="Slave">
        <catalog>Slave.xml</catalog>
    </component>
</component>
</insarApp>
```

This is an *xml* file. The format of this type of file may seem unfamiliar or strange to you, but

with the following description of the basics of the format, it will hopefully become more familiar. The first thing to point out is that the indentations and line breaks seen above are not required and are simply used to make the structure more clear and the file more readable to humans. The xml file provides structure to data for consumption by a computer. As far as the computer is concerned the data structure is equally readable if all of the information were contained on a single very long line, but human readers would have a hard time reading it in that format.

The next thing to point out is the method by which the data are structured through the use of *tags* and *attributes*. An item enclosed in the < (less-than) and > (greater-than) symbols is referred to as a *tag*. The name enclosed in the < and > symbols is the *name* of the *tag*. Every tag in an xml file **must** have an associated closing tag that contains the same name but starts with the symbol `</` and ends with the symbol `>`. This is the basic unit of structure given to the data. Data are enclosed inside of opening and closing tags that have names identifying the enclosed data. This structure is nested to any order of nesting necessary to represent the data. The Python language (in which the ISCE user interface is written) provides powerful tools to parse the xml structure into a data structure object and to very easily "walk" through the structure of that object.

In the above xml file the first and last tags in the file are a tag pair: `<insarApp>` and `</insarApp>` (note again, tags **must** come in pairs like this). The first of these two tags, or the *opening tag*, marks the beginning of the contents of the tag and the second of these two tags, or the *closing tag*, marks the end of the contents of the tag. ISCE expects a "file tag" of this nature to bracket all inputs contained in the file. The actual name of the file tag, as far as ISCE is concerned, is user selectable. In this example it is used, as a convenience to the user, to document the ISCE application, named `insarApp.py`, for which it is meant to provide inputs; it could have been named `<foo>` and insarApp.py would have been equally happy provided that the closing tag were `</foo>`.

The next tag is `<component name="insarApp">`. Its closing tag `</component>` is located at the penultimate line of the file (one line above the `</insarApp>` tag). The name of this tag is `component` and it has an *attribute* called `name` with value `"insarApp"`. The component tags bound a collection of information that is used by a computational element within ISCE that has the name specified by the `name` attribute. The name `"insarApp"` in the first `component` tag tells ISCE that the enclosed information correspond to a functional component in ISCE named `"insarApp"`, which in this case is actually the application that is run at the command line.

In general, `component` tags contain information in the form of other `component` tags or `property` tags, all of which can be nested to any required level. In this example the `insarApp` component contains a `property` tag and two other `component` tags.

The first tag we see in the `insarApp component` tag is the `property` tag with attribute `name="sensor name"`. The `property` tag contains a `value` tag that contains the name

of the sensor, ALOS in this case. The next tag is a component tag with attribute name="Master". This tag contains a catalog tag containing Master.xml. The catalog tag in general informs ISCE to look in the named file (Master.xml in this case) for the contents of the current tag. The next component tag has the same structure with the catalog tag containing a different file named Slave.xml.

The contents of the Master.xml and Slave.xml files are the following:

```
> cat Master.xml
<component name="Master">
    <property name="IMAGEFILE">
        <value>../20070215/IMG-HH-ALPSRP056480670-H1.0__A</value>
    </property>
    <property name="LEADERFILE">
        <value>../20070215/LED-ALPSRP056480670-H1.0__A</value>
    </property>
    <property name="OUTPUT">
        <value>20070215.raw</value>
    </property>
</component>

> cat Slave.xml
<component name="Slave">
    <property name="IMAGEFILE">
        <value>../20061231/IMG-HH-ALPSRP049770670-H1.0__A</value>
    </property>
    <property name="LEADERFILE">
        <value>../20061231/LED-ALPSRP049770670-H1.0__A</value>
    </property>
    <property name="OUTPUT">
        <value>20061231.raw</value>
    </property>
</component>
```

The component tag that contains the information in each of these files (named “Master” and “Slave”) can be found also in the file insar_20070215_20061231.xml surrounding the <catalog> entries that specify these filenames. The Master.xml and Slave.xml files each contain three property tags that give the names of the IMAGEFILE, LEADERFILE, and the OUTPUT file. The ALOS PALSAR data are delivered with the IMAGEFILE and LEADERFILE plus a few other files that are not used by ISCE. You may choose any name you like for the OUTPUT filename. The OUTPUT filename is the name of the raw file that ISCE creates in its initial steps of processing. In the above example, we have chosen a ROI_PAC style convention of using the date in the format yyyymmdd (year month day). The base of the name you give

(the part of the name before the `.raw`) is also used in the name of the single-look complex files (`SLC`s) created by ISCE.

The `<value>` tag for the properties `IMAGEFILE` and `LEADERFILE` in `Master.xml` and `Slave.xml` contain the symbol `/` (commonly referred to as slash) in its name, which indicates that these are *paths* in the file system. The <value> tag for the output file does not contain any `/` symbols, which indicates that the file will be located in the directory from where the processing command is issued, which was the `/home/ubuntu/lab1/20070215_20061231` directory in lab1. The paths used in these example files begin with the symbol `../` which indicates that they are *relative paths* from where we are to where the files are located. The other type of path is an *absolute path* and would start with the `/` symbol without the leading two dots as in the result of the `pwd` command (see above for example).

To understand how to interpret the relative path consider, for instance, the `IMAGEFILE` given in the `Master.xml` file where we find the value,

```
../20070215/IMG-HH-ALPSRP056480670-H1.0__A
```

The `../` part of this name indicates to look one directory above the current directory. Then the `20070215` part indicates to look in the directory `20070215` found relative to there (*i.e.,* the directory `20070215` located in the directory one directory above the current directory). Finally, the `IMG-HH-ALPSRP056480670-H1.0__A` part names the `IMAGEFILE` located in that directory. To further help you understand relative paths, try the following commands:

```
> pwd
/home/ubuntu/data/lab1/20070215_20061231
> cd ../
> pwd
/home/ubuntu/data/lab1/
> ls
20061231  20070215  20070215_20061231  DEM
> cd 20070215
> pwd
/home/ubuntu/data/lab1/20070215
> ls
IMG-HH-ALPSRP056480670-H1.0__A  LED-ALPSRP056480670-H1.0__A
```

As you follow these steps you are following the relative path given in the `Master.xml` file and you see that the `IMAGEFILE` and `LEADERFILE` found in that directory are those given in the `Master.xml` file.

Now use the `cd` command (in one step) to go back to the processing directory and use the `ls` command to view the contents of the `20070215` directory without moving to that directory,

```
> cd ../20070215_20061231
> pwd
/home/ubuntu/data/lab1/20070215_20061231
> ls Master.xml
Master.xml
> ls ../20070215
IMG-HH-ALPSRP056480670-H1.0__A  LED-ALPSRP056480670-H1.0__A
```

You can see that the result of this last `ls` command issued from the directory `20070215_20061231` (where `Master.xml` is located) is the same as above where we used the `cd` command to change directories to the `../20070215` directory.

Note, in this example the relative paths involved a single `../` symbol in naming the relative path. A relative path in general may contain any number of `../` symbols and directory names necessary to locate the directory tree where the files are. Each `../` indicates to look one directory above the directory pointed to by any previous chain of `../` symbols. For example,

```
../../dir1/file1
```

points to a file named `file1` located in a directory named `dir1` located two directories *above* the current directory. We say a directory `dir1` is *above* directory `dir2` if `dir1` contains `dir2`, i.e., if the `ls` command used in `dir1` shows `dir2` in its listing of files and directories. Another example indicating a relative path going up and down the directory trees relative to the current directory: the relative path,

```
../../../../dir1/dir2/file1
```

indicates that `file1` is found by going up 4 directories from the current directory and then down from there into `dir1` and then `dir2`.

An alternative to using the relative path would be to use the absolute path, which is the path shown by the `pwd` command above when we changed directories to the `20070215` directory where the `IMAGEFILE` and `LEADERFILE` were found. Using the absolute path, the `IMAGEFILE` tag would look as follows:

```
    <property name="IMAGEFILE">
        <value>

/home/ubuntu/data/lab1/20070215/IMG-HH-ALPSRP056480670-H1.0__A
        </value>
    </property>
```

Remember that the line breaks and indentations in the xml file are not interpreted by the computer and are only used to improve readability for humans. The absolute path method for the `LEADERFILE` would look similar in an obvious way except with the name of the leader file after the final `/` in the path. You are free to choose whether to use absolute paths or relative paths or a combination of both (for whatever reason).

The choice between the use of absolute and relative paths could involve more than a question of style. If you are doing a very small project, such as in this tutorial, then it matters little which you choose. If there ends up being a long chain of `../` symbols to point to the input files, then an absolute path may be more readable. If you are working on a large project involving many processing runs and a complex directory structure, then the use of absolute paths could result in a waste of time and money when the project directory tree is moved within the file system or to another computer and the absolute paths in the input files have to be modified. The benefit of using relative paths is that if an entire project data directory tree were moved from one location to another on the same file system or to another computer, while preserving the internal structure of the data directory tree, then all of the input files that use relative paths that point to paths in the project data directory tree will continue to work without modification. Any input files with absolute paths will have to be modified, which could be a very costly and laborious process.

The ISCE input data in the above example were split between three different files, `insarApp.xml`, `Master.xml`, and `Slave.xml`. An alternative is to use a single file containing all of the needed information as in the following:

```
<insarApp>
<component name="insarApp">
    <property name="sensor name">
        <value>ALOS</value>
    </property>
    <component name="Master">
        <property name="IMAGEFILE">
            <value>../20070215/IMG-HH-ALPSRP056480670-H1.0__A</value>
        </property>
        <property name="LEADERFILE">
            <value>../20070215/LED-ALPSRP056480670-H1.0__A</value>
        </property>
        <property name="OUTPUT">
            <value>20070215.raw</value>
        </property>
    </component>
    <component name="Slave">
        <property name="IMAGEFILE">
            <value>../20061231/IMG-HH-ALPSRP049770670-H1.0__A</value>
```

```
        </property>
        <property name="LEADERFILE">
            <value>../20061231/LED-ALPSRP049770670-H1.0__A</value>
        </property>
        <property name="OUTPUT">
            <value>20061231.raw</value>
        </property>
    </component>
</component>
</insarApp>
```

A final point on relative paths: They are interpreted relative to the current working directory. Thus if you are working in directory A, but you have an xml file in directory B below A that references `../file.dat`, this will resolve to a path *a level above* A, not at level A.

There are many more possible input options for commanding the processing that we will reveal as we go along in these tutorials. In the next step of this tutorial you will pick one of these styles for input files and try processing some ALOS data using ISCE. The details of the different input files for the other types of sensors supported by ISCE can be found at the following links.

3. Processing ALOS PALSAR data with ISCE

It is time to test your understanding of the input files needed to run `insarApp.py` by creating your own input files for a new pair of ALOS PALSAR images. In this exercise, you will create the necessary input files based on the examples provided in Step 2. To create these files you will need to be able to use a text editor on the virtual machine. Many of you are familiar with text editors like "vi" or "emacs" and you are welcome to use them. For those unfamiliar with text editors, the virtual machine instance provides a simple tool call "nano" that has a few basic "control commands" to open and close files, cut and paste text, etc. It is mostly self-explanatory, but you can look at this tutorial for more information. So let's get started. First we need to position ourselves in the directory where these new data reside:

```
> cd
> cd data/lab3
```

The first `cd` command simply sends you back to your home directory. The second positions you at the level where the data for this lab resides. Let's see what's in this directory:

```
> ls
alos
```

For the moment, we are interested in ALOS PALSAR, so we will position ourselves there:

```
> cd alos
> ls
20070612  20090802
```

These names are directories containing the ALOS data for two dates, one in 2007 and the other in 2009. We can examine the contents:

```
> ls 20070612
IMG-HH-ALPSRP073630230-H1.0__A  LED-ALPSRP073630230-H1.0__A
IMG-HV-ALPSRP073630230-H1.0__A
> ls 20090802
IMG-HH-ALPSRP187700230-H1.0__A  LED-ALPSRP187700230-H1.0__A
IMG-HV-ALPSRP187700230-H1.0__A
```

Now it is time to create the input files as above. To organize your data, let's create a new directory where all the results will go:

```
> mkdir 20070612_20090802
> cd 20070612_20090802
```

At this point, you must create the input files. As described above you have a choice to create one input file that contains all information or to spread the information across three files. If you choose to create it all in one input file, start by creating an empty file:

```
> touch insar_allinput.xml
```

The `touch` command simply creates an empty file if that file does not already exist. If it does exist, it simply updates the modification date. If you choose to create three files, start by creating three empty files:

```
> touch insar_input.xml
> touch 20070612.xml
> touch 20090802.xml
```

Whichever style you choose, with the information provided in Step 2 above and armed with your favorite text editor, you should be able to construct your input files with the appropriate information.

Go for it! When you think your input files are ready, you have can either “play it safe” or “play it risky”. If you want to play it safe, look at these examples to see what these files should look like. If you want to play it risky, just run the processor script!

```
> insarApp.py insar_allinput.xml
```

or

```
> insarApp.py insar_input.xml
```

Go get another cup of coffee, and come back in about 20 minutes while the processing occurs. If the program terminates unexpectedly because of an input error, compare your files to the examples.

4. Your completed run

After insarApp.py completes, you should see a text message on your screen similar to the following:

```
.
.
.
      runGeocode - Outputs
------------------------------------------------------------------------
------------------------------
runGeocode.outputs.MINIMUM_GEO_LONGITUDE = 40.3883333333
runGeocode.outputs.MAXIMUM_GEO_LATITUDE = 11.0975
runGeocode.outputs.MAXIMUM_GEO_LONGITUDE = 41.2466666667
runGeocode.outputs.GEO_LENGTH = 2048
runGeocode.outputs.LONGITUDE_SPACING = 0.000833333333333
runGeocode.outputs.LATITUDE_SPACING = -0.000833333333333
runGeocode.outputs.MINIMUM_GEO_LATITUDE = 12.8033333333
runGeocode.outputs.GEO_WIDTH = 1031
########################################################################
##############################
2013-07-10 00:50:24,564 - isce.insar - INFO - Total Time: 709 seconds
```

Note for your run, the date and times will be different, and the `Total Time` may be longer or shorter than 709 seconds, depending on the kind of virtual machine you are running.

Congratulations you have successfully run ISCE for an ALOS data set. You can view the list of output files that were generated by insarApp.py using the ls command. You should see the following list of files:

```
> ls

20070612.xml                                        insar.log
20090802.xml                                        insarProc.xml
azimuthOffset.mht                                   isce.log
azimuthOffset.mht.xml                          lat.rdr
catalog                                                lon.rdr
dem.crop                                            rangeOffset.mht
demLat_N11_N14_Lon_E040_E042.dem           rangeOffset.mht.xml
demLat_N11_N14_Lon_E040_E042.dem.wgs84     resampImage.amp
demLat_N11_N14_Lon_E040_E042.dem.wgs84.xml  resampImage.amp.xml
demLat_N11_N14_Lon_E040_E042.dem.xml       resampImage.int
```

```
filt_topophase.flat
resampImage.int.xml
filt_topophase.flat.geo
resampOnlyImage.amp
filt_topophase.flat.geo.xml
resampOnlyImage.int
filt_topophase.flat.xml
resampOnlyImage.int.xml
IMG-HH-ALPSRP073630230-H1.0__A.raw          simamp
IMG-HH-ALPSRP073630230-H1.0__A.raw.aux      topophase.cor
IMG-HH-ALPSRP073630230-H1.0__A.raw.xml      topophase.cor.xml
IMG-HH-ALPSRP073630230-H1.0__A.slc          topophase.flat
IMG-HH-ALPSRP073630230-H1.0__A.slc.xml      topophase.flat.xml
IMG-HH-ALPSRP187700230-H1.0__A.raw          topophase.geo
IMG-HH-ALPSRP187700230-H1.0__A.raw.aux      topophase.geo.xml
IMG-HH-ALPSRP187700230-H1.0__A.raw.xml      topophase.mph
IMG-HH-ALPSRP187700230-H1.0__A.slc          topophase.mph.xml
IMG-HH-ALPSRP187700230-H1.0__A.slc.xml      z.rdr
insar_allinput.xml                          zsch.rdr
insar_input.xml
```

The listing from your processing run may be different from what you see above, as the ISCE is continuously under development, and these labs will use the latest version of the software. However, most should have identical names, and you can using your knowledge of mdx.py from Lab 2 to explore many of these files easily. Similarly, there may be small differences in the displayed images or phase values relative to the examples in these tutorials.

At this point you can continue on to Lab 3.2 to explore in detail the output files you see in the above listing or you can jump ahead to learn about running insarApp on the datasets from the other sensors supported by ISCE in the Labs 4–7.

1. ALOS xml input file examples

The contents of `insar_input.xml` should look like the following (where the names of the files, `20070612.xml` and `20090802.xml`, are most likely different from the names you may have chosen; the names don't matter as long as they are correctly identifying the files containing the data indicated below):

```
<insarApp>
<component name="insarApp">
      <property name="sensor name">
          <value>ALOS</value>
      </property>
      <component name="Master">
          <catalog>20070612.xml</catalog>
      </component>
      <component name="Slave">
          <catalog>20090802.xml</catalog>
      </component>
</component>
</insarApp>
```

The contents of `20070612.xml` should look like:

```
<component name="Master">
      <property name="IMAGEFILE">
          <value>../20070612/IMG-HH-ALPSRP073630230-H1.0__A</value>
      </property>
      <property name="OUTPUT">
          <value>IMG-HH-ALPSRP073630230-H1.0__A.raw</value>
      </property>
      <property name="LEADERFILE">
          <value>../20070612/LED-ALPSRP073630230-H1.0__A</value>
      </property>
</component>
```

The contents of `20090802.xml` should look like:

```
<component name="Slave">
      <property name="IMAGEFILE">
          <value>../20090802/IMG-HH-ALPSRP187700230-H1.0__A</value>
      </property>
      <property name="OUTPUT">
```

```
        <value>IMG-HH-ALPSRP187700230-H1.0__A.raw</value>
    </property>
    <property name="LEADERFILE">
        <value>../20090802/LED-ALPSRP187700230-H1.0__A</value>
    </property>
</component>
```

The contents of `insar_allinput.xml` should look like the following:

```
<insarApp>
<component name="insarApp">
    <property name="sensor name">
        <value>ALOS</value>
    </property>
    <component name="Master">
    <property name="IMAGEFILE">
        <value>../20070612/IMG-HH-ALPSRP073630230-H1.0__A</value>
    </property>
    <property name="OUTPUT">
        <value>IMG-HH-ALPSRP073630230-H1.0__A.raw</value>
    </property>
    <property name="LEADERFILE">
        <value>../20070612/LED-ALPSRP073630230-H1.0__A</value>
    </property>
    </component>
    <component name="Slave">
    <property name="IMAGEFILE">
        <value>../20090802/IMG-HH-ALPSRP187700230-H1.0__A</value>
    </property>
    <property name="OUTPUT">
        <value>IMG-HH-ALPSRP187700230-H1.0__A.raw</value>
    </property>
    <property name="LEADERFILE">
        <value>../20090802/LED-ALPSRP187700230-H1.0__A</value>
    </property>
    </component>
</component>
</insarApp>
```

1. Understanding ISCE Output Files and Formats

If you ran the lab 3.1 or labs 4–7, you will have a directory containing many output files. These files are the outputs at various stages of the workflow, and can be explored with mdx.py to gain insight into their characteristics: dimensions, intrinsic feature characteristics, noise characteristics, and other aspects. Each of these files is either a “flat” binary file containing numbers that represent the images with no other formatting information, or an XML metadata file describing the attributes of a corresponding image. In all cases, the binary data file has the form <prefix>.<extension> and its corresponding XML metadata file has the form <prefix>.<extension>.xml. The file extensions and their implication as to the data type stored in the file and function in the ISCE workflow is given in the following table.

File Extension	Datatype	Description	ISCE outputs.
.raw	Byte, two channel	Byte samples for I and Q channels. (BIP format)	Used to store raw radar echoes.
.slc	complex64, single channel	Complex floating point data, 8 bytes per sample, 4 for real and 4 for imaginary	Used to store SLCs. Short for single look complex image.
.int	complex64, single channel	same as .slc	Used to store complex valued interferograms. Short for interferogram.
.amp	float32, two channels	One line of first amplitude channel, followed by one line of channel 2. (BIL format)	Used to store amplitude files. Short for amplitude.
.cor	float32, two channels	One line of amplitude, followed by one line of coherence (BIL format)	Used to store coherence files. Short for correlation.
.unw	float32, two channels	One line of amplitude followed by one line of unwrapped phase in radians (BIL format)	Used to store unwrapped phase files. Short for unwrapped.
.mph	complex64, one channel	same as .slc	Used for simulated data to clearly indicate that the contents do not contain real radar observations.

			Short for magnitude/phase.
.flat	complex64, one channel	same as .slc	Used to store interferograms whose topography phase component has been removed. Short for flattened.
.mht	float32, two channels	One line of amplitude, followed by one line of data (BIL format)	Used for simulate data to clearly indicate that the contents do not contain real radar observations. Short for magnitude/height.
.geo	Variable	Depending on the basename of the file	.geo is used to indicate that the file contains geocoded data.
.rdr	float32, one channel	Depending on the file basename	This extension is used to indicate geometry parameters like lat,lon, z etc. in radar coordinates.

As an example, let's look at a ".int" file. Change your directory to the outputs of Lab 3.1.

```
> cd ~/data/lab3/alos/20070612_20090802
> ls *.int *.int.xml
resampImage.int  resampImage.int.xml  resampOnlyImage.int
resampOnlyImage.int.xml
```

`resampImage.int` is a binary file containing a floating point representation of complex numbers: 4 bytes for the real part followed by 4 bytes for the imaginary part. Using the unix `more` or `less` commands, you can examine some of the metadata fields. For example we see in `resampImage.int.xml` some of the fields:

```
      <property name="DATA_TYPE">
          <value>CFLOAT</value>
      </property>
      <property name="IMAGE_TYPE">
          <value>cpx</value>
      </property>
...
      <property name="NUMBER_BANDS">
          <value>1</value>
      </property>
...
      <property name="WIDTH">
          <value>5194</value>
      </property>
      <property name="LENGTH">
          <value>5529</value>
```

```
    </property>
    <property name="SCHEME">
        <value>BIP</value>
    </property>
```

These attributes state that the file data types are complex floating point numbers, that the file is BIP (band interleaved by pixel) with only 1 band, and has the number of samples across ("WIDTH") of 5194 pixels and number of samples down ("LENGTH") of 5529 pixels, making for an image with 28,717,626 complex pixels, with each pixel being 8 bytes. So the total size of the binary data file should be 229,741,008 bytes. And indeed that is the file's size:

```
> ls -l resampImage.int
-rw-rw-r-- 1 ubuntu ubuntu 229741008 Jul 25 19:02 resampImage.int
```

Feel free to explore other files and formats. The `.amp` file is an unusual format for instance, being a 2-band BIP file with each of the image amplitudes in the two bands. For `insarApp.py`, the final product is a `.geo` file, which is a geocoded version of an interferogram or other derived product. The band interleaving scheme for `.geo` files varies depending on how the data were derived. `topophase.flat.geo` is a complex 1-band BIP file like the `.int` file, as is `filt_topophase.flat.geo`. If an unwrapping filter had been applied to the interferogram, however, the file would be BIL (band interleaved by line) with two bands.

1. Exploring the `insarApp.py` processing option space

`insarApp.py` represents the simplest kind of InSAR processing workflow, that of taking two images acquired from nearly the same vantage point in orbit but at different times and creating an interferogram in geocoded coordinates that represents any motion on the ground that may have occurred between these times. In subsequent labs, we will see more sophisticated processing of a time series of data allowing us to track these changes over time. First, however, we will illustrate some of the flexibility built into the insarApp.py workflow. These flexibilities are built into the framework, and relate to configurable parameters of individual processing "components," so these would be applicable to other workflows built from these components.

There are multiple ways to control the workflow. One way is to alter one of the configurable parameters in the input xml file that controls `insarApp.py`. The list of configurable parameters can be seen by using the `--help` command line option, which also prints a usage statement. The other way is to manipulate the steps of the work using the `--steps` command line option. `--steps` allows the user to start the processing from a particular point in the workflow and end it at another location. Clearly the processing cannot be started beyond a point where a previous processing run completed (for a fresh data set, you must start at the beginning!), but `--steps` allows the user to run individual workflow components one at a time or in sequence allowing the alteration of input parameters for each workflow component.

In this lab, we will exercise the `--steps` option to prepare the raw data for a data set, then alter a processing parameter to reduce the total size of the data to be processed. Once we then process all the way through, we will alter another processing parameter to allow phase unwrapping of the result. With these simple examples, we will convey the main ideas of flow control, and you will then be prepared to experiment on your own with other parameters and steps options.

2. Preparing the raw data

Let's first look at what is possible with `--steps`.

```
> insarApp.py --steps --help
2013-09-18 00:52:07,992 - isce.insar - INFO - ISCE VERSION = 1.0.0,
RELEASE_SVN_REVISION = 739,RELEASE_DATE = 20120814, CURRENT_SVN_REVISION = 1154M
ISCE VERSION = 1.0.0, RELEASE_SVN_REVISION = 739,RELEASE_DATE = 20120814,
CURRENT_SVN_REVISION = 1154M

       Insar Application:
       Implements InSAR processing flow for a pair of scenes from
       sensor raw data to geocoded, flattened interferograms.

A description of the individual steps can be found in the README file
and also in the ISCE.pdf document

Use command line options '--start=<step>', '--end=<step>', --dostep=<step> to choose
the step names from the following list:

self.step_list =  ['startup', 'preprocess', 'verifyDEM', 'pulsetiming',
'estimateHeights', 'mocompath', 'orbit2sch', 'updatepreprocinfo', 'formslc',
'offsetprf', 'outliers1', 'prepareresamps', 'resamp', 'resamp_image',
'mocompbaseline', 'settopoint1', 'topo', 'shadecpx2rg', 'rgoffset', 'rg_outliers2',
'resamp_only', 'settopoint2', 'correct', 'coherence', 'filter', 'unwrap', 'geocode',
'endup']

If --start is missing, then processing starts at the first step.
If --end is missing, then processing ends at the last step.
If --dostep is used, then only the named step is processed.
```

Note that each of the names in the list called `self.step_list` are workflow component names, each carrying out a specific function briefly described in the table below (see ISCE.pdf for a description of each component).

Step name	**Short functional description**
startup	Initialization of python objects for interferogram processing.
preprocess	Extract raw radar echoes from original sensor files and store them in an ISCE compatible format. Populate metadata fields for use in processing.
verifyDEM	Check if the user has provided a DEM. If not download a DEM from the SRTM archive.
pulsetiming	Determine antenna position for every raw echo line by interpolating the

	state vectors.
estimateHeights	Estimates the average heights for each of the SAR acquisitions from the interpolated state vectors.
mocomppath	Determines the reference mocomp orbit for focusing the SAR acquisitions.
orbit2sch	Transforms the state vector information for both SAR acquisitions into the SCH coordinate system, while accounting for the reference mocomp orbit.
updatepreprocinfo	Updates the parameters with common values for SAR focusing.
formslc	Focuses raw radar echoes into a single-look complex image.
offsetprf	Estimates offsets between the master and slave image while accounting for slight differences in the PRFs.
outliers1	Culls the offset field by removing noisy offset estimates.
prepareresamps	Setup the resampling routine for interferogram generation.
resamp	Resample the slave SLC and cross multiply with the master SLC, to create an interferogram.
resamp_image	Dump the offset field that was used for resampling as images.
mocompbaseline	Estimate the baseline to be used for topography removal, on a line by line basis.
settopoint1	Sets the file names for the output of the topo module. To be read as <set_topo_int1>.
topo	Estimate the DEM in radar coordinates using the master orbit information.
shadecpx2rg	Simulate an amplitude image from the estimated DEM in radar coordinates.
rgoffset	Determine offset field between interferogram and simulated amplitude.
rg_outliers2	Cull the offsetfield to remove noise offset estimates.
resamp_only	Resample the interferogram to match the DEM. In this paradigm, we trust the orbits and the geometry more than the focusing modules.
settopoint2	Sets the file names for the correct module. <To be read as set_topo_int2>.

correct	Remove the topography component of phase using the outputs of topo module and mocompbaseline.
coherence	Estimate the coherence from the topo-corrected interferogram.
filter	Filter the corrected interferogram using an adaptive filter. Also estimate the coherence for filtered interferogram using the phase standard deviation.
unwrap	Unwrap the interferogram using method of choice.
geocode	Geocode the requested set of outputs.
endup	Clean up and close files as needed.

For our purposes, we simply want to prepare the data for image formation. `formSLC` is the image formation component, so we want to run the following workflow:

```
> cd /home/ubuntu/data/lab3/alos/20070612_20090802
> insarApp.py --steps --end="updatepreprocinfo" insar_input.xml
```

This will process from raw data all the way to where the inputs are established for running form SLC. `--steps` creates a `PICKLE` directory which stores all the information needed to restart the process. This directory is referenced when the next run with `--steps` is executed. If you examine the screen output at the end of the above command, you should see:

```
2013-09-18 00:42:14,352 - isce.insar.runFdMocomp - INFO - Updated Doppler
Centroid: 0.0419756826644
Dumping the application's pickle object _insar to file
PICKLE/updatepreprocinfo
The remaining steps are (in order):  ['formslc', 'offsetprf', 'outliers1',
'prepareresamps', 'resamp', 'resamp_image', 'mocompbaseline', 'settopoint1',
'topo', 'shadecpx2rg', 'rgoffset', 'rg_outliers2', 'resamp_only',
'settopoint2', 'correct', 'coherence', 'filter', 'unwrap', 'geocode', 'endup']
```

with the appropriate time stamp for your run. This shows you that you processed successfully up to the step before `formSLC`. Now we will see how to control the processing to process only a portion of the available data.

3. Altering the processing parameters

The best way to see the possible options in a nutshell is with the help function of `insarApp.py`, as follows:

```
> insarApp.py --help
2013-09-18 00:49:20,766 - isce.insar - INFO - ISCE VERSION = 1.0.0,
RELEASE_SVN_REVISION = 739,RELEASE_DATE = 20120814, CURRENT_SVN_REVISION =
1154M
ISCE VERSION = 1.0.0, RELEASE_SVN_REVISION = 739,RELEASE_DATE = 20120814,
CURRENT_SVN_REVISION = 1154M

      Insar Application:
      Implements InSAR processing flow for a pair of scenes from
      sensor raw data to geocoded, flattened interferograms.

The currently supported sensors are:  ['ALOS', 'GENERIC', 'RADARSAT2',
'ENVISAT', 'COSMO_SKYMED_SLC', 'COSMO_SKYMED', 'RADARSAT1', 'ERS', 'TERRASARX',
'JERS']
Usages:
insarApp.py <input-file.xml>
insarApp.py --steps
insarApp.py --help
insarApp.py --help --steps

See the table of configurable parameters listed in the table
below for alist of parameters that may be specified in the
input file.  See example input xml files in the isce 'examples'
directory.  Read about the input file in the ISCE.pdf document.

The user configurable inputs are given in the following table.
Those inputs that are of type 'component' are also listed in
table of facilities below.
To configure these parameters, enter the desired value in the
input file using a property tag with public_name = to the name
given the table
```

name	type	mandatory	doc
sensor name	str	mandatory	Sensor name
slc offset method	str	optional	SLC offset estimation method name. Use value=ampcor to run ampcor

Master	component	mandatory	Master raw data component
slc offsetter	component	optional	SLC offset estimator.
peg longitude (deg)	float	optional	Peg Longitude in degrees
demFilename	str	optional	Filename of the DEM init file
posting	int	optional	posting for interferogram
Slave	component	mandatory	Slave raw data component
Form SLC	component	optional	SLC formation module
use_dop	float	optional	Choose whether to use master, slave, or average Doppler for processing.
range looks	float	optional	Number of range looks to use in resamp
doppler method	str	optional	Doppler calculation method.Choices: 'useDOPIQ', 'useCalcDop', 'useDoppler'.
Run Unwrapper	component	optional	Unwrapping module
useHighResolutionDemOnly	int	optional	If True and a dem is not specified in input, it will only download the SRTM highest resolution dem if it is available and fill the missing portion with null values (typically -32767).
geoPosting	float	optional	Output posting for geocoded images in degrees (latitude = longitude)
peg latitude (deg)	float	optional	Peg Latitude in degrees
offset search window size	int	optional	Search window size used in offsetprf and rgoffset.
unwrap	bool	optional	True if unwrapping is desired. To be unsed in combination with UNWRAPPER_NAME.
geocode list	tuple	optional	List of products to geocode.
unwrapper name	str	optional	Unwrapping method to use. To be used in combination with UNWRAP.
azimuth looks	float	optional	Number of azimuth looks to use in resamp
correlation_method	str	optional	Select coherence estimation method: cchz=cchz_wave phase_gradient=phase gradient
Slave Doppler	component	optional	Master Doppler calculation method
Dem	component	optional	Dem Image configurable component. Do not include this in the input file and an SRTM

			Dem will be downloaded for you.
peg radius (m)	float	optional	Peg Radius of Curvature in meters
peg heading (deg)	float	optional	Peg Heading in degrees
gross range offset	int	optional	Override the value of the gross range offset for offsetestimation prior to interferogram formation
azimuth patch size	int	optional	Size of overlap/save patch size for formslc
pickle dump directory	str	optional	If steps is used, the directory in which to store pickle objects.
gross azimuth offset	int	optional	Override the value of the gross azimuth offset for offset estimation prior to interferogram formation
Culling Sequence	tuple	optional	TBD
patch valid pulses	int	optional	Size of overlap/save save region for formslc
number of patches	int	optional	How many patches to process of all available patches
Master Doppler	component	optional	Master Doppler calculation method
pickle load directory	str	optional	If steps is used, the directory from which to retrieve pickle objects

The help list above illustrates a number of parameters that control the workflow. Rather than describing each one in detail, we will focus on just a few to illustrate how you would go about changing them, and showing the effect of changing them on the processing. Let's start with a simple one: "number of patches". This input parameter would be specified in the .xml input file that is read by insarApp.py. The processing of radar imagery is done in chunks, where several thousand lines of raw data are read, and processed to form a sub-image, then the next chunk is read in with some overlap to create the next subimage, and so forth, until the entire image is processed. These sub-images are then put together to form the complete image. Each chunk is traditionally called a "patch", and the user is allowed to either take the default, which is to process the entire image, or specify the number of patches they wish to process.

First examine our familiar `insarApp.xml` (we used this in Lab 3.1):

```
> more insarApp.xml
<insarApp>
```

```
<component name="insarApp">
    <property name="sensor name">
      <value>ALOS</value>
    </property>
    <component name="Master">
      <catalog>Master.xml</catalog>
    </component>
    <component name="Slave">
      <catalog>Slave.xml</catalog>
    </component>
</component>
</insarApp>
```

Note that most parameters are not specified. Defaults are taken based on the sensor data. Now let's set the number of patches parameter here. Adding the lines

```
    <property name="number of patches">
      <value>1</value>
    </property>
```

will do the trick. We have done this in a new file `insarApp_1patch.xml`, which you can list to verify:

```
> more insarApp_1patch.xml
<insarApp>
<component name="insarApp">
    <property name="sensor name">
      <value>ALOS</value>
    </property>
    <property name="number of patches">
      <value>1</value>
    </property>
    <component name="Master">
      <catalog>Master.xml</catalog>
    </component>
    <component name="Slave">
      <catalog>Slave.xml</catalog>
    </component>
</component>
</insarApp>
```

Now by issuing the following command, we can pick up from where we left off with, specifying that we only want to process one patch of data.

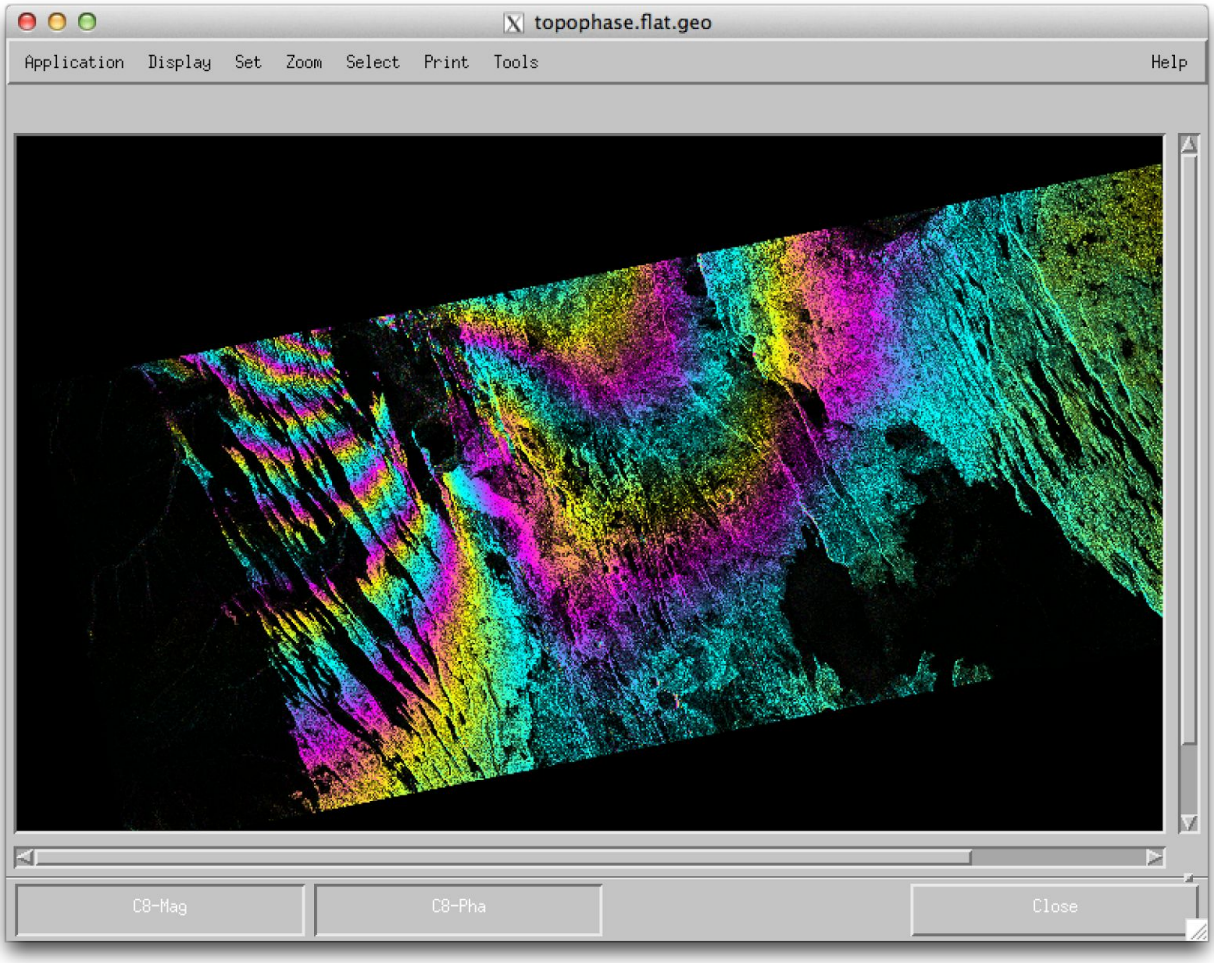
```
> insarApp.py --steps --start='formslc' insarApp_1patch.xml
```

(When `--start` or `--end` is specified, the `--steps` switch is technically not needed.) At the end of this process, you will have completed the full processing run. Now we can play with another processing option: unwrapping.

4. Turning on unwrapping

Now that we've completed the full processing run (for a 1-patch subset of the full image to speed up the demo), we can see that the output contains a geocoded interferogram, but it is not unwrapped.

```
> mdx.py topophase.flat.geo
```



We can see some nice deformation fringes on the top left of the image. To be able to exploit this signature in further analysis such as stack processing, we need to unwrap this image. Having run `--steps` previously, we can restart the process at the unwrapping stage by modifying the `insarApp_1patch.xml` file to include the unwrap option. This change has been made in the file named `insarApp_1patch_unwrap.xml`. The additional lines added to the `insarApp_1patch_unwrap.xml` file for unwrapping are the following (we are using the unwrapper called icu, which is one of a few options available in isce):

```
    <property name="unwrap">
```

```
        <value>True</value>
    </property>
    <property name="unwrapper name">
        <value>icu</value>
    </property>
```

We can now run just the unwrapper to the end of the workflow.

```
> insarApp.py --steps --start='unwrap' insarApp_1patch_unwrap.xml
```

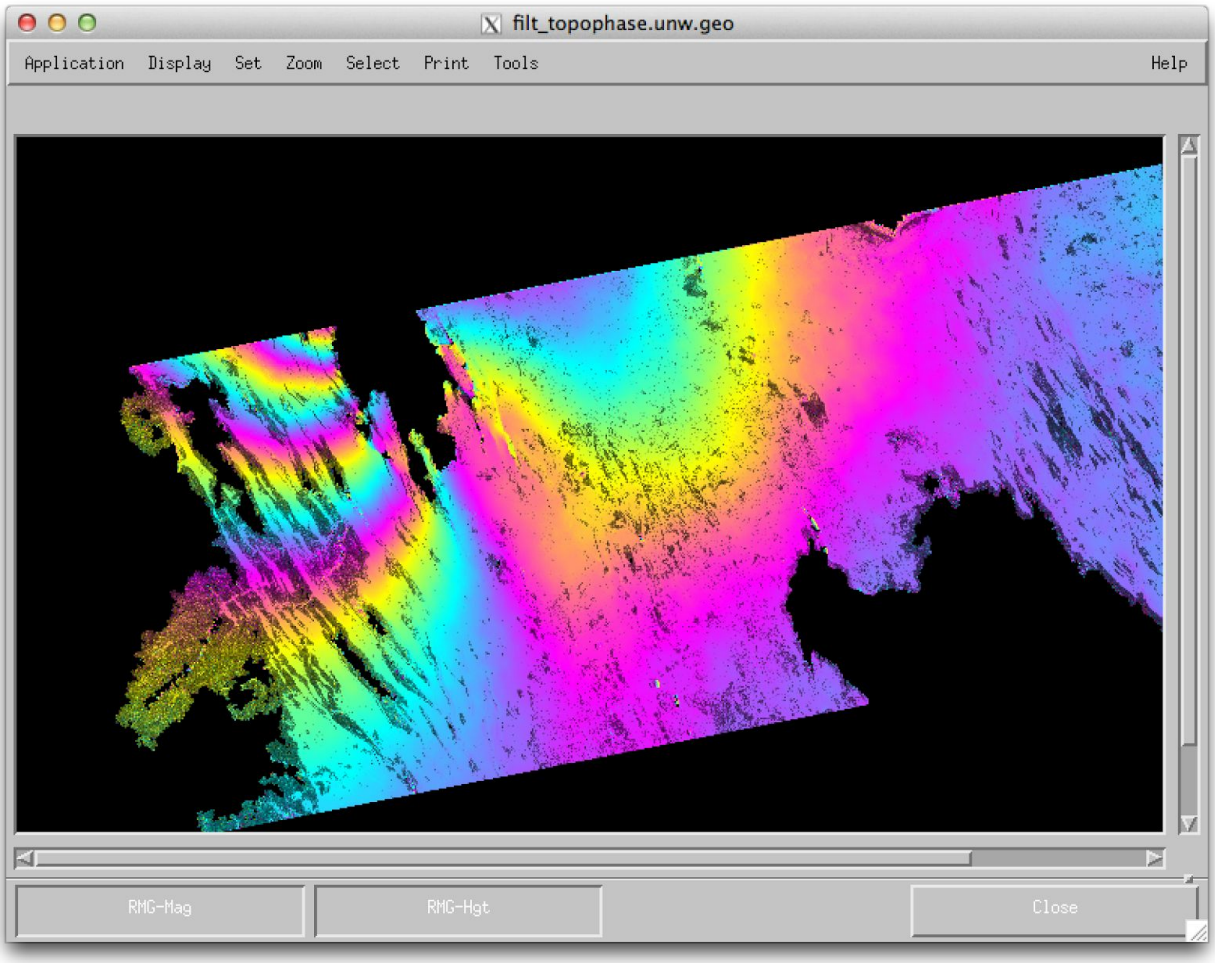
When this is finished, we can see additional files in the directory.

```
> ls -1tr
.
.
.
insarApp_1patch_unwrap.xml
insar.log
filt_topophase.unw
filt_topophase.unw.xml
geo.log
topophase.cor.geo
topophase.cor.geo.xml
filt_topophase.flat.geo
filt_topophase.flat.geo.xml
topophase.flat.geo
topophase.flat.geo.xml
phsig.cor.geo.xml
phsig.cor.geo
los.geo
los.geo.xml
resampOnlyImage.amp.geo
resampOnlyImage.amp.geo.xml
dem.crop
filt_topophase.unw.geo
filt_topophase.unw.geo.xml
catalog
isce.log
insarProc.xml
```

Note that in addition to `topophase.flat.geo`, there are a number of other files. The geocoded unwrapped data is in `filt_topophase.unw.geo`. Using `mdx.py` to display, then

changing the scale to `4*PI` color wrap (right-click on `Pha` button and set wrap to 12.56), you should see:

```
> mdx.py filt_topophase.unw.geo
```



In comparing to the image above, you note that the phase colors are much smoother (due to a filtering operation applied before unwrapping), and the phase is no longer subject to a restriction to the interval from 0 to 2 PI. The black regions are places the unwrapper failed to unwrap, either due to no data on the periphery, or low correlation.

You have now successfully completed the insarApp.py workflow exploring a number of options, including unwrapping. If you are interested in seeing this entire scene rather than the 1-patch subset, feel free to start from the beginning deleting the “number of patches” attribute in the `.xml` file.

In a later lab, you will apply your skills to preparing a stack for time-series processing.

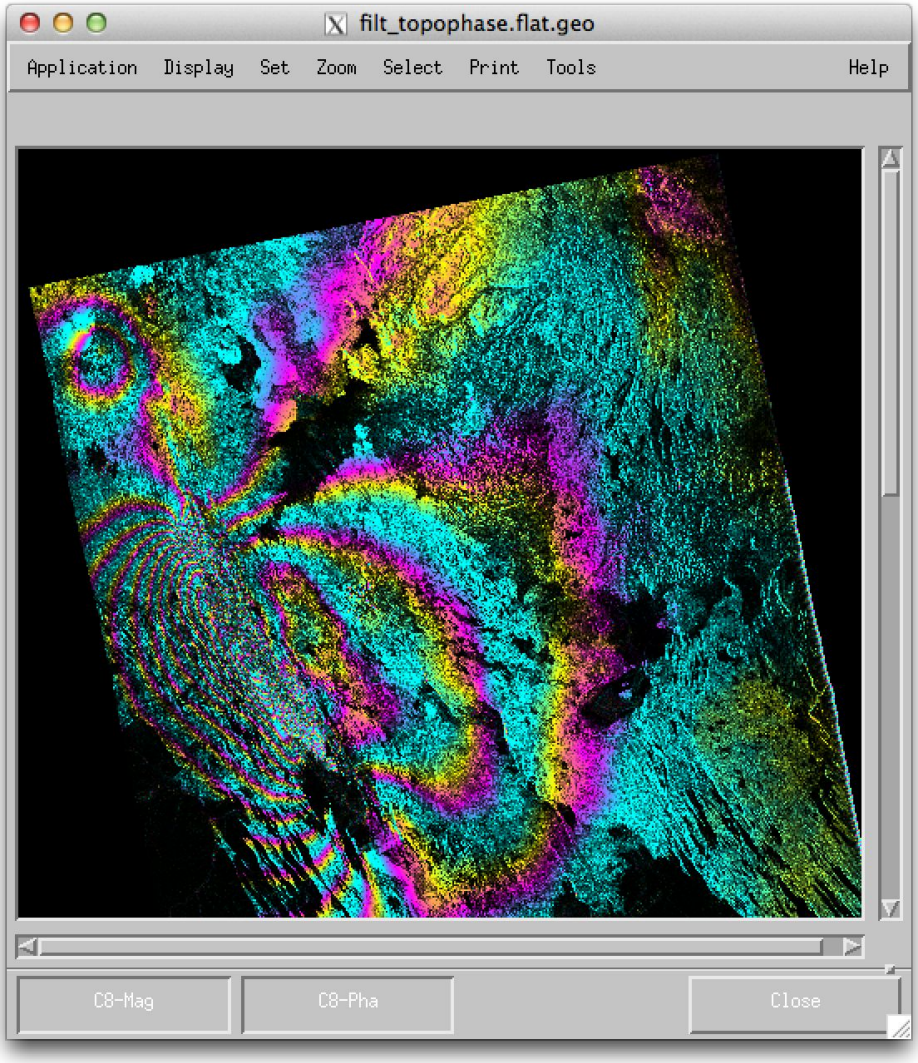
Now that you are experienced at processing data from a variety of sensors and understand the data formats, let's spend some time examining a few data sets to see if we can understand the intrinsic characteristics of the data. We will use two data sets: one from Afar at L-band and one from Hawaii at X-band. These data sets should be familiar to you from the processing labs. Along the way we will expose to you some of the more advanced features of mdx.

First, let's look at the L-band data in the Afar.

```
> cd /home/ubuntu/data/sites/Afar_alos/A599/0230/20070612_20090802
```

To get an overview of the situation, let's display the geocoded interferogram with a zoom factor of -2 (zooming out).

```
> mdx.py -z -2 filt_topophase.flat.geo
```



Note that relative to the borders of the display window, this image is rotated by about 8 degrees

counterclockwise. This is because the image is geocoded to a north-south, east-west grid, but the satellite is on an orbit that is a few degrees off from a north-south orbit. Other than the black border triangles where there is no data for this pair of scenes, the interferogram shows nearly complete coverage. There are several places where there is water in this scene that are also black and show no interferometric fringes, but otherwise the phase is complete.

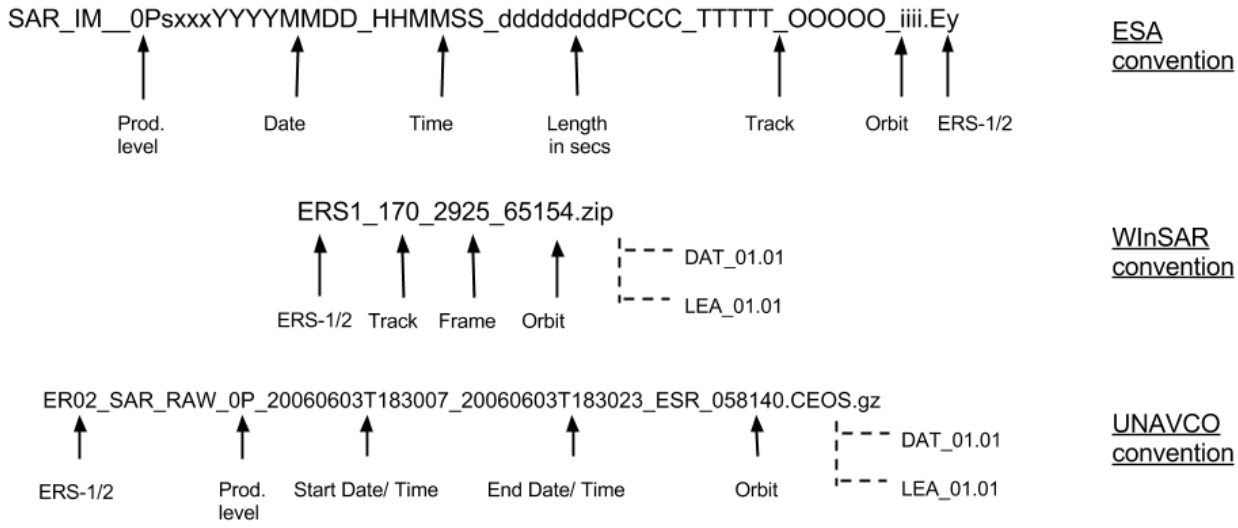
Contrast this to the X-band data over Hawaii.

CHAPTER 5

Processing ERS Data

1. Understanding ERS Data Set Names

The European Space Agency (ESA) launched and operated two nearly identical satellites call the European Remote Sensing (ERS) satellites, ERS-1 (1992–2000) and ERS-2 (1995-2011, although data acquired after 2000 has problems) (see also Table 1 of Lab 3.0). In this lab, we will learn how to process data from these satellites. Because the two satellites had the same radar characteristics and the same orbits for most of their operation, most of the data from the two satellites can be used interchangeably. The two satellites were operated in what ESA called a Tandem mission especially during 1995–1996 and 1998–1999 where the ERS-1 and ERS-2 satellites acquired data with a 1-day time separation over most of the Earth's land area.



The graphic above shows three standard naming conventions for ERS data files from three different processing and distribution centers in the world: The ESA archives, the Alaska Satellite Facility, and UNAVCO. Some ERS data is also available from the GEO Geohazards Supersites. The ESA "Envisat-style" file format (top line) combines the data with the metadata into a single file, similar to the format for Envisat data (see Lab 5). ASF and UNAVCO supply compressed files that contain two files each, the data, and metadata (so-called "leader" file), with different names for the container file. The data files inside have very simple and generic names as show above.

2. How to insert ERS filenames into the ISCE xml input files

The basic ISCE input file is similar to that for ALOS PALSAR. If you have not gone through the tutorial for running ISCE with ALOS, you should at least read through Step 2 of that tutorial, if not also setting up the input file and processing the ALOS data used in Step 3 of that tutorial. In this tutorial we assume you have read Step 2 of the ALOS tutorial and will only talk about the differences for processing ERS-1 and ERS-2 data.

As in the ALOS input files the ERS input files contain the `MASTER` and `SLAVE` component tags that contain the property tags `IMAGEFILE`, `LEADERFILE`, and `OUTPUT`. Different from the ALOS input files is two new property tags (the orbit type and the path to the directory containing the orbit files) that contain information about the orbit data for the data take. The ALOS `LEADERFILE` contained sufficiently accurate orbit data for InSAR processing. The ERS-1 orbit data in the `LEADERFILE`, however, are generally not adequate for accurate processing of InSAR data. The Delft Institute for Earth-oriented Space Research (DEOS) provides precise orbits for the entire ERS-1 mission (and most of the ERS-2 mission). We have installed these orbits in the directory `/home/ubuntu/data/orbits/ERS/ODR/ERS1`.

If you use the `ls` command to list the contents of that directory you will see over 500 files named `ODR.nnn`. These files contain the orbital data records (time and position) for several day long "arcs" of the orbit. The number `nnn` in the name of the ODR file indicates the arc number. In that directory you will find the key to determining which arc is needed for your data take in the `arclist` file,

```
> ls /home/ubuntu/data/orbits/ERS/ODR/ERS1/arclist
```

The `arclist` file lists the start and end time for each of the ODR orbital data record arc files. You can use the `cat` command to list the more than 500 lines of the arclist file or you can use the `more` command to see just one page full of the contents of the arclist file, just to get a taste of the contents of that file (page through the file using the "spacebar" and terminate the output by typing "q"),

```
> more /home/ubuntu/data/orbits/ERS/ODR/ERS1/arclist
```

You will never need to read the `arclist` file to find the correct ODR arc file to use. The ISCE processor does that for you. You only need to tell ISCE the type of orbit to use in the processing and the name of the directory containing the `arclist` and ODR files. The way to tell ISCE this information is to insert the following `ORBIT_TYPE` and `ORBIT_DIRECTORY` tags in *both* the `Master` and the `Slave component`:

```
<property name="ORBIT_TYPE">
    <value>ODR</value>
</property>

<property name="ORBIT_DIRECTORY">
    <value>/home/ubuntu/data/orbits/ERS/ODR/ERS1</value>
</property>
```

3. Processing ERS data with ISCE

You should change your directory to the lab4 directory for ERS processing. You can position yourself in that directory from wherever you might be positioned currently as follows:

```
> cd
> cd data/lab4/ers
```

There are two directories there for two different data acquisition dates:

```
> ls
19950421  19971227
```

Within those directories you will see 'tar.gz' files, which are compressed containers of several files. The names of these 'tar.gz' files follow the UNAVCO name convention described at the start of this lab. To unpack these files use the tar command as follows,

```
> cd 19950421
> ls
ER01_SAR_IM__0P_19950421T183128_19950421T183145_DPA_19697_0000.CEOS.tar.gz
> tar -xzvf ER01_SAR_IM__0P_19950421T183128_19950421T183145_DPA_19697_0000.CEOS.tar.gz
> ls
DAT_01.001
ER01_SAR_IM__0P_19950421T183128_19950421T183145_DPA_19697_0000.CEOS.tar.gz
LEA_01.001
NUL_DAT.001
SAR_IM__0PXDLR19950421_183128_00000017G145_00170_19697_7041.E1.ps
VDF_DAT.001
```

The relevant files as inputs to ISCE are the file `LEA_01.001`, which is the leader file, and the file `DAT_01.001`, which is the image file.

Similarly for the `19971227` directory:

```
> cd ../19971227
> ls
ER02_SAR_IM__0P_19971227T183128_19971227T183145_DPA_14052_0000.CEOS.tar.gz
> tar -xzvf ER02_SAR_IM__0P_19971227T183128_19971227T183145_DPA_14052_0000.CEOS.tar.gz
> ls
DAT_01.001
ER02_SAR_IM__0P_19971227T183128_19971227T183145_DPA_14052_0000.CEOS.tar.gz
LEA_01.001
```

```
NUL_DAT.001
SAR_IM__0PXDLR19971227_183128_00000017A028_00170_14052_7031.E2.ps
VDF_DAT.001
```

Note that the `tar.gz` file in the 1995 directory starts with `ER01` indicating that it was from the ERS-1 instrument and the `tar.gz` file in the 1997 directory starts with `ER02` indicating that it was from the ERS-2 instrument. We will process these data sets from these two different compatible instruments using `insarApp.py`. In each directory there is an IMAGERY file named `DAT_01.001` and a LEADER file named `LEA_01.001` corresponding to two different frames with different acquisition times encoded in the associated `tar.gz` filenames. Because the actual file names for each date are the same, it is essential that the data for each date be in separate directories.

To prepare for processing the ERS data, first make a new processing directory in the `data/lab4/ers` directory (first making sure that you are in the proper directory using the `pwd` command or use the `cd` command once without an argument to first move to the top directory and then with the argument `data/lab4/ers` to move into the proper directory),

```
> cd
> cd data/lab4/ers
> pwd
/home/ubuntu/data/lab4/ers

> mkdir 19950421_19971227
> cd 19950421_19971227
> pwd
/home/ubuntu/data/lab4/ers/19950421_19971227
```

At this time you should be able to copy your input files (whether you used the all-in-one or the multiple input files styles) from the `lab3/alos/20070612_20090802` directory into the current `lab4/ers/19950421_19971227` directory,

```
> cp ../../../lab3/alos/20070612_20090802/insar_input.xml .
```

Notice the "`.`" at the end of that command line. It is shorthand for the "current directory". This command copies the file `insar_input.xml` at the path

```
../../../lab3/alos/20070612_20090802/insar_input.xml
```

into the current directory giving it the same name `insar_input.xml`. Note that the name of this file is not significant as it is entered as the first (usually only) xml file on the command line. You may call it anything you like, such as `insar_19950421_19971227.xml` or `insarApp.xml`.

If you used the all-in-one style, then this is the only file you need to copy. If you used multiple files, then you will need to copy the "Master" and "Slave" xml files also. Those files can have any name you chose. Remember, all that matters is that the file names you use are also the names entered in the catalog tags in the Master and Slave component tags in the insar_input.xml file.

If you followed the names in the text of the ALOS Datasets lab, then you probably used Master.xml and Slave.xml, and you can use the cp command as above with the "." at the end of the command to copy those files from the alos directory to the current directory using the same names.

If you followed the "date" naming convention given in the ALOS examples link, however, then you may have called them 20070612.xml and 20090802.xml. When you copy those files to the current directory, then you should change the date on the filename. For the Master you would use the command,

```
> cp ../../../lab3/alos/20070612_20090802/20070612.xml 19950421.xml
```

The second argument on this version of the cp command says to copy the contents of the file 20070612.xml in directory ../../alos/20070612_20090802 into a file named 920424.xml in the current directory.

SImilarly for the Slave,

```
> cp ../../../lab3/alos/20070612_20090802/20090802.xml 19971227.xml
```

Now use your favorite editor to enter the paths and names of this lab's ERS files into the IMAGEFILE, LEADERFILE, and OUTPUT tags in your input files. Then you should be able to copy or type in the ORBIT_TYPE and ORBIT_DIRECTORY tags given in Step 2 above.

Once you have inserted these tags into *both* your Master and Slave component tags, then you should be ready to process the data. At this point you can either go ahead and try to run insarApp.py and let the computer tell you if you have prepared the input file(s) correctly (if the processing runs to the end in about 20 minutes) or incorrectly (if the processing terminates early with a computer generated "traceback" indicating the location of the error in the code),

```
> insarApp.py insar_input.xml
```

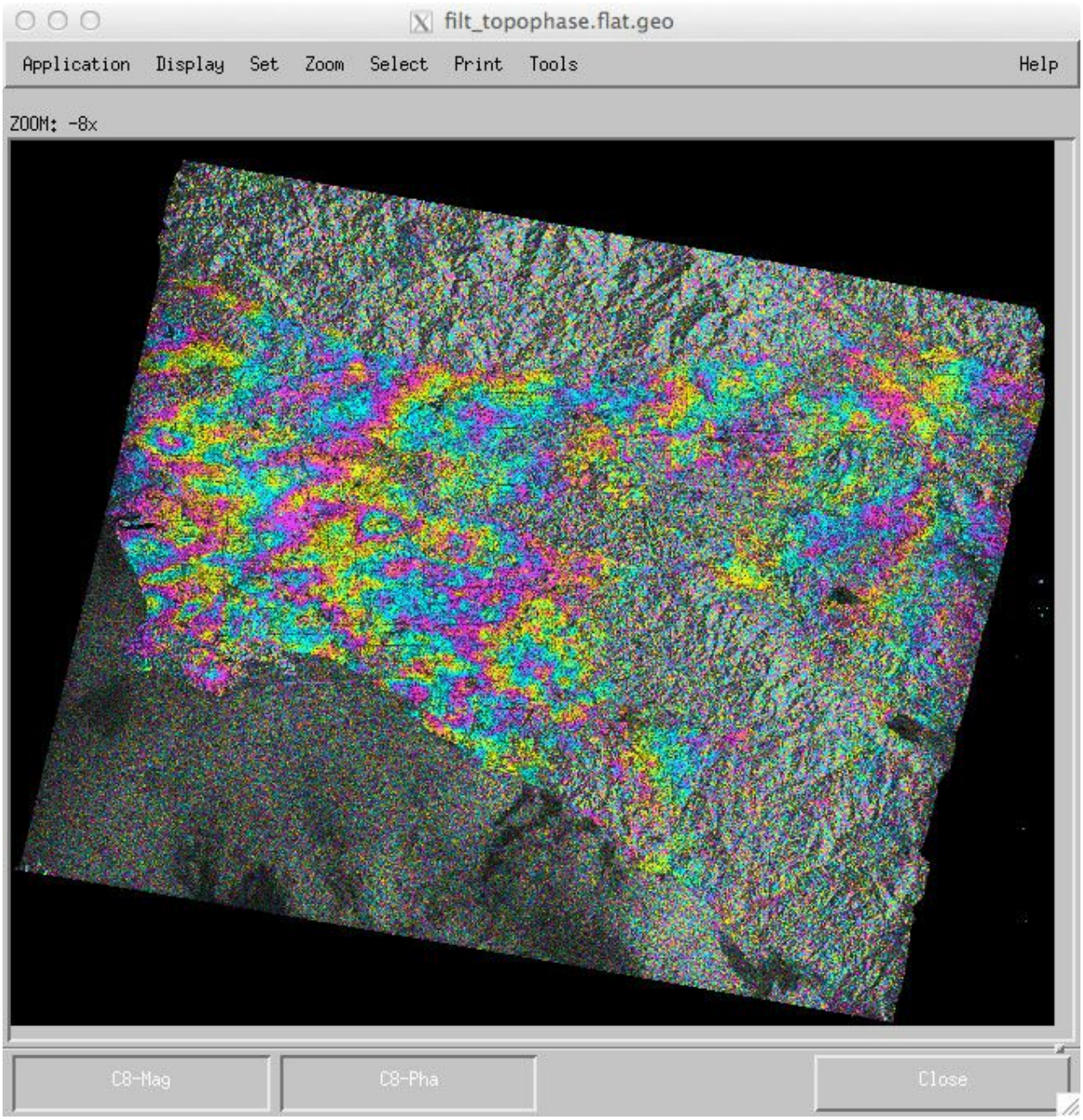
where insar_input.xml is the file you have prepared using either the all-in-one file containing the complete Master and Slave component tags or the multiple input files where the insar_input.py file contains catalog tags that point to two other xml files containing the information for the Master and Slave components.

If you choose to be cautious or if you attempted to process and it did not run to the end successfully and you cannot figure out what is wrong with your input files, then you can compare your input file(s) with these examples.

4. Your completed run

Now we can look at the final geocoded, wrapped interferogram with MDX:

```
> mdx.py filt_topophase.flat.geo &
```


filt_topophase.flat.geo
Application
Display
Set
Zoom
Select
Print
Tools
Help
ZOOM: -8x
C8-Mag
C8-Pha
Close

1. ERS-1 xml input file examples

The contents of `insar_input.xml` for the multiple input files style should look like the following (where the names of the files, `19950421.xml` and `19971227.xml`, are most likely different from the names you may have chosen; the names don't matter as long as they are correctly identifying the files containing the data indicated below):

```
<insarApp>
<component name="insarApp">
      <property name="sensor name">
          <value>ERS</value>
      </property>
      <component name="Master">
          <catalog>19950421.xml</catalog>
      </component>
      <component name="Slave">
          <catalog>19971227.xml</catalog>
      </component>
</component>
</insarApp>
```

The contents of `19950421.xml` should look like:

```
<component name="Master">
    <property name="IMAGEFILE">
        <value>../19950421/DAT_01.001</value>
    </property>
    <property name="LEADERFILE">
        <value>../19950421/LEA_01.001</value>
    </property>
    <property name="ORBIT_TYPE">
        <value>"ODR"</value>
    </property>
    <property name="ORBIT_DIRECTORY">
        <value>/home/ubuntu/data/orbits/ERS/ODR/ERS1</value>
    </property>
    <property name="OUTPUT">
        <value>"master.raw"</value>
    </property>
</component>
```

The contents of `19971227.xml` should look like:

```
<component name="Slave">
    <property name="IMAGEFILE">
        <value>../19971227/DAT_01.001</value>
    </property>
    <property name="LEADERFILE">
        <value>../19971227/LEA_01.001</value>
    </property>
    <property name="ORBIT_TYPE">
        <value>"ODR"</value>
    </property>
    <property name="ORBIT_DIRECTORY">
        <value>/home/ubuntu/data/orbits/ERS/ODR/ERS2</value>
    </property>
    <property name="OUTPUT">
        <value>"slave.raw"</value>
    </property>
</component>
```

The contents of the all-in-one style of input file `insar_allinput.xml` should look like the following:

```
<insarApp>
<component name="insarApp">
    <property name="sensor name">
         <value>ERS</value>
    </property>
    <component name="Master">
        <property name="IMAGEFILE">
            <value>../19950421/DAT_01.001</value>
        </property>
        <property name="LEADERFILE">
            <value>../19950421/LEA_01.001</value>
        </property>
        <property name="ORBIT_TYPE">
            <value>"ODR"</value>
        </property>
        <property name="ORBIT_DIRECTORY">
            <value>/home/ubuntu/data/orbits/ERS/ODR/ERS1</value>
        </property>
        <property name="OUTPUT">
```

```
            <value>"master.raw"</value>
        </property>
    </component>
    <component name="Slave">
        <property name="IMAGEFILE">
            <value>../19971227/DAT_01.001</value>
        </property>
        <property name="LEADERFILE">
            <value>../19971227/LEA_01.001</value>
        </property>
        <property name="ORBIT_TYPE">
            <value>"ODR"</value>
        </property>
        <property name="ORBIT_DIRECTORY">
            <value>/home/ubuntu/data/orbits/ERS/ODR/ERS2</value>
        </property>
        <property name="OUTPUT">
            <value>"slave.raw"</value>
        </property>
    </component>
</insarApp>
```

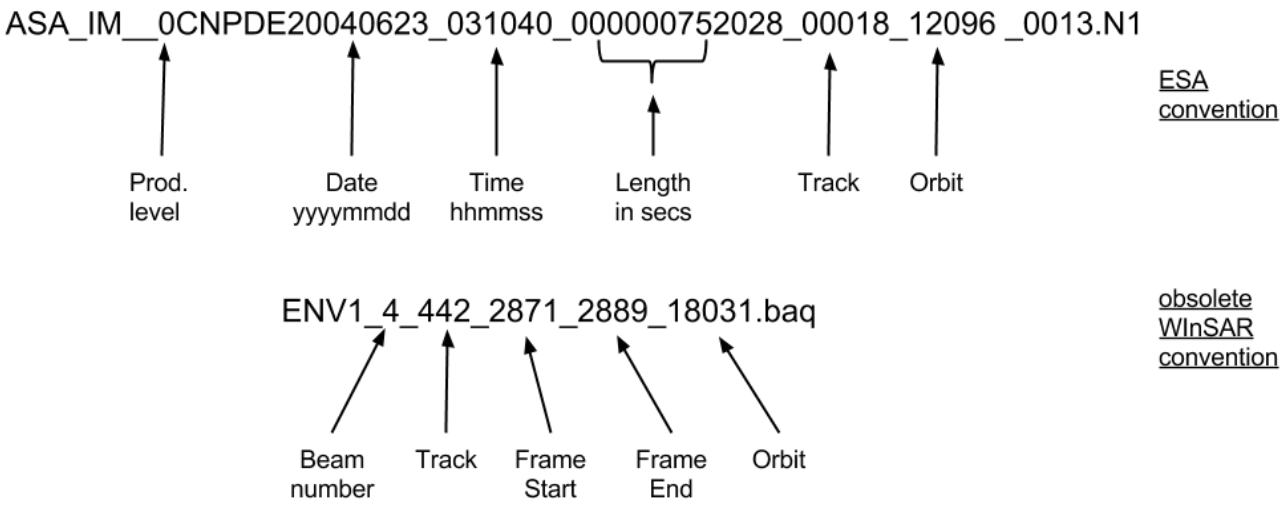
CHAPTER 6

Processing Envisat Data

1. Understanding Envisat Data Set Names

In this tutorial we will process an Envisat dataset covering the 2010 M7.2 Baja California, Mexico earthquake (official name El Mayor-Cucapah earthquake).

We will learn how to process SAR data from the European Space Agency's Envisat satellite (actually called advanced SAR or ASAR). Envisat SAR data are typically obtained either directly from ESA, UNAVCO's WInSAR archive or through the GEO Geohazards Supersites archive. The files may be named differently depending on the source.



Some of the files in the WInSAR archive in the past used a different naming convention from the standard ESA naming convention, as shown in the diagram above. All of the Envisat data in the WInSAR archive at UNAVCO is now converted back to the original ESA names, but files downloaded some years ago might have the special ".baq" name. In both cases, the file content is the same, only the name is different.

Envisat SAR data files have both metadata and binary data in the same file. The metadata at the beginning is ASCII text and can be viewed in your terminal, but the binary data later in the file is not directly viewable (and viewing it may confuse your terminal). ISCE presently supports only processing the raw (Level 0) data from Envisat ASAR acquired in imaging or stripmap mode, so make sure the filename starts with ASA_IM__0 if it has the standard ESA name.

2. How to insert Envisat filenames into the ISCE xml input files

The basic ISCE input file for Envisat is similar to that for ALOS PALSAR. If you have not gone through the tutorial Lab 3.1 for running ISCE with ALOS, you should at least read through Step 2 of that tutorial, if not also setting up the input file and processing the ALOS data used in Step 3 of that tutorial. In this tutorial we assume you have read Step 2 of the ALOS tutorial and will only talk about the differences for processing Envisat data.

As in the ALOS and ERS input files the Envisat input files contain the `MASTER` and `SLAVE` component tags that contain the property tags `IMAGEFILE` and `OUTPUT`, but for Envisat there is no `LEADERFILE` tag because the leader information is included in the image file. Different from the ALOS and ERS input files are two new property tags (the ancillary orbit and instrument files) that contain information about the orbit data for the data take and SAR instrument calibration. Both of these properties are required for Envisat data processing.

The way to tell ISCE the orbit and instrument file information is to insert the following `ORBITFILE` and `INSTRUMENTFILE` tags (order is not important) in *both* the `Master` and the `Slave` components (the orbit files will be different for each date and INS files may also differ so we have to specify these for both input scenes):

```
<property name="INSTRUMENTFILE">
     <value>"/home/ubuntu/data/instruments/ENVISAT/ASA_INS_AXVI
     EC20091217_114637_20090428_100000_20101231_235959"</value>
</property>

<property name="ORBITFILE">
     <value>"/home/ubuntu/data/orbits/ENVISAT/VOR/DOR_VOR_AXVF-
     P20100423_084900_20100327_215526_20100329_002326"</value>
</property>
```

The Envisat data files do not contain the orbit information. You have to download the orbit files separately. We will use the DORIS orbit files (DORIS is the name of the method they used to determine the Envisat orbits) that can be downloaded from ESA, which requires registration with ESA. The location of the link to the orbit data (at the moment) is https://earth.esa.int/web/guest/data-access/browse-data-products/-/article/doris-precise-orbit-state-vectors-1502. If this link does not work, then search for “Envisat DORIS orbits” in your favorite search tool. The Delft Institute for Earth-oriented Space Research (DEOS) also provides precise orbits for the early part of the Envisat mission, but they stopped calculating the orbits circa 2007, so they are not so useful. We have installed the necessary Envisat DORIS orbits in the directory `/home/ubuntu/data/orbits/ENVISAT/VOR` The Envisat DORIS orbit file names have three dates and times in their file names, with the first date and time showing the time the file was produced, the second date and time telling the start of the time period covered

by the file, and the third date and time telling the end of the period, e.g., `DOR_VOR_AXVF-P20100423_084900_20100327_215526_20100329_002326` was produced 2010/04/23 and covers the time interval from 2010/03/27 21:55:26 to 2010/03/29 00:23:26 or basically the entire day 2010/03/28 plus two hours the day before and 23 minutes the day after. The `DOR` means it is a DORIS orbit file and `VOR` means that it is the final verified orbit. At present, you have to find the correct orbit data file and enter the name in the ISCE input files, unlike the automated search implemented for ERS orbits. Automated search should be added to ISCE later.

You also need the Envisat SAR instrument calibration file or INS file to read the Envisat data. They updated the INS files at several times during the Envisat mission, so there are different files for different time periods. The INS files can be downloaded from ESA's Envisat ASAR auxiliary data directory (http://earth.eo.esa.int/services/auxiliary_data/asar/current/) without registration. The INS file names also have three dates and times in their file names, with the first date and time showing the time the file was produced, the second date and time telling the start of the time period covered by the file, and the third date and time telling the end of the period (just like the orbit file names), e.g., `ASA_INS_AXVIEC20091217_114637_20090428_100000_20101231_235959` was produced 2009/12/17 and covers the time interval from 2009/04/28 to 2010/12/31. Sometimes ESA produced a new INS file before the planned end of a previous file, so you should use the newest file that covers the time interval of the data you have. Again, ISCE does not presently have the capability to search through a directory of INS files and select the correct one for the image files, so you need to manually select the file.

3. Processing Envisat data with ISCE

You should change your directory to the lab5 directory for Envisat processing. We will use the Envisat scenes that we downloaded from UNAVCO earlier for this lab. You can position yourself in that directory from wherever you might be positioned currently as follows:

```
> cd
> cd data/lab5/env
```

There should be two directories there for two different data acquisition dates, if you completed the download earlier:

```
> ls
20100328  20100502
```

Let's check the two data directories to make sure that the files are there. First go into the `20100502` directory:

```
> cd 20100502
> ls
ASA_IM__0CNPDE20100502_175001_000000172089_00084_42723_0354.N1
ASA_IM__0CNPDE20100502_175016_000000172089_00084_42723_0354.N1
```

There should be two data files as shown, because we downloaded two frames for this date. Make sure the files have `20100502` in the name so that the right dates are in this directory. As we saw earlier in this lab, the scenes have `_00084` in the name so they are from Envisat track (relative orbit) 84.

Now let's check the data directory for the other date `20100328`. We can go up and back down into that directory with this `cd` command:

```
> cd ../20100328/
> ls
ASA_IM__0CNPDE20100328_175004_000000162088_00084_42222_9504.N1
ASA_IM__0CNPDE20100328_175019_000000162088_00084_42222_9504.N1
```

Again, check to see if you have the two data files for the `20100328` date as shown. Unlike the ERS data, the Envisat data files do not require any unpacking before we can use them.

To prepare for processing the Envisat data, first make a new processing directory in the `data/lab5/env` directory (first making sure that you are in the proper directory using the `pwd` command or use the `cd` command once without an argument to first move to the top directory and then with the argument `data/lab5/env` to move into the proper directory),

```
> cd
> cd data/lab5/env
> pwd
/home/ubuntu/data/lab5/env

> mkdir 20100502_20100328
> cd 20100502_20100328
> pwd
/home/ubuntu/data/lab5/env/20100502_20100328
```

For Envisat data, you can either use the separate files method, with the input information about each date (master and slave components) in a separate file, or the all-in-one method. Here we will explain the separate file method that is a little easier to read (in my opinion).

First, create the input component file for the 20100502 date:

```
> nano 20100502.xml
```

Then enter the IMAGEFILE, INSTRUMENTFILE, ORBITFILE, and OUTPUT information about this date as described above. For this processing, we will use a list of input images (file names with paths in quotes, inside square brackets) to process the two frames from each date together:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>

<component name="Envi">
    <property name="IMAGEFILE">

<value>['../20100502/ASA_IM__0CNPDE20100502_175001_000000172089_00084
_42723_0354.N1','../20100502/ASA_IM__0CNPDE20100502_175016_0000001720
89_00084_42723_0354.N1']</value> <!-- image files -->
    </property>
    <property name="INSTRUMENTFILE">

<value>"/home/ubuntu/data/instruments/ENVISAT/ASA_INS_AXVIEC20091217_
114637_20090428_100000_20101231_235959"</value> <!-- instrument file
-->
    </property>
    <property name="ORBITFILE">

<value>"/home/ubuntu/data/orbits/ENVISAT/VOR/DOR_VOR_AXVF-P20100604_1
04000_20100501_215526_20100503_002326"</value> <!-- orbitfile -->
```

```
    </property>
    <property name="OUTPUT">
        <value>"20100502.raw"</value> <!-- output raw file -->
    </property>
</component>
```

Let's check that the paths for some of the files specified in the XML work on your system (watch out for extra breaks in the long lines):

```
> ls
../20100502/ASA_IM__0CNPDE20100502_175001_000000172089_00084_42723_03
54.N1
../20100502/ASA_IM__0CNPDE20100502_175001_000000172089_00084_42723_03
54.N1
> ls
/home/ubuntu/data/instruments/ENVISAT/ASA_INS_AXVIEC20091217_114637_2
0090428_100000_20101231_235959
/home/ubuntu/data/instruments/ENVISAT/ASA_INS_AXVIEC20091217_114637_2
0090428_100000_20101231_235959
```

Now create the input component file for the `20100328` date, and again enter all the required information for this component:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>

<component name="Envi">
    <property name="IMAGEFILE">

<value>['../20100328/ASA_IM__0CNPDE20100328_175004_000000162088_00084
_42222_9504.N1','../201
00328/ASA_IM__0CNPDE20100328_175019_000000162088_00084_42222_9504.N1'
]</value> <!-- image files -->
    </property>
    <property name="INSTRUMENTFILE">

<value>"/home/ubuntu/data/instruments/ENVISAT/ASA_INS_AXVIEC20091217_
114637_20090428_100000_
20101231_235959"</value> <!-- instrument file -->
    </property>
    <property name="ORBITFILE">

<value>"/home/ubuntu/data/orbits/ENVISAT/VOR/DOR_VOR_AXVF-P20100423_0
84900_20100327_215526_2
```

```
0100329_002326"</value> <!-- orbitfile -->
    </property>
    <property name="OUTPUT">
        <value>"20100328.raw"</value> <!-- output raw file -->
    </property>
</component>
```

Now we need to create the main `insarApp.xml` file. We will use the `20100502` scene as the master scene, and we will add some other properties here. The first one is to set the `posting` or spacing of pixels in the interferogram, so that insarApp will adjust the number of looks or SLC pixels averaged in the interferogram. We will also request phase unwrapping and use the "`icu`" unwrapping program.

```
<?xml version="1.0" encoding="UTF-8"?>
<insarApp>
<component name="insar">
<!-- Posting is automatically calculated if not specified here.
-->
        <property name="Posting">
            <value>40</value>
        </property>
    <property name="unwrap">
        <value>True</value>
    </property>
        <property name="unwrapper name">
            <value>icu</value>
        </property>
        <property name="Sensor Name">
            <value>Envisat</value>
        </property>
        <property name="Doppler Method">
            <value>useDOPIQ</value>
        </property>
        <component name="Master">
            <catalog>20100502.xml</catalog>
        </component>
        <component name="Slave">
            <catalog>20100328.xml</catalog>
        </component>
</component>
</insarApp>
```

Now we are ready to run the processing, using the steps option, in case we want to re-run steps later. You may want to take a coffee or water break.

```
> insarApp.py insarApp.xml --steps
```

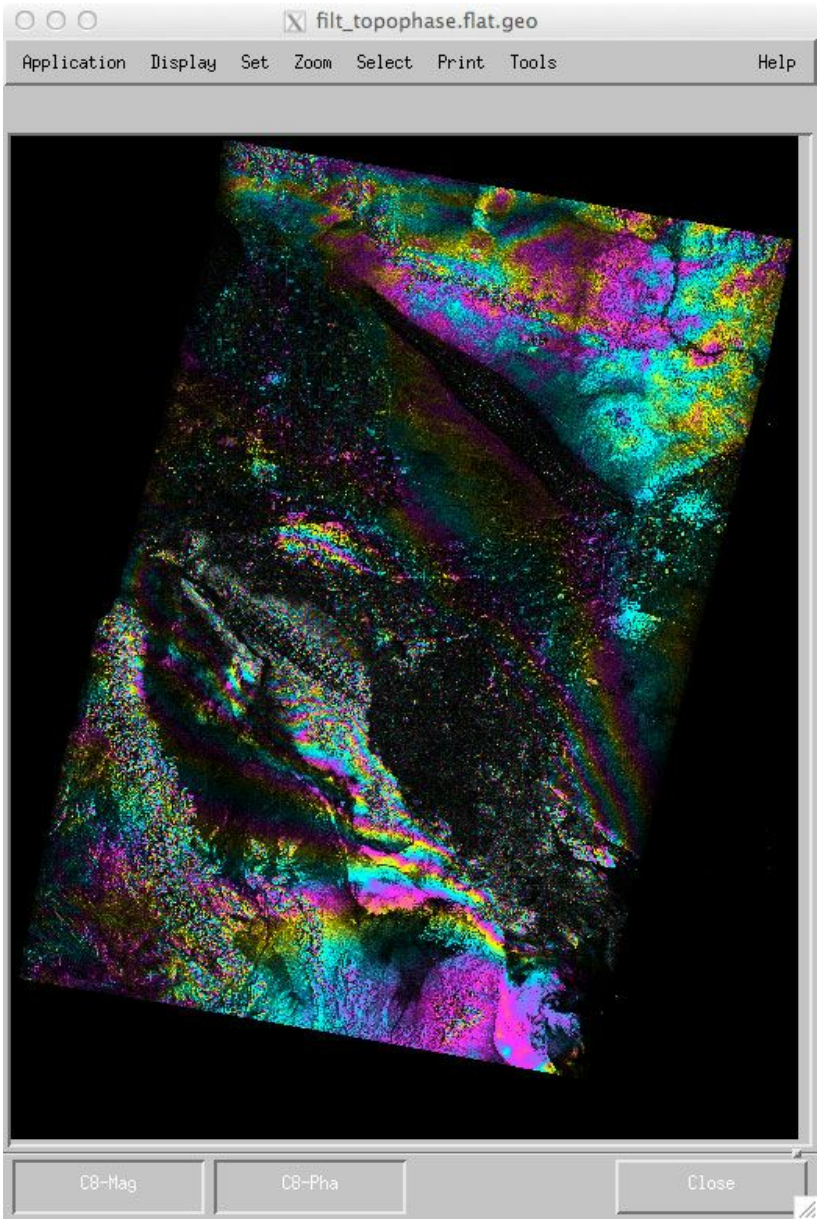
4. Your completed run

At the end of the processing, you should see something like this:

```
################################################################################
################################
2014-07-30 17:29:18,420 - isce.insar.runGeocode - INFO -
################################################################################
################################
    runGeocode - Outputs
--------------------------------------------------------------------------------
------------------------------
runGeocode.outputs.LONGITUDE_SPACING = 0.0008333333333333334
runGeocode.outputs.MINIMUM_GEO_LONGITUDE = -116.09083333333334
runGeocode.outputs.LATITUDE_SPACING = -0.0008333333333333334
runGeocode.outputs.MAXIMUM_GEO_LATITUDE = 31.355000000000004
runGeocode.outputs.MAXIMUM_GEO_LONGITUDE = -114.4175
runGeocode.outputs.GEO_LENGTH = 2560
runGeocode.outputs.MINIMUM_GEO_LATITUDE = 33.487500000000004
runGeocode.outputs.GEO_WIDTH = 2009
################################################################################
################################
2014-07-30 17:29:18,420 - isce.insar - INFO - Total Time: 807 seconds
```

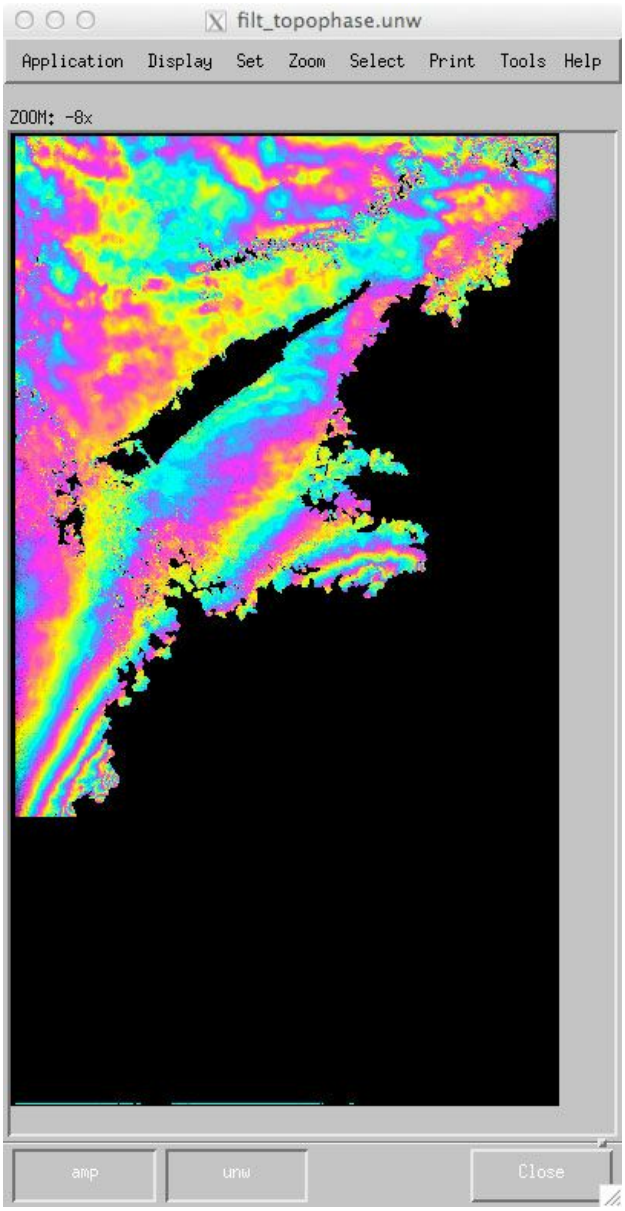
Note that ISCE has downloaded the SRTM 3-arcsecond DEMs, and the final geocoding has this spacing (`LONGITUDE_SPACING = 0.0008333333333333334`), because part of the area is outside the USA so the 1-arcsecond SRTM data is not publicly available (yet). Let's have a look at the geocoded wrapped interferogram (it is large, so we tell MDX to zoom out by 4 to start):

```
> mdx.py filt_topophase.flat.geo -z -4 &
```



Now we can see a lot of fringes! This is the deformation from the M7.2 earthquake. Let's look at the unwrapped phase now, in radar coordinates:

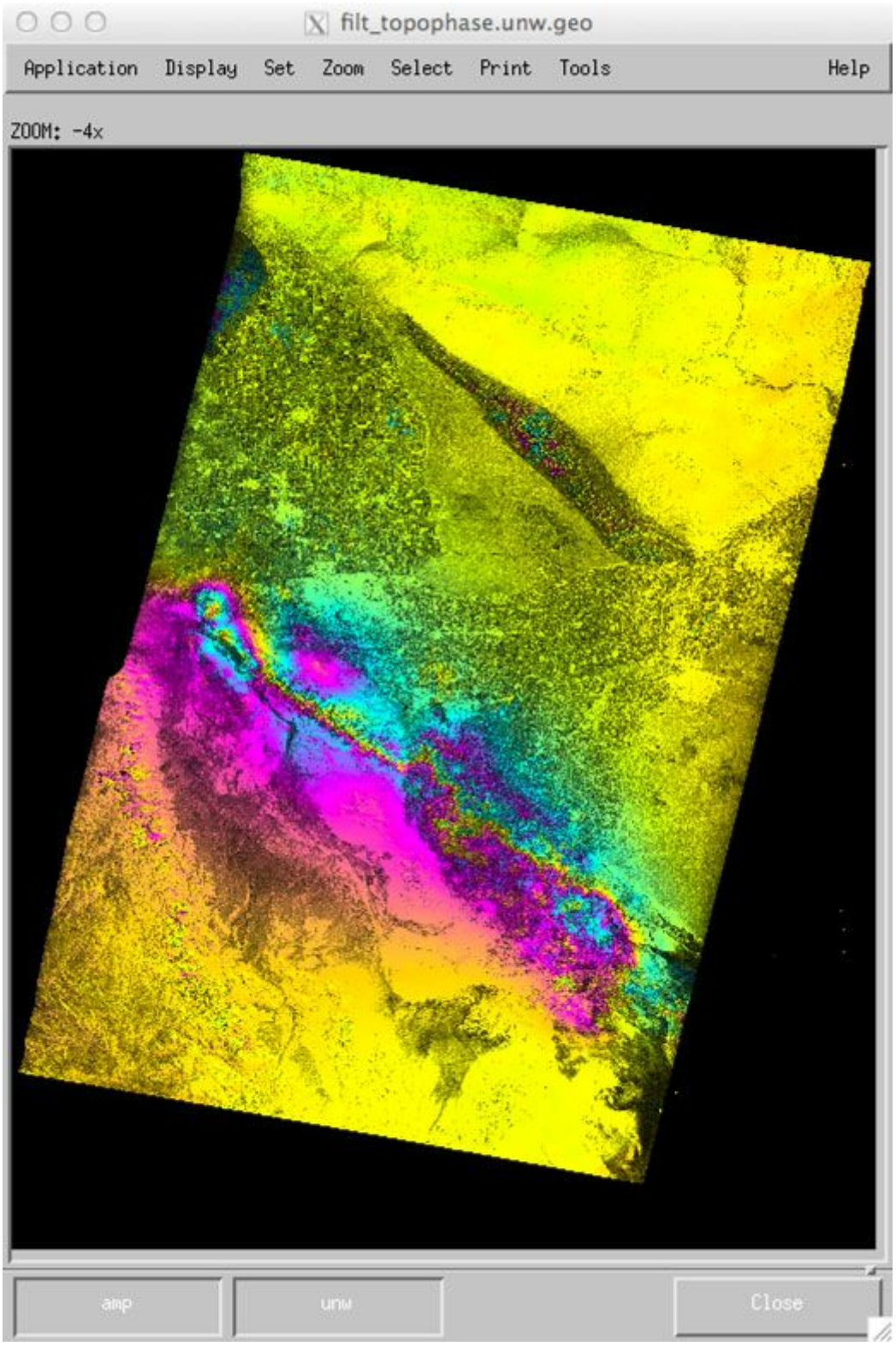
```
> mdx.py filt_topophase.unw -z -8 &
```



We can see that the “icu” unwrapper only managed to unwrap the top (north) side of the fault rupture, and it stopped unwrapping after the first patch (3700 lines of the multi-look interferogram). Let’s try a different unwrapping program called SNAPHU (Statistical-cost, Network-flow Algorithm for Phase Unwrapping) written by Curtis Chen when he was at Stanford (see Chen and Zebker, 2002). Edit your insarApp.xml file and change the “unwrapper name” property value from “`icu`” to “`snaphu`”. Since we ran the initial processing with “`--steps`”, we can just restart the processing at the `unwrap` step without having to rerun the earlier steps (we also don’t need to specify the `--steps` flag in addition to the `--start=unwrap`):

```
> insarApp.py insarApp.xml --start=unwrap
```

The SNAPHU program can take a while to run, depending on how large and noisy your interferogram is. Low coherence, noisy data take a long time to unwrap and “snaphu” unwraps everything, unlike the “grass” and “icu” unwrappers that mask out the noisy areas before unwrapping. When the unwrapping is complete, take a look at the `filt_topophase.unw` or `filt_topophase.unw.geo` file with `mdx.py`. Change the color wrap on the phase to 100 radians to see the large displacement from the earthquake like this (I also adjusted the exponent of the amplitude image to 0.5):



The 2010 M7.2 earthquake ruptured about 120 km of faults in northern Mexico. Rocks on the southwest side of the fault moved up to 2 meters northwest relative to the northeast side. See

this paper for more information:

Wei, S., Fielding, E. J., Leprince, S., Sladen, A., Avouac, J.-P., Helmberger, D. V., Hauksson, E., Chu, R., Simons, M., Hudnut, K. W., Herring, T., & Briggs, R. W. (2011). Superficial simplicity of the 2010 El Mayor–Cucapah earthquake of Baja California in Mexico. *Nature Geosci, 4*, 615-618. http://www.nature.com/ngeo/journal/v4/n9/full/ngeo1213.html

CHAPTER 7

Processing COSMO-SkyMed Raw Data

1. Processing COSMO-SkyMed data from raw data files

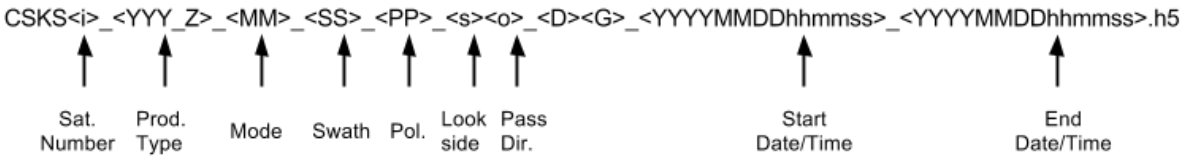
In this lab, you will learn how to process COSMO-SkyMed data from raw data files, while exploring some of the configurability capabilities of ISCE. As we've seen in previous labs, InsarApp.py is set up with sensor-dependent default parameters for all the processing steps. For many data sets, all the user needs to do is set up the input data file names in the master and slave XML files and update insarApp.py's control file (typically named insar.xml) to reference them. Then the command

```
insarApp.py insar.xml
```

will process the data from raw to geocoded interferograms. There are times however when the default parameters are not optimal. We saw a simple example of this in lab 3.3 where the default settings did not unwrap the interferogram. We could set the unwrap flag in the control file to cause unwrapping to occur.

In this lab we will demonstrate how to affect control parameters through configurability files associated with individual processing components. The README file at the top level of the ISCE distribution describes the details of the configurability options and hierarchy of where ISCE looks to find parameters it needs to process. Here we will first process a patch of data to completion to allow us to see just the beginning of the file. We will examine the outputs and decide that we want to change a particular processing option, create a configuration file for the appropriate component, and reprocess with that new option. We will then look at the output again to see how things have changed.

2. Understanding CSK Data Set Names



The graphic above shows the standard naming conventions for COSMO-SkyMed (CSK) data files. CSK data comes with all of the data and metadata, including the orbit, in a single HDF5 (.h5) file. Sometimes the data is delivered with some additional files, but these are not used in the ISCE processing.

3. How to insert CSK filenames into the ISCE xml input files

The file names can be inserted into a master and slave component through configuration files as shown previously:

```
> cd
> cd data
> cd lab6
> cat Master.xml
<component name="Master">
    <property name="HDF5">
      <value>data/CSKS4_RAW_B_HI_08_HH_RA_SF_20110327162502_201103271
62510.h5</value>
    </property>
    <property name="OUTPUT">
        <value>20130327.raw</value>
    </property>
</component>

> cat Slave.xml
<component name="Slave">
    <property name="HDF5">
           <value>data/CSKS4_RAW_B_HI_08_HH_RA_FF_20110311162513_2011
      0311162521.h5</value>
    </property>
    <property name="OUTPUT">
        <value>20061231.raw</value>
    </property>
</component>
```

Note that in this case, the `h5` data files reside in a subdirectory called `data`, so the filename is prepended with the relative path `data/`. These lines can also be directly inserted into the insarApp input file. For this lab, the input file is named `insar_130327_130311.xml`. If you print that file to the screen, you will see these lines verbatim inline in the file.

> `cat insar_130327_130311.xml`

(shows the same as above text in Master.xml and Slave.xml)

4. Running a patch of data using component configurability and examining the results

Let's start with an input file that has some particular values set : `insar_130327_130311.xml`. In this input file, the only specified parameters are the posting of the output grid and the master and slave data files, and of course the sensor name. All other parameters are set to their defaults that are deemed appropriate for this sensor.

```
> cat insar_130327_130311.xml
<insarApp>
    <component name="insar">
        <property name="posting">
            <value>20</value>
        </property>
        <property name="Sensor Name">
            <value>COSMO_SKYMED</value>
        </property>
        <component name="master">
            <property name="HDF5">
<value>data/CSKS4_RAW_B_HI_08_HH_RA_SF_20110327162502_20110327162510.
h5</value>
            </property>
            <property name="OUTPUT">
                <value>20130327.raw</value>
            </property>
        </component>
        <component name="slave">
            <property name="HDF5">
<value>data/CSKS4_RAW_B_HI_08_HH_RA_FF_20110311162513_20110311162521.
h5</value>
            </property>
            <property name="OUTPUT">
                <value>20130311.raw</value>
            </property>
        </component>
    </component>
</insarApp>
```

Sometimes it is convenient to process only the beginning portion of data set the first time through to get a feel for the data - examine its correlation and fringe quality, check the focus of the image, etc. There are several ways to accomplish this; we saw in Lab 3 that the number of patches can be set in the input file directly. But suppose we don't want to or do not have permission to alter the input file, or we want to apply the same parameter modifications to a number of interferogram processing runs. We can take advantage of component configurability to accomplish the same thing.

The ISCE architecture has mechanisms to allow each parameter in a workflow component in `insarApp.py` to be configurable. This feature was added recently so not all components are yet configurable, but certain key components are. The image formation component, known as `formslc`, is one such configurable component. `formslc` is a patch-based focusing system, performing convolution through FFT-based circular convolution one chunk of pulses at a time. This is the module where specifying a single patch for quick examination takes place. To set parameters specific to `formslc` in a configuration file, we create a file called `formslc.xml` and place it in the directory where we are running `insarApp.py`. In this case, formslc is the f*amily name* of the what would be a possible set of image formation modules. Thus the name *formslc* in `formslc.xml` refers to the family name. Let's set the parameter for the number of patches, called `NUMBER_PATCHES`, to 1 in `formslc.xml`:

```
> cat > formslc.xml
<dummy>
<component name="formslc">
    <property name="NUMBER_PATCHES">1</property>
</component>
</dummy>

Press Ctrl-D to exit cat
```

For configuration files, the bounding xml construct we've seen before in `insar.xml` is not needed, so we see above the use of `<dummy>`. This could be any string.

Now let's run one patch, with `formslc.xml` present in the current working directory.

```
> insarApp.py insar_130327_130311.xml --steps
```

After waiting a while, you will find that the run ends in an error at the end of a long traceback:

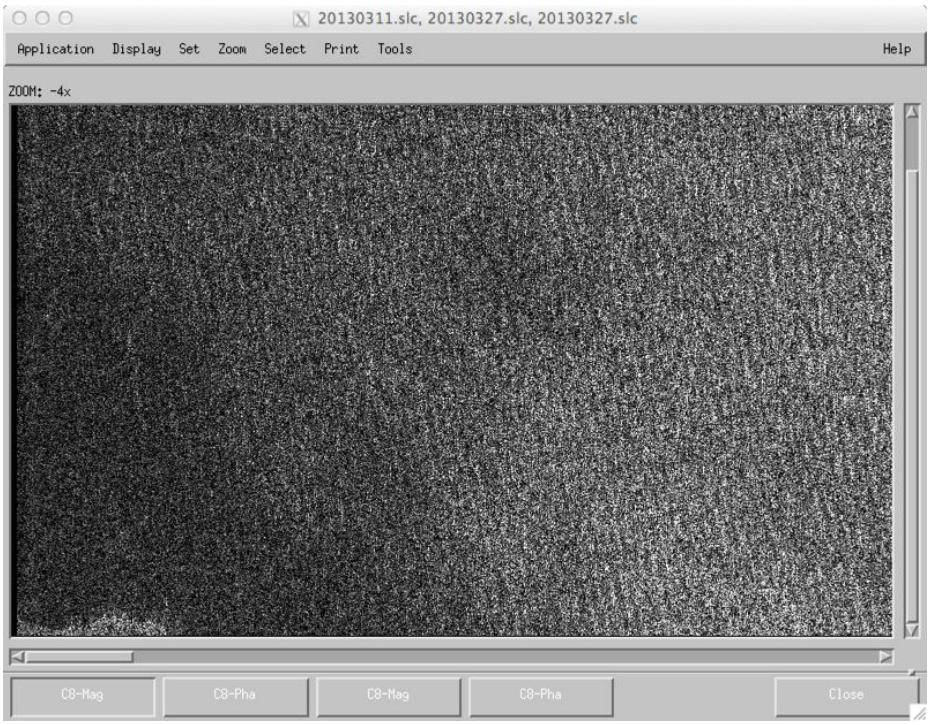
```
"/Users/parosen/Applications/Installs/isce_py33/isce/components/isceobj/Util/EstimateOffsets.p
y", line 347, in checkImageLimits
    raise ValueError('Too small a reference image in the height direction')
ValueError: Too small a reference image in the height direction
```

Is there some fundamental problem with the data set? Typically at this point, it is good to look

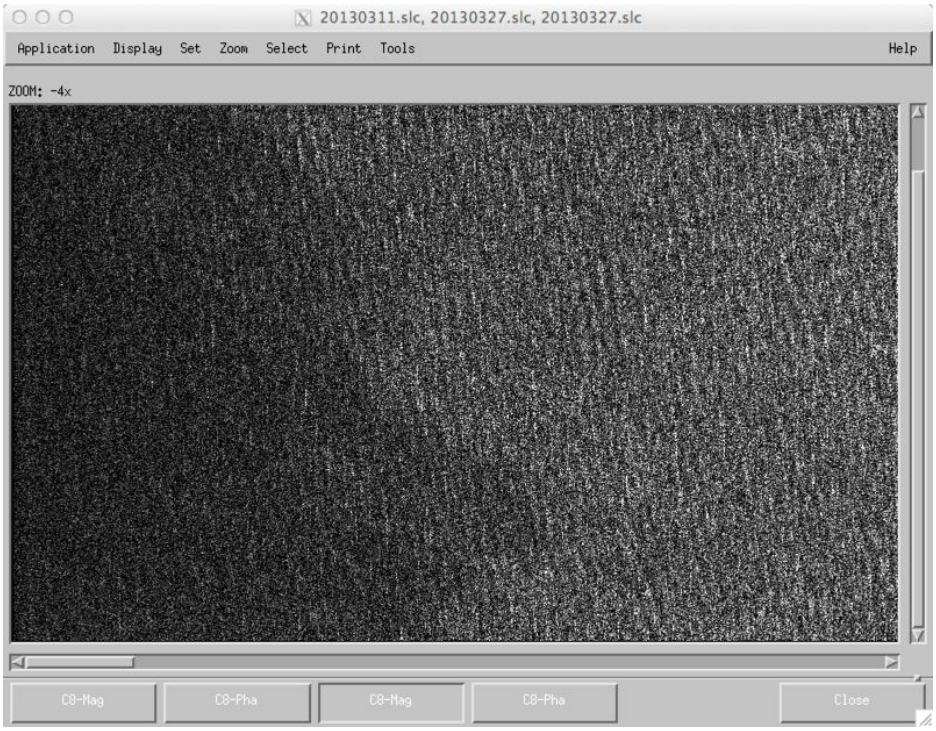
at the images (slcs) to see if there is an issue.

```
> mdx.py *.slc
```

shows something interesting. If you zoom out by a factor of 4, then scroll to the bottom left, then click on C8-Mag left most button, you see the amplitude of the master image, as follows.



If you do the same but click on the other C8-Mag button, you see the amplitude of the slave image.



These images look similarly non-descript except for a small feature in the bottom left corner of the master image. This is a piece of land from Hawaii, and the rest of the image is just water. So in this particular case, processing 1 patch was not particularly useful because most of the image is water in this area, and clearly the ability to estimate offsets, both between images and between interferogram and topography, is compromised with only a patch of ocean coverage. So we should really try more patches.

5. Running more patches of data using component configurability

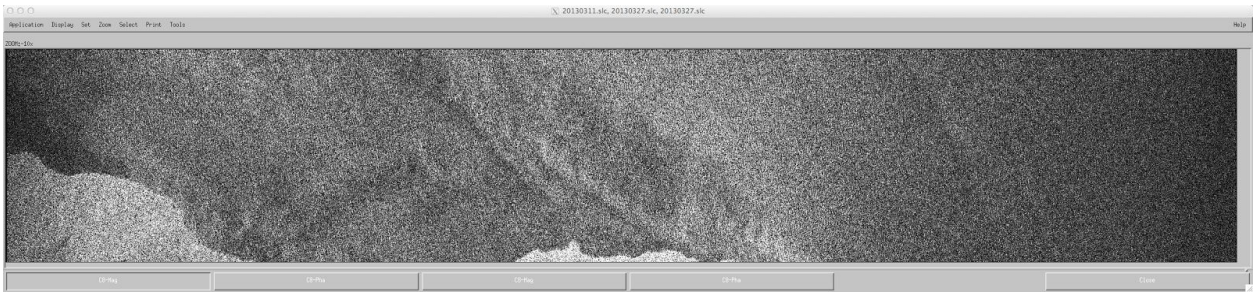
Let's just try 2. Modify `formslc.xml` to set `NUMBER_PATCHES` to 2 and run it again.

```
> insarApp.py insar_130327_130311.xml --start=formslc
.
.
.
  File
"/Users/parosen/Applications/Installs/isce_py33/isce/components/isceobj/InsarProc/runOffoutlie
rs.py", line 60, in runOffoutliers
    raise Exception('Offset estimation Failed.')
Exception: Offset estimation Failed.
```

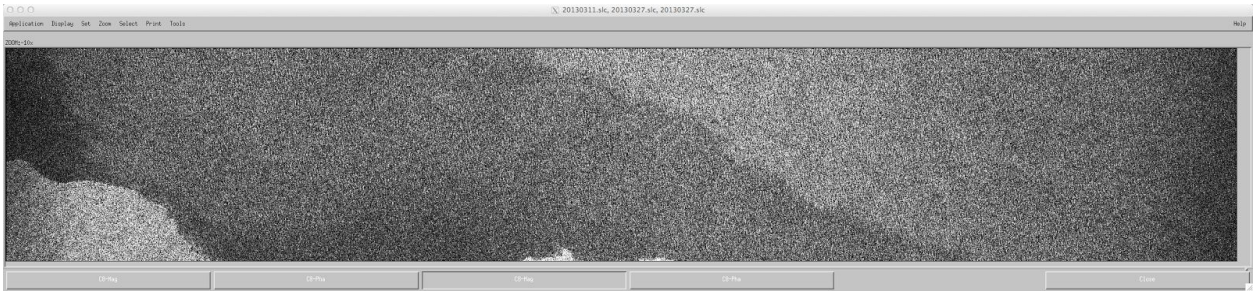
We've failed again, with a different error message! What could have gone wrong this time?

```
> mdx.py *.slc
```

This time, zoom out by a factor of 10 and stretch the window to be able to see the entire processed region. (If your screen is too small to do this, try zooming out even more until it fits.) The master looks as follows:



And the slave as so:



The bright region in the lower left is land, The other brightness features are water backscatter which varies from time to time and does not correlate. So clearly in this case, there are not enough points of common correlation over land to estimate the alignment of the data, so the offset estimation failed.

This is a piece of land from Hawaii, and the rest of the image is just water. So in this particular case, processing 1 patch was not particularly useful because most of the image is water in this area, and clearly the ability to estimate offsets, both between images and between interferogram and topography, is compromised with only a patch of ocean coverage. So we should really try more patches.

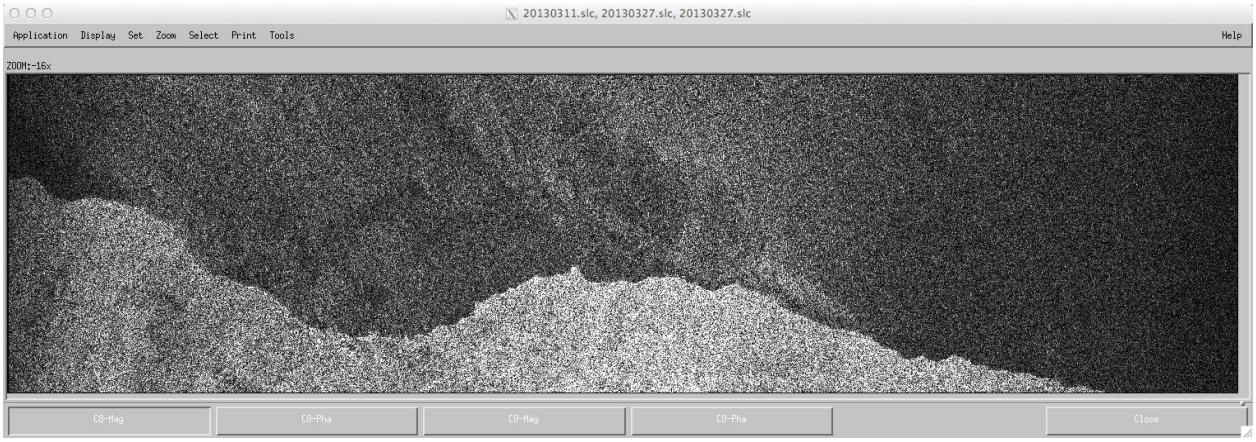
So how do we get this to work better? It looks like the data are good, but we are confounded by geography relative to our rectangular window into the world dictated by the radar imaging process. One option is to process more patches to get more land. Another is move the starting location to process the data. Another yet is to increase the size of each patch to cover more real estate with each step.

6. Running with a larger patch size using component configurability

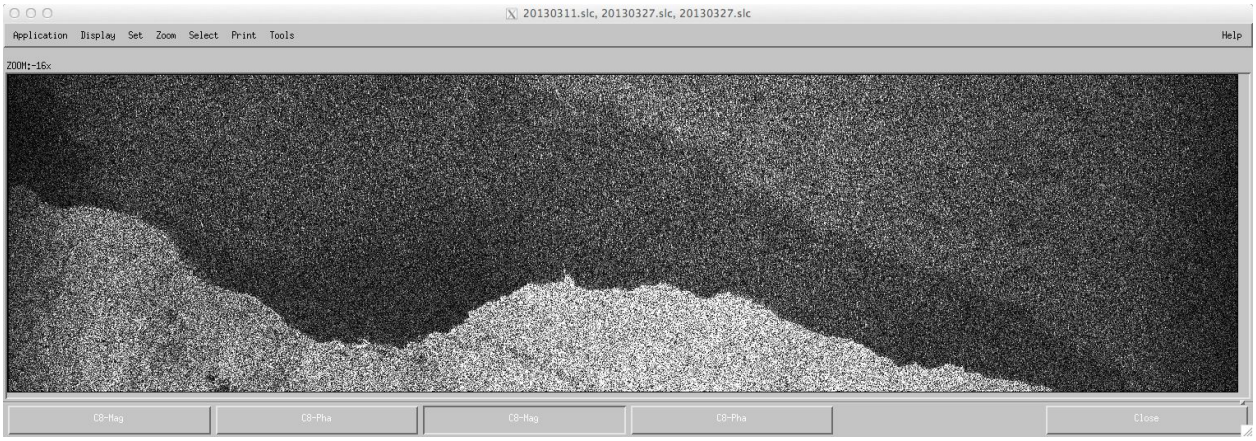
Let's try the last option, specifying 1 patch but a larger patch size. The default setting for patch size saves 2048 pulses (look at the mdx window and scroll to the bottom, then click on a pixel to see how many lines there are in the file, or look at the metadata for the slc. For 2 patches, the slc's are 4096 lines, so each patch is 2048 lines). Let's try a patch size of 8192 pulses `formslc.xml` should now look like:

```
<dummy>
<component name="formslc">
    <property name="NUMBER_PATCHES">1</property>
    <property name="AZIMUTH_PATCH_SIZE">8192</property>
</component>
</dummy>
```

We've increased the size of the patch by a factor of 4, so we should see more land. Indeed, looking at the slcs, we see for the master:



and for the slave:

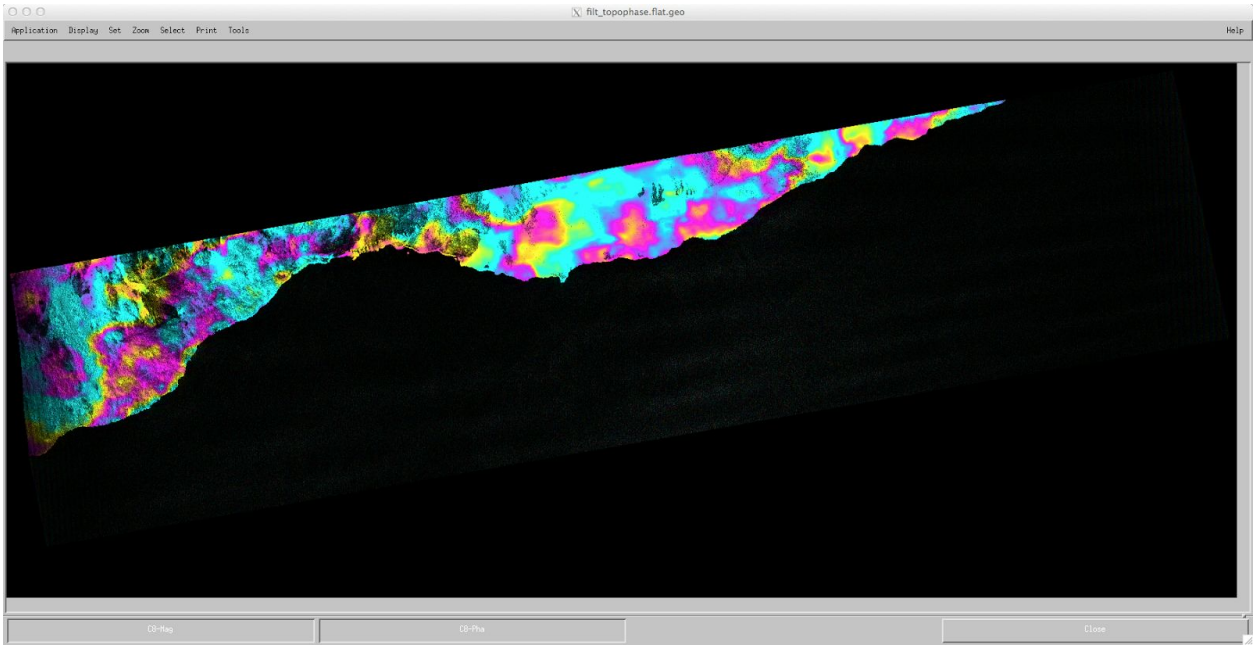


Note also that the run processed all the way through to the end, as evidenced by the screen output:

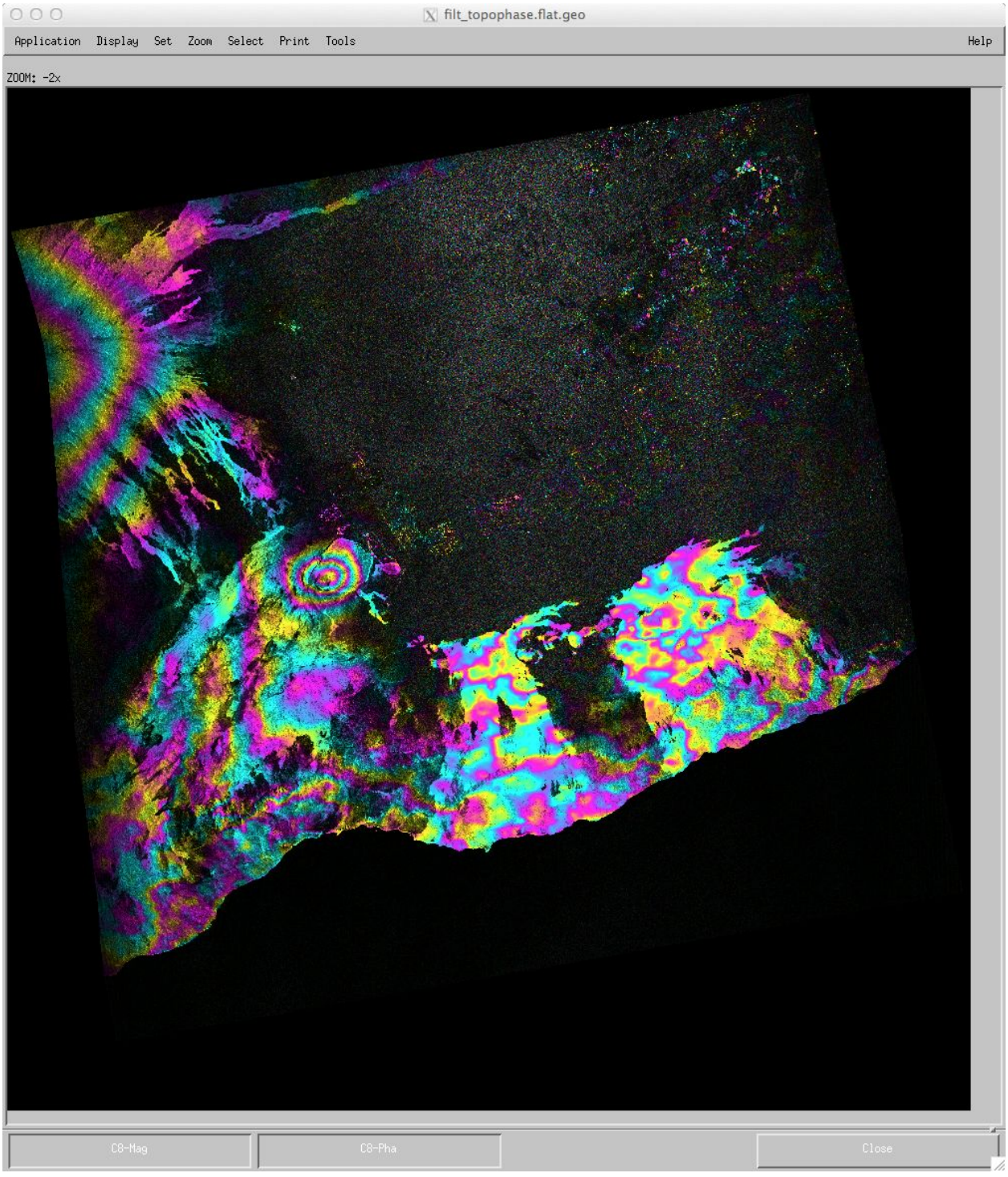
```
      runGeocode - Outputs
------------------------------------------------------------------------------------------------
------
runGeocode.outputs.LATITUDE_SPACING = -0.0002777777777777778
runGeocode.outputs.GEO_WIDTH = 1770
runGeocode.outputs.MAXIMUM_GEO_LATITUDE = 19.130277777777778
runGeocode.outputs.MINIMUM_GEO_LATITUDE = 19.343333333333334
runGeocode.outputs.LONGITUDE_SPACING = 0.0002777777777777778
runGeocode.outputs.MAXIMUM_GEO_LONGITUDE = -154.92333333333332
runGeocode.outputs.GEO_LENGTH = 768
runGeocode.outputs.MINIMUM_GEO_LONGITUDE = -155.41472222222222
################################################################################################
######
2014-07-28 12:26:34,690 - isce.insar - INFO - Total Time: 214 seconds
```

It is instructive to look at the final decoded interferogram `filt_topophase.flat.geo`. This is a geocoded interferogram with the topography removed and a smoothing filter applied to reduce the phase noise. Even with a fairly narrow sliver of land, the end-to-end processing works quite well, as seen in this screen shot of this image

```
> mdx.py filt_topophase.flat.geo
```



Encouraged by this, we can process the entire image by deleting the `formslc.xml` entirely, or if we want to keep the larger azimuth patch (which is more efficient in general), we can just delete the `NUMBER_PATCHES` property in the file. The result is the image below. Note the bulls-eye around Kilauea crater and the atmosphere-related fringes in the top right over Mauna Loa. Note also the decorrelation at X-band in COSMO-SkyMed data is high over vegetation, so there are few fringes visible north east in the forested areas.


filt_topophase.flat.geo
Application
Display
Set
Zoom
Select
Print
Tools
Help
ZOOM: -2x
C8-Mag
C8-Pha
Close

7. Notes about component configurability

This lab has used a new sensor type - COSMO-SkyMed - to illustrate a few of the configurability options available to users. Configurability is a potentially powerful way to control the behavior of the workflow components. However, users should be aware of some of the limitations and features that comes along with configurability, particularly at this early stage of development.

1. Some modules may not have not been architected to be configurable. Not all are expected to need it, so the developers have focused on those that are most commonly configured.
2. For those that have been architected appropriately, all nominal input parameters that are computed by the control scripts and passed into a compute module are made configurable, but some don't make sense to change. For example, you would not want to change the radar wavelength under any normal circumstance (though it there are cases where it might make sense!) or any of the other radar specific parameters.
3. Some parameters that you might want to change have interactions with other parts of the workflow, and though you would expect them to work, they don't. For example, in the example above, changing the starting pulse actually does not work robustly (try it!). It does actually process the data starting at the specified location, but things break further downstream because other parameters were set up outside `formslc` that assumed starting at the beginning. This needs to be improved in subsequent iterations of configurability.
4. It is non-trivial to discover the names of the configurable parameters if you are not a developer. There are a lot of them, and the developers have not yet documented them all.
5. Some parameters can be specified in the input file as well as in a configuration file. They do not have the same names, which is confusing. This will be improved in subsequent releases.

CHAPTER 8

Processing From SLC: COSMO-SkyMed, TerraSAR-X, RadarSAT-2, and others

1. Processing CSK from SLC

COSMO-SkyMed data are delivered either as processed imagery or as raw data. TerraSAR-X and RadarSAT-2 (and the future ALOS-2) data are always delivered as processed imagery; there is no possibility to process from Level 0. We saw in Lab 6 how to process CSK data from Level 0. In this lab, we will see how to set up ISCE for the three data sets that are or can be delivered as SLC data.

We saw in Lab 6 that the naming convention for the CSK data sets encodes all different data levels. The HDF5 files contain all the information needed to describe the radar data, the processing parameters of the imagery, the orbit and the geolocation of the products. Thus setting up the input files for CSK SLC data is essentially the same as it is for CSK raw data. In the lab7 directory, you will find two `.h5` data sets and an `insar.xml` input file.

```
> cd
> cd data
> cd lab7
> cd csk
> ls
CSKS2_SCS_U_HI_05_VV_RD_SF_20110525020512_20110525020519.h5
insarApp.xml
CSKS3_SCS_U_HI_05_VV_RD_SF_20110526020511_20110526020518.h5
```

The product type is seen to be SCS_U, which indicates SLC data (sometimes the name is SCS_B for CSK SLCs). We have created an input file with the right elements in it for these data to process:

```
> cat insarApp.xml
<insarApp>
<component name="insar">
    <property name="Sensor name">
        <value>COSMO_SKYMED_SLC</value>
    </property>
    <component name="master">
        <property name="OUTPUT">20110526.slc</property>
        <property
name="HDF5">CSKS3_SCS_U_HI_05_VV_RD_SF_20110526020511_20110526020518.
h5</property>
    </component>
    <component name="slave">
        <property name="OUTPUT">20110525.slc</property>
        <property
```

```
name="HDF5">CSKS2_SCS_U_HI_05_VV_RD_SF_20110525020512_20110525020519.
h5</property>
    </component>
<!--
    <component name="Dem">

<catalog>/u/proj8/dev/isce_regression/dem/demLat_N35_N37_Lon_W121_W12
0.dem.wgs84.xml</catalog>
    </component>
-->
    <property name="posting">
        <value>20</value>
    </property>
    <property name="unwrap">
        <value>False</value>
    </property>
    <property name="unwrapper name">
        <value>snaphu_mcf</value>
    </property>
    <property name="doppler method">
        <value>useDOPCSKSLC</value>
    </property>
</component>
</insarApp>
```

The beginning of the file should be familiar now, specifying the master and slave images. Since we are starting with SLC images, it is less confusing to set the OUTPUT names to end in .slc (but you can actually use .raw). The DEM specification is commented out, so ISCE will look to the internet database of SRTM data to download a DEM. The posting and unwrap properties should also be familiar from previous labs. What's new is the specification of the "`doppler method`" property to have a value of `useDOPCSKSLC`. Because the imagery are already processed, we do not have the freedom to specify the Doppler centroid for processing. For each data type that is delivered processed, we must specify a method to interpret and utilize the Doppler history in the imagery. If this parameter is not specified, the processing run fails because the Doppler is unspecified. Though the .h5 files have knowledge of their Doppler history, it is possible that the user might want to post-process the image with a different Doppler history. Allowing a method to be specified on input enables this possibility (though it is not implemented in `insarApp.py`). There is another possible value for this called `useDEFAULT`. This has the same effect as `useDOPCSKSLC`.

Once the input is set up properly, one can run `insarApp.py` as usual.

```
> insarApp.py insarApp.xml --steps
```

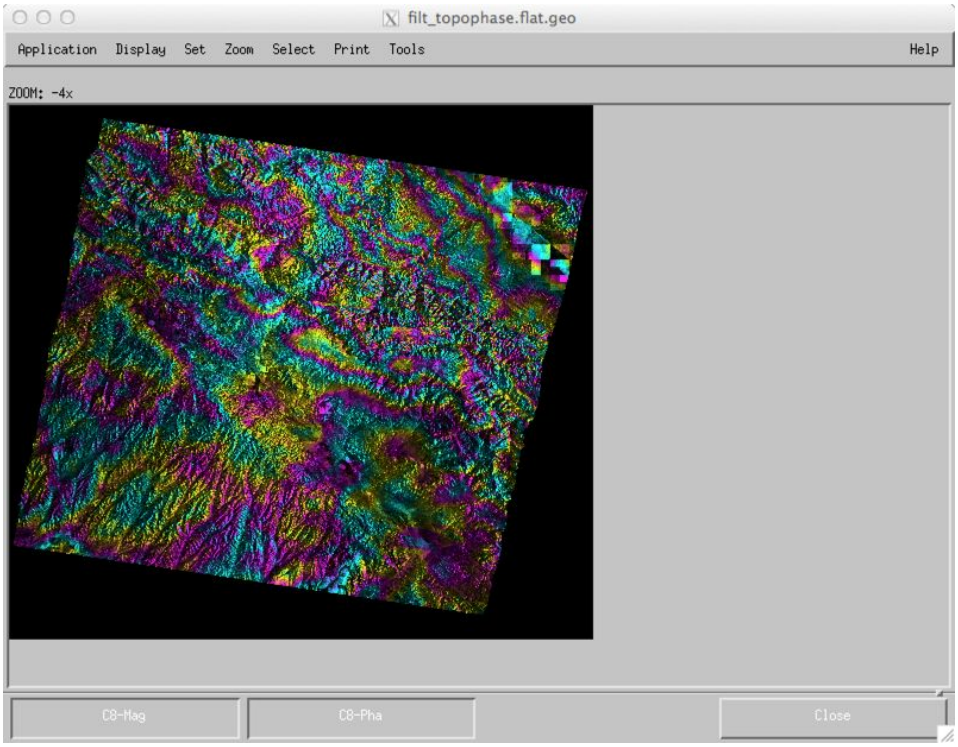
You'll note in the screen output that the steps are the same as for raw data even though formslc doesn't actually form an slc.

```
2014-07-29 20:30:18,466 - isce.insar - INFO - ISCE VERSION = 2.0.0,
RELEASE_SVN_REVISION = 1544,RELEASE_DATE = 20140724,
CURRENT_SVN_REVISION = 1525:1549M
ISCE VERSION = 2.0.0, RELEASE_SVN_REVISION = 1544,RELEASE_DATE =
20140724, CURRENT_SVN_REVISION = 1525:1549M
The Dem specified was not properly initialized. An SRTM Dem will be
downloaded.
Processing steps
self.step_list =  ['startup', 'preprocess', 'verifyDEM',
'pulsetiming', 'estimateHeights', 'mocompath', 'orbit2sch',
'updatepreprocinfo', 'formslc', 'offsetprf', 'outliers1',
'prepareresamps', 'resamp', 'resamp_image', 'mocompbaseline',
'settopoint1', 'topo', 'shadecpx2rg', 'rgoffset', 'rg_outliers2',
'resamp_only', 'settopoint2', 'correct', 'coherence', 'filter',
'unwrap', 'geocode', 'endup']
.
.
.
```

This is because `insarApp.py` is designed to preserve the overall interferometric flow, and `formslc` for an SLC input is essentially a resampling step since the image is already formed.

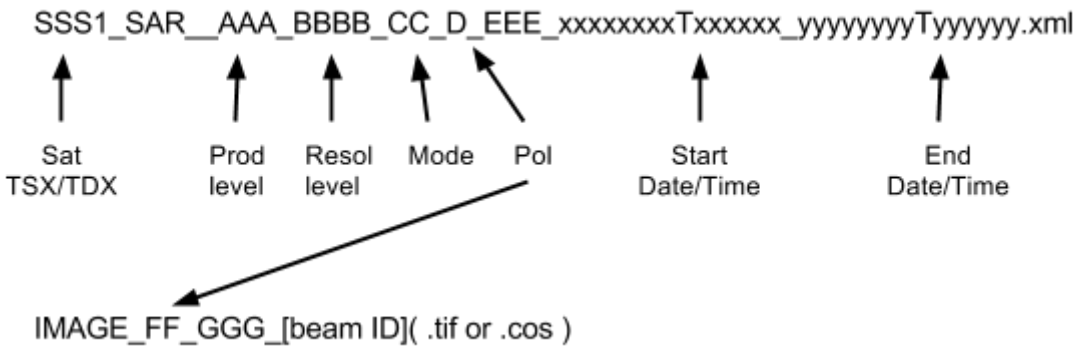
Configurability and setting of input file parameters and properties is possible for data sets starting from SLC as well. Certain parameters, for example the number of patches to process in formslc, obviously have no meaning when starting from SLCs, and the usual caveats about arbitrarily configuring parameters that should not be configured obtain. The final output of this CSK data over Parkfield looks pretty noisy from atmosphere and overly-smoothed decorrelation, but the interferogram has quite good fringe visibility because the time interval is only 1 day.

```
> mdx.py filt_topophase.flat.geo
```


filt_topophase.flat.geo
Application
Display
Set
Zoom
Select
Print
Tools
Help
ZOOM: -4x
C8-Mag
C8-Pha
Close

2. Processing TSX from SLC

The TerraSAR-X team has a similarly lengthy and descriptive file name as CSK.



This XML file captures a directory-based hierarchy of files that support the processing, including imagery, annotation, thumbnails, and other ancillary files. It is buried a few levels down in a directory tree for the data. We have set up the `master.xml` and `slave.xml` files to show you how to construct the path to the xml files.

```
> cd
> cd data/lab7/tsx
> cat master.xml
<component>
    <property name="XML">
dims_op_oc_dfd2_204281662_1/TSX-1.SAR.L1B/TSX1_SAR__SSC_____SM_S_SRA
_20091205T042212_20091205T042220/TSX1_SAR__SSC_____SM_S_SRA_20091205
T042212_20091205T042220.xml
    </property>
    <property name="OUTPUT">20091205.raw</property>
</component>
```

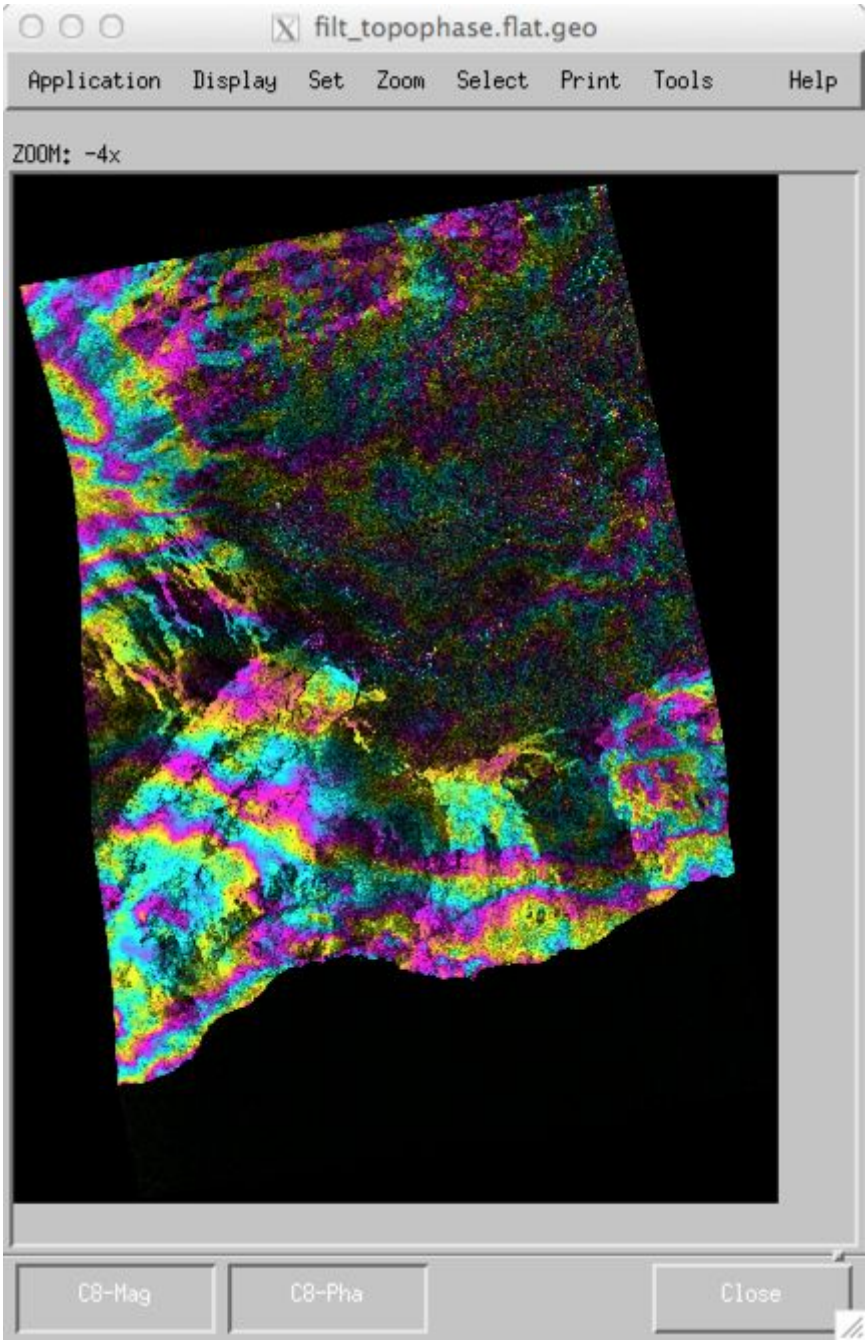
The top level directory is `dims_op_oc_dfd2_204281662_1/TSX-1.SAR.L1B.`
The next level directory is `TSX-1.SAR.L1B.`
The next level directory is
`TSX1_SAR__SSC_____SM_S_SRA_20091205T042212_20091205T042220.`
And finally, the xml file can be found in this directory:
`TSX1_SAR__SSC_____SM_S_SRA_20091205T042212_20091205T042220.xml.`

The `insarApp.py` input file is quite simple. It points to the master and slave catalogs, and specifies the posting and sensor name. As with CSK, the doppler method must also be specified, this time as `useDOPTSX`. It could also be `useDEFAULT`.

```
> cat insar_091205_091216.xml
<insarApp>
    <component name="insar">
        <property name="posting">
            <value>20</value>
        </property>
        <property name="Sensor Name">TerraSARX</property>
        <property name="doppler method">useDOPTSX</property>
        <component name="master">
            <catalog>20091205.xml</catalog>
        </component>
        <component name="slave">
            <catalog>20091216.xml</catalog>
        </component>
    </component>
</insarApp>
```

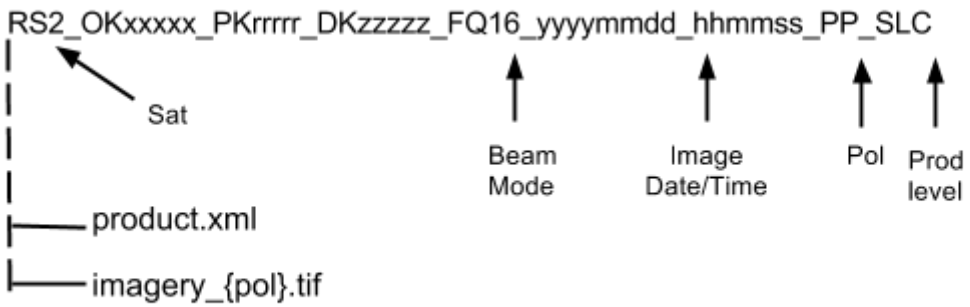
After running `insarApp.py` with this input file, you can display the output.

```
> mdx.py filt_topophase.flat.geo
```


filt_topophase.flat.geo
Application
Display
Set
Zoom
Select
Print
Tools
Help
ZOOM: -4x
C8-Mag
C8-Pha
Close

3. Processing RadarSAT-2 from SLC

By now you should know the drill pretty well: the SLC input files have long and complicated names but the data are nicely organized and self-contained, making the setup of the input files relatively straightforward. Let's do the same for RadarSAT-2. The data are typically delivered in a zipped file with the long complicated name as described here:



but upon unzipping the file, two simply named files are created, `product.xml` and `imagery_XX.tif`, where XX stands for the polarization state of the radar transmitter and receiver for this product. The xml description has a great deal of information about the nature of the data and how it was processed. The tif file is the imagery itself, in geotiff format.

Note in this lab, we have organized the master and slave data sets into two directories named by date:

```
> cd
> cd data/lab7/rs2
> ls 2010*
20100402/:
imagery_HH.tif  product.xml

20100707/:
imagery_HH.tif  product.xml
```

which makes it easy to setup the insarApp.py input file:

```
> cat insarApp.xml
<insarApp>
    <component name="insar">
        <property name="unwrap">
```

```
            <value>True</value>
        </property>
        <property name="doppler method">useDEFAULT</property>
        <property name="Sensor Name">
            <value>RADARSAT2</value>
        </property>
        <property name="slc offset method">ampcor</property>
        <property name="posting">30</property>
        <component name="master">
            <property name="xml">
                <value>20100707/product.xml</value>
            </property>
            <property name="tiff">
                <value>20100707/imagery_HH.tif</value>
            </property>
            <property name="OUTPUT">
                <value>20100707.raw</value>
            </property>
        </component>
        <component name="slave">
            <property name="xml">
                <value>20100402/product.xml</value>
            </property>
            <property name="tiff">
                <value>20100402/imagery_HH.tif</value>
            </property>
            <property name="OUTPUT">
                <value>20100402.raw</value>
            </property>
        </component>
    </component>
</insarApp>
```

This file follows a familiar pattern and exploits some properties that perhaps we have not seen before, for example `slc_offset_method`, which specifies a particular component to use, and the `doppler method` is now explicitly set to `useDEFAULT` rather than something specific to RadarSAT-2.

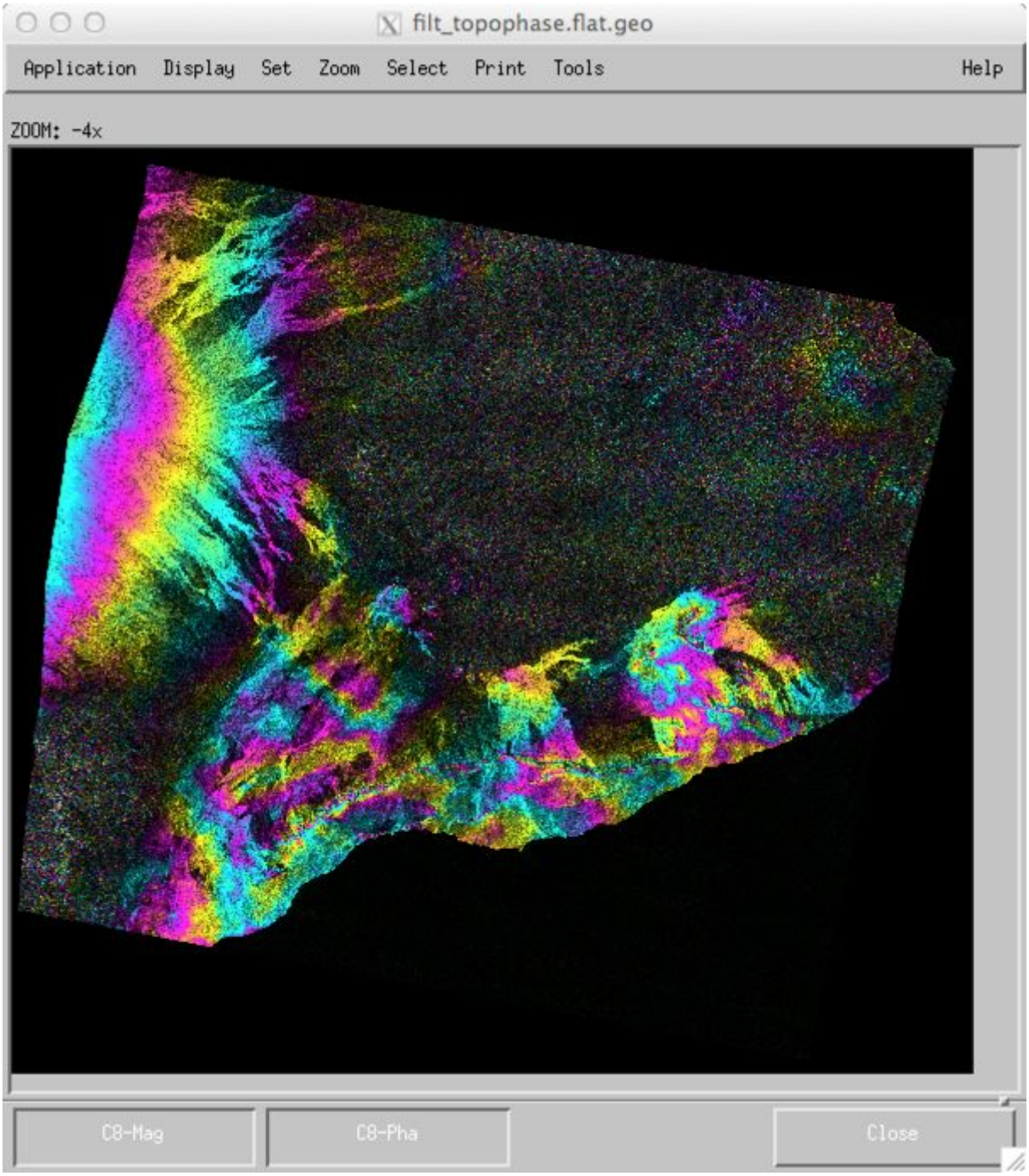
After running this example,

```
> insarApp.py insarApp.xml
```

we can display the output

```
> mdx.py filt_topophase.flat.geo
```

(see below). Note that unwrapping was turned on, but the output data (filt_topophase.unw.geo) is blank, so clearly it didn't work. This is because the default unwrapping scheme was used, and it doesn't work well when the middle of the scene, where the unwrapping origin is placed, is very noisy. Moving the origin to a less noisy region would likely fix this. Can you figure out how to do it?


filt_topophase.flat.geo
Application
Display
Set
Zoom
Select
Print
Tools
Help
ZOOM: -4x
C8-Mag
C8-Pha
Close

4. Running ISCE modules outside insarApp.py

New lab exercise from Piyush

```
> cd lab5/env/20100502_20100328/
> python3
Python 3.2.3 (default, Feb 27 2014, 21:31:18)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
enabling readline
>>> import isce
>>> import isceobj
>>> img = isceobj.createDemImage()
>>> img.load('filt_topophase.flat.geo.xml')
>>> img.length
2560
>>> img.width
2009
>>> img.renderEnviHDR()
>>> exit
Use exit() or Ctrl-D (i.e. EOF) to exit
>>>
> more filt_topophase.flat.geo.hdr
ENVI
description = {Data product generated using ISCE}
samples = 2009
lines   = 2560
bands   = 1
header offset = 0
file type = ENVI Standard
data type = 6
interleave = bip
byte order = 0
coordinate system string =
{GEOGCS["GCS_WGS_1984",DATUM["D_WGS_1984",SPHEROID
["W
GS_1984",6378137,
298.257223563]],PRIMEM["Greenwich",0],UNIT["Degree",0.01745
329
25199433]]}
map_info = {Geographic Lat/Lon, 1.0, 1.0, -116.09083333333334,
```

```
33.48750000000
000
4, 0.0008333333333333334, 0.0008333333333333334, WGS-84,
units=Degrees}
>
```

CHAPTER 9

ISCE Stack Processing for GIAnT

1. Stack Processing for GIAnT

In the previous lab sessions, we learned to use insarApp.py to generate individual interferograms. In this lab, we will explore the possibility of processing interferogram stacks i.e, multiple interferograms that are co-located on a common image grid using ISCE. The process of resampling multiple interferograms to a common grid is called “stacking”.

“Stacking” can be performed in radar image coordinates (range and azimuth axes) or in geo coordinates (lat and lon axes). In this lab session, we demonstrate a simple stacking approach in geo-coordinates. To generate our interferograms stacks, we will combine the following features of ISCE that were discussed in the previous sessions:

1. Using insarApp.py with the --steps option
2. Modifying the input XML files to customize the output geocoded grid
3. Modifying the input XML files to customize the set of geocoded output products

In this lab session, we explain the various steps involved in generating stacks with ISCE. We will not be processing any actual interferogram stacks from scratch due to space and time restrictions, but will walk through the various steps in sequence. No advanced Python programming skills are assumed for this tutorial. Users can implement these steps with the scripting language of their choice - Python, bash etc. Future versions of ISCE will include an application called isceApp.py which will automate the processing of intereferogram stacks.

2. Organizing the data

We demonstrate our approach to stack processing using an hypothetical COSMO SkyMed dataset (Note again that we won't be processing any real data). We organize the COSMO-SkyMed data corresponding to our stack in the following directory structure:

```
lab8
|-- dem   (Directory for storing DEM)
|   |-- demLat_N36_N38_Lon_W123_W121.dem
|   `-- demLat_N36_N38_Lon_W123_W121.dem.xml
|-- raw      (Directory for storing sensor data)
|   |-- 20130531 (Directory for data acquired on a particular date)
|   |-- 20130616
|   |-- 20130702
|   |-- 20130718
|   |-- 20130803
|   |-- 20130819
|   `-- 20130904
`-- insar   (Directory to store the processed interferograms)
```

Each of the date directories could again contain multiple files corresponding to consecutive frames. The XML configuration files shown in this walk through will automatically be combine the multiple frames to create a single raw image file. An example date directory is shown below:

```
20130616/
|-- CSKS1_RAW_B_HI_09_HH_RA_SF_20130616135654_20130616135701.h5
|-- CSKS1_RAW_B_HI_09_HH_RA_SF_20130616135659_20130616135706.h5
`-- CSKS1_RAW_B_HI_09_HH_RA_SF_20130616135704_20130616135710.h5
```

3. Determining viable interferogram pairs

In this section, we demonstrate the simplest method of generating the baseline plot for our stack of acquisitions. Choose one particular date (say the earliest acquisition as reference). For our hypothetical example, that would be “20030531”.

Step 1:

In the insar directory create directories named 20030531_date2, where date2 corresponds to all the other acquisition dates in the stack.

Step 2:

Populate each of the InSAR pair directories with the minimal insarApp.xml and SAR catalog files following instructions in Lab 6.

The insarApp.xml file would look like:

```
<insarApp>
    <component name="insar">
        <component name="Dem">
<catalog>../dem/demLat_N36_N38_Lon_W123_W121.dem.xml</catalog>
        </component>
        <component name="slave">
            <catalog>20130531.xml</catalog>
        </component>
        <property name="Sensor name">
            <value>COSMO_SKYMED</value>
        </property>
        <component name="master">
            <catalog>20130616.xml</catalog>
        </component>
    </component>
</insarApp>
```

SAR catalog file for the 20130531 acquisition would look like:

```
<component name="sar">
    <property name="OUTPUT">
        <value>20130531.raw</value>
    </property>
    <property name="HDF5">
      <value>['../../h5/20130531/CSKS1_RAW_B_HI_01_HH_RD_SF_20130531021035_2013
0531021042.h5','../../h5/20130531/CSKS1_RAW_B_HI_01_HH_RD_SF_20130531021030_201
30531021037.h5','../../h5/20130531/CSKS1_RAW_B_HI_01_HH_RD_SF_20130531021040_20
```

```
130531021047.h5']</value>
    </property>
</component>
```

Step 3:

In each of the directories, run insarApp.py till the step named “preprocess”. This will add required baseline information to the insarProc.xml in each directory.

```
> insarApp.py --end=preprocess
> tail -n 12 insarProc.xml
    </slave>
    <baseline>
            <horizontal_baseline_top>96.7928771266</horizontal_baseline_top>
            <horizontal_baseline_rate>-1.73848950218e-05</horizontal_baseline_r
            ate>
            <horizontal_baseline_acc>8.52094426395e-11</horizontal_baseline_acc
      >
            <vertical_baseline_top>29.755105435</vertical_baseline_top>
            <vertical_baseline_rate>8.06177637777e-07</vertical_baseline_rate>
            <vertical_baseline_acc>-4.07016302367e-12</vertical_baseline_acc>
            <perp_baseline_top>-64.1586857667</perp_baseline_top>
            <perp_baseline_bottom>-63.4138138722</perp_baseline_bottom>
    </baseline>
</insarProc>
```

Use the perpendicular baseline information and acquisition dates to determine the interferogram pairs that you would like to process. Save the dates corresponding to the pairs that you want to process to a text file in the parent directory (“lab8”). Clear out the “insar” directory.

Note:
ISCE is a modular toolbox, and users familiar with Python programming can directly use ISCE modules to generate baseline plots. Use isce/components/isceobj/InsarProc/runPreprocessor.py as a template.

4. Determine common output grid

Before processing all the viable interferograms, we need to determine the common output grid. We will do so by processing one pair from the stack (typically a pair with short spatial and temporal baselines). To save time, we will turn off phase unwrapping and only geocode the wrapped interferogram. Lets say we process the pair 20130616_20130531 in the "insar/20130616_20130531" directory.

Modify the insarApp.xml file for the chosen pair to look like this:

```
<insarApp>
    <component name="insar">
        <component name="Dem">
<catalog>../dem/demLat_N36_N38_Lon_W123_W121.dem.xml</catalo
g>
        </component>
        <component name="slave">
            <catalog>20130531.xml</catalog>
        </component>
        <property name="unwrap">
            <value>False</value>
        </property>
        <property name="geocode list">
            <value>['filt_topophase.flat']</value>
        </property>
        <property name="Sensor name">
            <value>COSMO_SKYMED</value>
        </property>
        <component name="master">
            <catalog>20130616.xml</catalog>
        </component>
    </component>
</insarApp>
```

Now run insarApp.py in the example directory.

```
> insarApp.py --steps
```

To determine the bounding box of the processed interferogram, scroll down to the end of insarProc.xml

```
> tail -n 12 insarProc.xml
     <outputs>
     <GEO_WIDTH>3853</GEO_WIDTH>
```

```
            <MAXIMUM_GEO_LONGITUDE>-121.64999999999999</MAXIMUM_GEO_LONGITUDE>
       <MINIMUM_GEO_LONGITUDE>-122.72</MINIMUM_GEO_LONGITUDE>
       <LATITUDE_SPACING>-0.00027777777777777778</LATITUDE_SPACING>
       <GEO_LENGTH>4864</GEO_LENGTH>
       <LONGITUDE_SPACING>0.00027777777777777778</LONGITUDE_SPACING>
       <MAXIMUM_GEO_LATITUDE>36.899444444444434</MAXIMUM_GEO_LATITUDE>
       <MINIMUM_GEO_LATITUDE>38.250277777777775</MINIMUM_GEO_LATITUDE>
      </outputs>
    </runGeocode>
</insarProc>
```

We now have the information needed to set up the bounding box ([South, North, West, East]) in the insarApp.xml files. You may add a little padding around this estimated bounding box if needed. Clear out the “insar” directory.

5. Preparing directories for interferogram generation

In this step, we will set up the directories for all viable interferograms and the insarApp.xml files with correct values for generating stacks.

Step 1:

Create a new directory for each of the viable interferograms under the “insar” directory and create corresponding SAR catalog XML files as well in each sub directory.

Step2:

Determine common processing parameters. Setup insarApp.xml and SAR catalog XML files accordingly. Some of the commonly manipulated parameters for stack processing are:

Processing parameter	**Value**	**Description**
Doppler method	useDEFAULT	COSMO SkyMed metadata includes doppler information
Unwrap method	snaphu_mcf	Snaphu unwrapper using the MCF algorithm
Geocode bounding box	[36.85, 38.3, -122.75, -122.6]	SNWE determined from Step 4 above.
Geocode list	[‘filt_topophase.unw’, ‘phsig.cor’, ‘topophase.cor’, ‘los.rdr’]	Geocode only the files needed by GIAnT or for modeling.
Filter strength	0.5	Goldstein filter strength
Posting	20	Choose a posting comparable but less than DEM posting for best performance.

The corresponding insarApp.xml file looks like:

```
<insarApp>
    <component name="insar">
        <property name="posting">
            <value>20</value>
        </property>
         <property name="doppler method">
```

```
            <value>useDEFAULT</value>
        </property>
        <property name="unwrap">
            <value>True</value>
        </property>
        <property name="unwrapper name">
            <value>snaphu_mcf</value>
        </property>
         <property name="geocode bounding box">
              <value>[36.85,38.3,-122.75,-122.6]</value>
         </property>
         <property name="geocode list">
              <value>['filt_topophase.unw', 'phsig.cor',
'topophase.cor','los.rdr']</value>
         </property>
         <property name="filter strength">
             <value>0.5</value>
        </property>
        <property name="Sensor name">
            <value>COSMO_SKYMED</value>
        </property>
        <component name="master">
            <catalog>20130616.xml</catalog>
        </component>
        <component name="slave">
            <catalog>20130531.xml</catalog>
        </component>
        <component name="Dem">
            <catalog>../../dem/demLat_N36_N38_Lon_W123_W121.dem.xml</catalog>
        </component>
    </component>
</insarApp>
```

Each interferogram directory is now ready to support an independent “insarApp.py” processing run.

Step 3:

Run “insarApp.py” in individual directories and the output geocoded products are ready to be directly ingested into GIAnT. The presented approach has following advantages:

1. Allows for customized processing of individual interferograms. Each interferogram processing is localized to a single directory. Add all configuration XML files to the processing directory. Can modify individual insarApp.xml files.
2. Independent processing of interferograms. Once the structure of the input XML files are known, interferograms can be processed independently on different machines / networks/ cloud instances.

CHAPTER 10

Working with GIAnT

1. Intro - Preparing the stack for analysis

Note: Execute all the commands in this lab session on a terminal in the Guacamole interface.

GIAnT is designed to work with outputs from multiple SAR/InSAR processors -e.g, ISCE, ROI_PAC etc. The very first stage of processing with GIAnT transforms InSAR products from their native formats (e.g, GMTSAR's grd files, ISCE''s binary files etc) to an internally consistent Hierarchical Data Format 5 (HDF5) format.

In this tutorial, we will describe the steps involved in transforming all the input data (described in the previous tutorial) into a HDF5 format needed by GIAnT. Again, we start with our test dataset located in the directory "synthetic":

```
> cd /home/ubuntu/data/giant/kilaeau/GIAnT
> ls
example.xml  ifg.list  prepdataxml.py  prepsbasxml.py  userfn.py
```

From amongst the various python scripts in the directory - "userfn.py" and "prepdataxml.py" are needed for preparing our data stack for analysis. The other python scripts are related to the actual time-series analysis and will be discussed in later tutorials.

2. userfn.py - Translating pair information to actual files on disk

As described in the previous tutorial, “ifg.list” is a four column text file that describes our interferogram network in a simple fashion.

```
> less ifg.list
20110310   20101212   -90.8372435038   CSK
20110322   20101220   -53.7819100954   CSK
20110407   20110326   183.599267061   CSK
20101228   20101220   -259.347791029   CSK
20110322   20110318   -112.975218835   CSK
...
```

We also mentioned that we stored our unwrapped phase and coherence files in individual sub-directories in a directory named “insar”. But we never provided the exact mapping between each line of “ifg.list” and the corresponding files in “insar”. This is accomplished through userfn.py .

```
>less userfn.py
def makefnames(dates1, dates2, sensor):
    dirname = '../insar'
    root = os.path.join(dirname, dates1+'_'+dates2)
    iname = os.path.join(root, 'filt_topophase.unw.geo')
    cname = os.path.join(root, 'topophase.cor.geo')
    return iname, cname
```

“userfn.py” should define a function named “makefnames” that takes the the master date, slave date and sensor name as inputs and returns two strings that represent the path to the unwrapped phase file and the coherence file. “userfn.py” should be located in your working directory.

This particular mechanism was devised to allow users to store InSAR outputs using their preferred directory and file name structure. Note that “userfn.py” should be considered as an user input, and each stack should be accompanied by its own “userfn.py”.

3. “prepdataxml.py” - setting up data characteristics.

“prepdataxml.py” is responsible for generating the input file “data.xml” which describes the characteristics of the dataset like dimensions, looks, formats etc.

```
> less prepdataxml.py
#!/usr/bin/env python

import tsinsar as ts
import argparse
import numpy as np

if __name__ == '__main__':

    ######Prepare the data.xml
    g = ts.TSXML('data')
    g.prepare_data_xml('example.xml', proc='ISCE',
                       xlim=[0,2118], ylim=[0, 1920],
                       rxlim = [1395,1405], rylim=[1420,1430],
                       latfile='', lonfile='', hgtfile='',
                       inc = 21., cohth=0.1, chgendian='False',
                       unwfmt='RMG', corfmt='RMG')
    g.writexml('data.xml')
```

We set up some basic parameters for processing our stack using “prepdataxml.py”. The complete list of all configurable parameters can be found in the GIAnT user manual. We describe the parameters that we have set up using prepdataxml.py below:

example.xml	Example ISCE insarProc.xml file with dimensions and wavelength information
proc	Default is RPAC. We set it to ‘ISCE’
xlim	X limits for cropping the image (Python convention). We use the full image here.
ylim	Y limits for cropping the image (Python convention). We use the full image here.
rxlim	X limits of reference region. Pixel 30-49 in range. (zero index)
rylim	Y limits of referenec region. Line 50-69 in azimuth. (zero

	index)
latfile, lonfile, hgtfile	Files for lat, lon, height in radar coordinates. This information is needed for atmospheric corrections, which are currently not used. These are described in the tutorial on advanced topics.
inc	Incidence angle (constant or file). Again only use for atmospheric corrections and GPS comparison. Not used in this tutorial.
cohth	Coherence threshold. All phase measurements with coherence less than this value are considered invalid.
chgendian	To the input files are in a different format than the native machine format.
unwfmt	FLT/RMG to indicate that the input is one or two channel file.
corfmt	FLT/RMG to indicate that the input is one or two channel file.

The default data type for all files is float32. See GIAnT user manual for complete list of options and default values.

We will then generate our “data.xml” script as follows:

```
> python prepdataxml.py
```

To view the generated “data.xml” file,

```
> less data.xml
```

```
<data>
  <proc>
    <value>ISCE</value>
    <type>STR</type>
    <help>Processor used for generating the interferograms.</help>
  </proc>
  <master>
    <width>
      <value>2118</value>
      <type>INT</type>
      <help>WIDTH of the IFGs to be read in.</help>
```

```
    </width>
    <file_length>
      <value>1920</value>
      <type>INT</type>
      <help>FILE_LENGTH of the IFGS to be read in.</help>
    </file_length>
    <wavelength>
      <value>0.0312283810417</value>
      <type>FLOAT</type>
      <help>WAVELENGTH of the Stack. If combining sensors,ensure that
they are all converted to same units.</help>
    </wavelength>
...
```

Note that the generated XML file can be modified in a text editor, and we include a help string to describe each of the parameters in the file.

We are now ready to gather data into a HDF5 file readable by GIAnT.

4. PrepIgramStack.py - preparing the stack

From the GIAnT working directory, execute PrepIgramStack.py.

(NOTE: The PrepIgramStack.py command and many of the rest in this lab need to be run from the X11 windows in the Remote Desktop function of EarthKit.)

```
> cd /home/ubuntu/data/giant/kilaeau/GIAnT

> PrepIgramStack.py
<module> - INFO - No common mask defined
<module> - INFO - Output h5file: Stack/RAW-STACK.h5
<module> - INFO - PNG preview dir: Figs/Igrams
<module> - INFO - Deleting previous Stack/RAW-STACK.h5
<module> - INFO - Reading in IFGs
[========================= 59% =>                    ]    134s / 93s
```

As indicated by the screen output, the program generates a file named “Stack/RAW-STACK.h5” in the Stack directory and another directory called “Figs/Igrams”.

```
> ls
data.xml     Figs      prepdataxml.py  Stack      userfn.pyc
example.xml  ifg.list  prepsbasxml.py  userfn.py


> ls Stack
RAW-STACK.h5

> ls Figs
Igrams

> ls Figs/Igrams
I001-20110310-20101212-CSK.png  I024-20110404-20110214-CSK.png
I002-20110322-20101220-CSK.png  I025-20110302-20101228-CSK.png
I003-20110407-20110326-CSK.png  I026-20110404-20110403-CSK.png
I004-20101228-20101220-CSK.png  I027-20110129-20101228-CSK.png
...
```

HDF5 outputs of all GIAnT programs are stored in the “Stack” directory and associated PNG previews are generated in a directory named “Figs”.

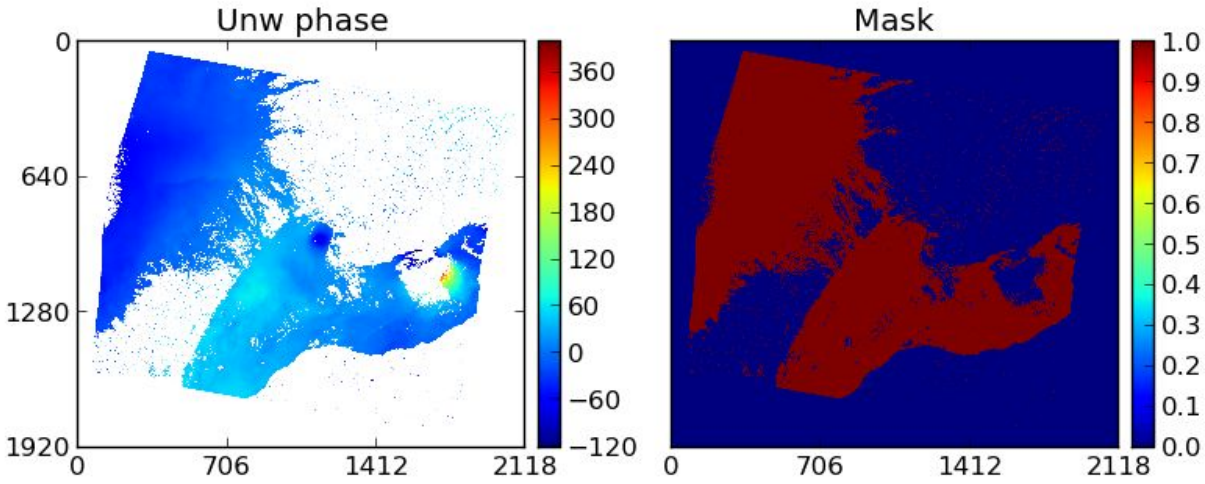
5. PNG previews - What does our data look like?

The directory Figs/Igrams contains PNG previews of all unwrapped interferograms listed in ifg.list . The PNG files are numbered in sequence. The PNG preview corresponding to the 80th interferogram in our test data set.

To preview the PNG files, run the following command: (NOTE: you will need to run this image preview command from the Remote Desktop as it is graphical in nature)

```
> cd Figs/Igrams
> eog *.png
```

Notice that a coherence threshold has been applied to the interferograms depending on the user inputs in data.xml. The unwrapped phase has been converted to mm at this stage.



6. Listing contents of RAW-STACK.h5

In this section, we will try to understand the structure of the HDF5 file `Stack/RAW-STACK.h5` created by “PrepIgramStack.py”. We can summarize the contents of this file using `h5ls`

```
> cd ../..
> h5ls Stack/RAW-STACK.h5
Jmat                     Dataset {46, 17}
bperp                    Dataset {46}
cmask                    Dataset {1920, 2118}
dates                    Dataset {17}
igram                    Dataset {46, 1920, 2118}
tims                     Dataset {17}
usat                     Dataset {17}
```

This lists the various arrays stored in the HDF5 file and their corresponding sizes.

The details on a particular variable, say Jmat, can be obtained using `h5dump`

```
> h5dump -a /Jmat/help Stack/RAW-STACK.h5
HDF5 "RAW-STACK.h5" {
ATTRIBUTE "/Jmat/help" {
   DATATYPE  H5T_STRING {
         STRSIZE 28;
         STRPAD H5T_STR_NULLPAD;
         CSET H5T_CSET_ASCII;
         CTYPE H5T_C_S1;
      }
   DATASPACE  SCALAR
   DATA {
   (0): "Connectivity matrix [-1,1,0]"
   }
}
}
```

Every HDF5 dataset created by GIAnT includes a self-explanatory “help” attribute which is listed in the “Data{}” section of the output from the `h5dump` command.

RAW-STACK.h5 has all the data we need to proceed to the next stage of time-series processing, stored in a convenient and easily accessible format.

1. Quick recap

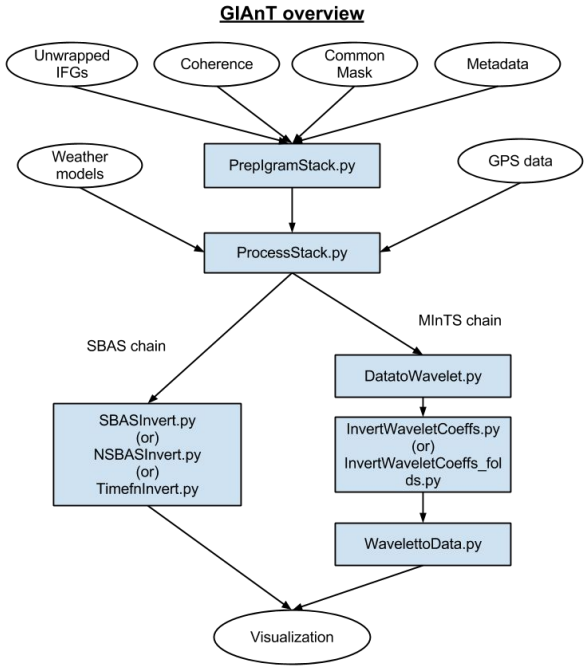
So far, we have gathered all the required network, unwrapped phase and coherence information into a HDF5 file using “PrepIgramStack.py” in the previous tutorial.

```
> cd /home/ubuntu/data/giant/kilaeau/GIAnT
> h5ls Stack/RAW-STACK.h5
Jmat                     Dataset {46, 17}
bperp                    Dataset {46}
cmask                    Dataset {1920, 2118}
dates                    Dataset {17}
igram                    Dataset {46, 1920, 2118}
tims                     Dataset {17}
usat                     Dataset {17}
```

In this tutorial, we will apply optional corrections to the ingested stack and estimate the deformation time-series using the SBAS technique. We will also teach users to interactively visualize some of the time-series results.

2. Setting up the processing parameters

In the previous tutorial, we described how the dataset parameters are controlled using “data.xml”. In this tutorial, we will learn to set up the processing parameters using a similar XML file - “sbas.xml”. This processing file is specific for the SBAS set of time-series inversions (See figure below).



In the example dataset directory, you will find a script named “prepsbasxml.py” .

```
> less prepsbasxml.py
#!/usr/bin/env python

import tsinsar as ts
import argparse
import numpy as np

if __name__ == '__main__':
    g = ts.TSXML('params')
    g. prepare_sbas_xml(nvalid = 30, netramp=True, atmos='',
                        demerr = False, filt=0.05)

    g.writexml('sbas.xml')
```

The complete list of all configurable parameters in “sbas.xml” can be found in the GIAnT user manual. We describe the parameters that we have set up using prepsbasxml.py below:

nvalid	Used for NSBAS inversion. Determines the minimum number of interferograms that a pixel should be coherent to be considered for inversion.
netramp	Boolean parameter controlling the deramping of interferograms. In this case, the applied ramp corrections are consistent over the entire network.
atmos	ProcessStack.py can download weather model data and use that for stratified trophospheric phase delay correction. This is beyond the scope of this tutorial. We use a empty string to indicate that no weather model corrections are to be applied.
demerr	Boolean parameter indicating if a DEM error term needs to be estimated. The baseline information from ifg.list is used for DEM error estimation.
filt	Width of the Gaussian filter to applied to the raw time-series to obtain the smoothed estimates. The value of this parameter is in years.

See GIAnT user manual for complete list of options and default values.

```
> python prepsbasxml.py
```

To view the generated “sbas.xml” file,

```
> less sbas.xml
<params>
  <proc>
    <nvalid>
      <value>30</value>
      <type>INT</type>
      <help>Minimum number of coherent IFGs for a single pixel. If
zero, pixel should be coherent in all IFGs.</help>
    </nvalid>
    <uwcheck>
      <value>False</value>
      <type>BOOL</type>
    </uwcheck>
```

```
    <netramp>
      <value>True</value>
      <type>BOOL</type>
      <help>Network deramp. Remove ramps from IFGs in a network
sense.</help>
    </netramp>
    <gpsramp>
      <value>False</value>
      <type>BOOL</type>
      <help>GPS deramping. Use GPS network information to correct
ramps.</help>
    </gpsramp>
    <stnlist>
      <value></value>
      <type>STR</type>
      <help>Station list for position of GPS stations.</help>
    </stnlist>
…...
</params>
```

Remember that the generated XML file can be modified in a text editor, and we again include a help string to describe each of the parameters in the file.

We are now ready to process our stack from the HDF5 file.

3. ProcessStack.py - Applying corrections

The first stage of processing, in which the data supplied by the users is modified, is accomplished using “ProcessStack.py”. The aim of this step is to

1. Correct for stratified troposphere artifacts, either
    a. Empirically by looking at relationship between InSAR phase and DEM
    b. Using weather models through the PyAPS package
2. Estimate ramps introduced due to orbital errors, either
    a. Either empirically by fitting a predefined orbit error function to data
    b. Using dense GPS observations

All the corrections are applied consistently across the interferogram network.

For this tutorial, we only choose to empirically deramping of interferograms. Details regarding other options and the associated fields in “sbas.xml” can be found in the GIAnT user manual.

Run “ProcessStack.py”

(NOTE: The ProcessStack.py command needs to be run from the X11 windows in the Remote Desktop function of EarthKit.)

```
> pwd
/home/ubuntu/data/giant/kilaeau/GIAnT

> ProcessStack.py
logger - INFO - GIANT Toolbox - v 1.0
logger - INFO - --------------------
<module> - INFO - Input h5file: Stack/RAW-STACK.h5
<module> - INFO - Deleting previous Stack/PROC-STACK.h5
<module> - INFO - Output h5file: Stack/PROC-STACK.h5
deramp - INFO - PROGRESS: Estimating individual ramps.
[====================== 98% ==================> ]     17s /      0s
deramp - INFO - PROGRESS: Network deramp of IFGs.
[====================== 98% ==================> ]     27s /      0s
<module> - INFO - PNG preview of Deramped images: Figs/Ramp
[======>                11%                     ]     42s /    342s
```

Outputs of “ProcessStack.py” include - a processed stack file “Stack/PROC-STACK.h5” and a directory of PNG previews of deramped interferograms “Figs/Ramp”.

```
> ls Stack
PROC-STACK.h5 RAW-STACK.h5
```

To preview the contents of the new stack file

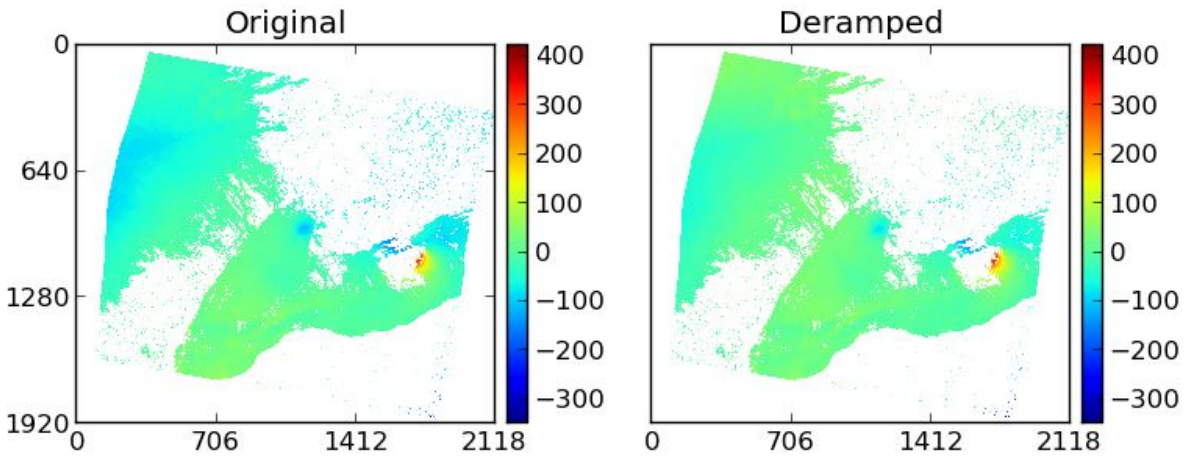
```
>h5ls Stack/PROC-STACK.h5
Jmat                     Dataset {46, 17}
bperp                    Dataset {46}
cmask                    Dataset {1920, 2118}
dates                    Dataset {17}
figram                   Dataset {46, 1920, 2118}
ramp                     Dataset {46, 3}
tims                     Dataset {17}
```

To view the contents of the directory with the PNG previews.

```
> ls Figs/Ramp
I001.png  I007.png  I013.png  I019.png  I025.png  I031.png  I037.png
I043.png
I002.png  I008.png  I014.png  I020.png  I026.png  I032.png  I038.png
I044.png
……..
```

To see the effect of deramping on the 7th interferogram in the Stack: (NOTE: you need to run this from Remote Desktop for the graphical viewer)

```
> eog Figs/Ramp/I007.png
```



Our stack is now deramped and ready for the final time-series inversion.

4. NSBASInvert.py - Final inversion

In this tutorial, we demonstrate the simplest time-series inversion algorithm implemented in GIAnT - the NSBAS algorithm. GIAnT also implements two other algorithms in the SBAS chain - SBASInvert.py and TimefnInvert.py. The detailed discussion on the differences between these approaches can be found in the GIAnT user manual.

The NSBAS algorithm estimates the differential displacement between one SAR acquisition and the next using a simple least squares approach. Optionally, it uses a user-defined temporal form to connect disconnected components of an interferogram network. By default, NSBAS uses a quadratic polynomial for connecting disconnected components. Our implementation of the algorithm estimates the time-series only for the pixels that are considered coherent over atleast "nvalid" interferograms in the entire stack.

```
> NSBASInvert.py -nproc 4
logger - INFO - GIANT Toolbox - v 1.0
logger - INFO - ---------------------
Timefn - INFO - Adding 17 linear pieces (SBAS)
<module> - INFO - No modification in the NSBAS constraint functional
form
<module> - INFO - Assuming a quadratic polynomial form
Timefn - INFO - Adding order 0 at T = 0.000000
Timefn - INFO - Adding order 1 at T = 0.000000
Timefn - INFO - Adding order 2 at T = 0.000000
<module> - INFO - Output h5file Stack/NSBAS-PARAMS.h5
<module> - INFO - Number of parallel processes: 4
<module> - INFO - Relative weight of polynomial constraint :
1.000000e-04
[====================== 99% ==================> ]     800s /      0s
```

This script has been parallelized for performance, and "-nproc 4" on the command line instructions the program to launch the analysis on 4 threads.

"NSBASInvert.py" stores the inversion results in "Stack/NSBAS-PARAMS.h5".

```
> h5ls Stack/NSBAS-PARAMS.h5
bperp                    Dataset {46}
cmask                    Dataset {1920, 2118}
dates                    Dataset {17}
gamma                    Dataset {SCALAR}
ifgcnt                   Dataset {1920, 2118}
mName                    Dataset {3}
```

```
masterind                Dataset {SCALAR}
parms                    Dataset {1920, 2118, 3}
rawts                    Dataset {17, 1920, 2118}
recons                   Dataset {17, 1920, 2118}
regF                     Dataset {3}
tims                     Dataset {17}
```

Note that the HDF5 file contains the raw time-series estimates (rawts) as well as the filtered time-series estimates (recons). In the next couple of sections, we will describe the visualization tools that are included with GIAnT.

5. plotts.py - Interactive visualization

GIAnT includes a script called “plotts.py” for interactive visualization of the generated time-series products. “plotts.py” requires a graphical desktop to run. It also requires that you set the matplotlib environment for an X-windows environment. To set matplotlib to run successfully in X-windows, edit this file:

```
> nano ~/.matplotlib/matplotlibrc
```

and set this value:

```
backend : TkAgg
```

Now we are ready to run “plotts.py”. Running the script the “-h” option list all the input parameters that can be controlled from command line.

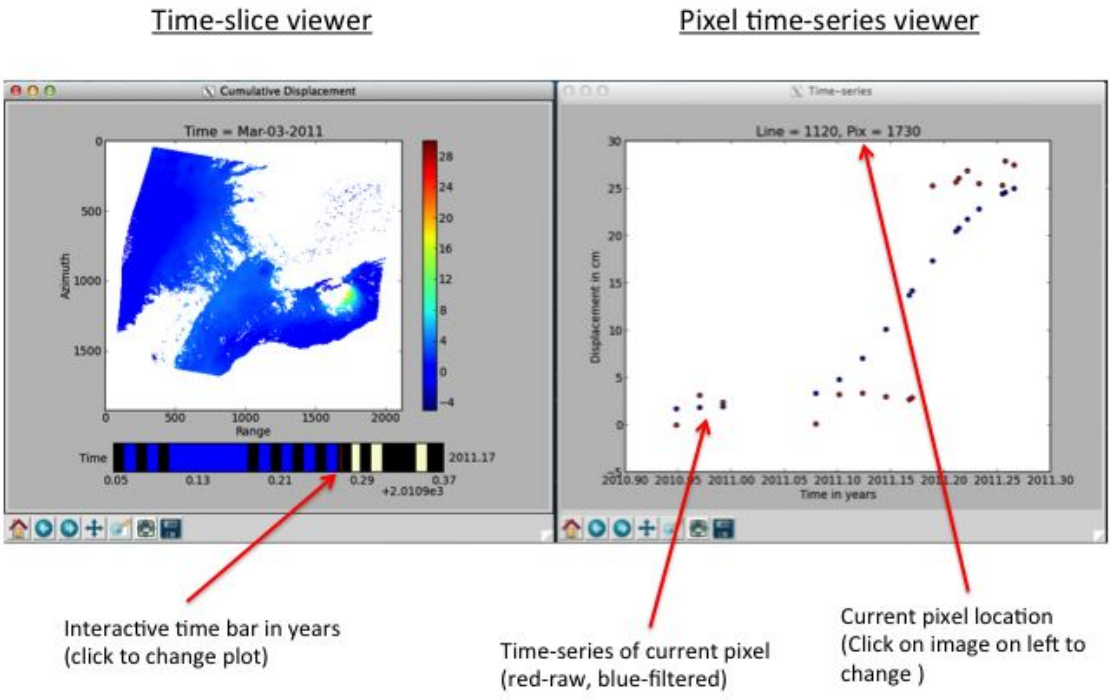
```
> plotts.py -h
logger - INFO - GIANT Toolbox - v 1.0
logger - INFO - ---------------------
usage: plotts.py [-h] [-e] [-f FNAME] [-i TIND] [-m MULT] [-y YLIM
YLIM]
                 [-ms MSIZE] [-raw] [-model] [-mask MASK MASK] [-zf]

Interactive SBAS time-series viewer

optional arguments:
  -h, --help       show this help message and exit
  -e               Display error bars if available. Default: False
  -f FNAME         Filename to use. Default: Stack/LS-PARAMS.h5
  -i TIND          Slice to display. Default: Middle index
  -m MULT          Scaling factor. Default: 0.1 for mm to cm
  -y YLIM YLIM     Y Limits for plotting. Default: [-25,25]
  -ms MSIZE        Marker size. Reduce if error bars are too small.
Default: 5
  -raw             Plot Un-Filtered Time Series as well, if available
  -model           Plot the individual model components as well. For
NSBAS, Timefn and MInTS.
  -mask MASK MASK  To mask out values. Need to provide 2 inputs -
Mask file in
                   float and xml file with dimensions. Default: None
  -zf              Changes time-origin to first acquisition for
```

```
showing time-
                    series.
```

To visualize the output from “NSBASInvert.py”,

```
> plotts.py -f Stack/NSBAS-PARAMS.h5 -y -5 30 -raw
```

This will open two plot windows - an interactive time-slice viewer and a pixel time-series viewer as shown below. The colorbar for the slice viewer ranges from -5 to 30 cm, and the raw time-series (red dots) is also shown along with the filtered time-series (blue); as requested using the command line flags -y and -raw.



Users can now view different time-slices by clicking on the time-bar and can view the time-series for different pixels by directly clicking on the pixel of interest in the image.

Some observations:

1. Our analysis clearly captures the large offset associated with the Mar 6, 2011 dike injection on Kilauea volcano East Rift zone.
2. GIAnT implements a simple Gaussian weighted moving average filter. Hence, the discrepancy between the raw displacement observations (red pixels) and the filtered observations (blue pixels).
3. Users can implement their own custom filtering with the raw time-series included in the HDF5 file.

Besides plotts.py, GIAnT can also export results as a movie through the “make_movie.py” script and as a Google Earth ready KML using “make_kml.py” scripts. For details and usage, refer to the GIAnT user manual. GIAnT also includes tools to export these datasets into GMT’s netcdf format and a GDAL compatible VRT.

CHAPTER 11

Hands On Lab On Polarimetric UAVSAR Data Processing for Land-cover Land-use Change Applications

1. Welcome

Welcome to the **Hands-on Lab on Polarimetric UAVSAR Data Processing for Land Cover / Land Use Change Applications**. Synthetic aperture radar (SAR) is a powerful tool for mapping and monitoring the characteristics of terrestrial landscapes. SAR polarimetry allows us to extract additional information from SAR data as compared to conventional radar backscatter, by separating the scattering mechanisms mixed in the radar return.

The purpose of this tutorial is to introduce basic polarimetric tools that can be used in the context of land cover / land use applications. In the next sessions we will walk you through the steps to:

(1) Obtain the necessary inputs to perform polarimetric analyses with UAVSAR GRD products.
(2) Familiarize yourself with polarimetric decompositions and classification techniques.
(3) Apply free, open source software package PolSARPro to analyze a forest fire dataset.
(4) Understand the advantages, challenges and limitations of working with airborne SAR datasets.

Let's get started and learn more about the data and software used in this practice.

2. The PolSARPro Tool

PolSARPro is a freely available, open-source software package funded by the European Space Agency (ESA) to process polarimetric radar data (http://earth.eo.esa.int/polsarpro/). Although PolSARPro has a basic graphical interface, in this tutorial we will call its tools from the linux command line, which is a highly flexible and scalable approach to process UAVSAR data. We will use the PolSARPro version currently available on the web (v4). A new version of PolSARPro (v5) is being developed by ESA and will be distributed in the near future.

Note:

- ➔ Commands are case sensitive.
- ➔ All relevant commands are preceded by numbers in square brackets (e.g. [23]) and can be copied and pasted into your terminal.
- ➔ Steps to be typed into the Remote Desktop are followed by "RD" (e.g. [23-RD])
- ➔ Steps in blue color are short "do it yourself" assignments.

PolSARPro routines are located in this directory:
`/home/ubuntu/install/polsarpro/Soft`
And PDFs with detailed instructions for each routine are here:
`/home/ubuntu/install/polsarpro/TechDoc/C_Routines`

For this tutorial, all PolSARPro routines are available in your current path. To see a function's usage, just type its name with no arguments.

For example, if you type:
[1] `UTM_LatLong.exe`
You should see:

```
> A processing error occured !
> UTM_LatLong x_coord y_coord UTM_zone
```

The error message can be disregarded as it is just telling us we are calling the routine with no input parameters.

The function `UTM_LatLong` is run by typing its name followed by its 3 arguments separated by spaces: the UTM Easting, UTM Northing, and the UTM zone. Type:
[2] `UTM_LatLong.exe 500000 0  23`
You should see:

```
> longitude = -45.000000    latitude = 0.000000
```

More detailed documentation can be found in the Documentation files.

Next, we will go over a few basic commands as we review the input files for this exercise.

3. Input Files

In the first exercise we will compare pre-fire and post-fire polarimetric UAVSAR images acquired for the 2009 Station Fire (CA). The input UAVSAR data are located in `/home/ubuntu/data/lab10/fire`. Type `cd` to move to the data directory.
[3] `cd /home/ubuntu/data/lab10/fire`

At any time, use the command `pwd` to see your current directory.
[4] `pwd`
`> home/ubuntu/data/lab10/fire`

Use the command `ls` to list the files in a directory.
[5] `ls`

As you can see, the `fire` directory contains 2 directories corresponding to 2 UAVSAR flights: flight_09010 (pre fire: February 26, 2009) and flight_09072 (post fire: September 18, 2009). The other directories will be used to store outputs.

See the files available for the pre-fire flight:
[6] `cd flight_09010`
[7] `ls`
Go up one directory
[8] `cd ../`
Take a look at the post-fire files
[9] `ls flight_09072`

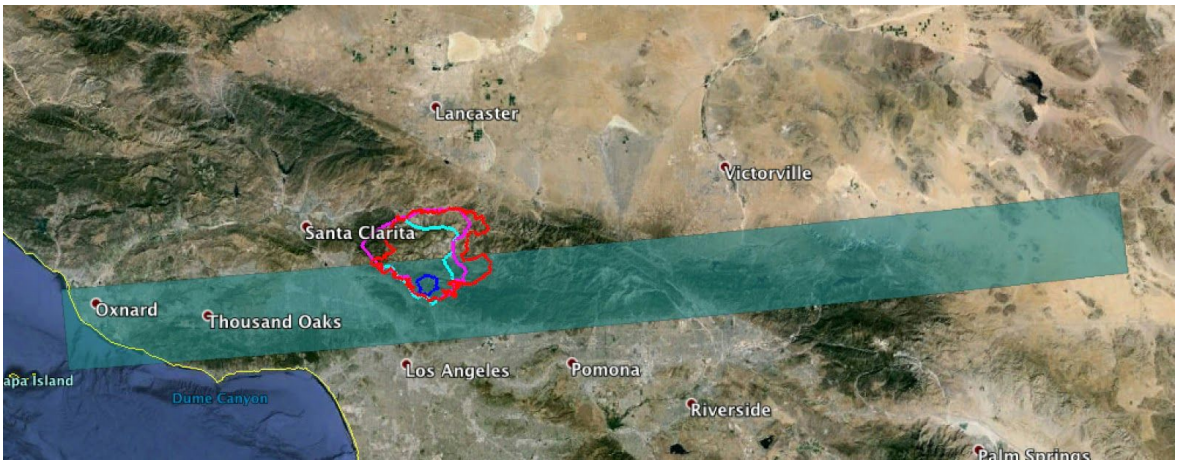
Each flight directory contains:

- ❏ Six ground-projected, polarimetric data files (*.grd).
- ❏ One file containing the local incidence angle in radians for each image pixel (*.inc). We'll use this file to mask out severe topography.
- ❏ One text file containing metadata information such as image dimensions, resolution, and geographic information (*.ann).

More details on file formats:
http://uavsar.jpl.nasa.gov/science/documents/polsar-format.html

In the next page we will see how to read these files within PolSARPro.

The transparent green rectangle represents the area imaged by UAVSAR (297 km long), whereas polygons show the fire progression between August 29 2009 and Sept 01 2009.


Lancaster
Victorville
Santa Clarita
Oxnard
Thousand Oaks
Los Angeles
Pomona
Riverside
Dume Canyon

4. Importing, Cropping, and Multi-looking UAVSAR GRD Images

We will now import the polarimetric UAVSAR GRD images in PolSARPro. As we are interested in a limited region (ROI) around the fire, we will crop the images and also take looks to reduce speckle noise. Importing, cropping and multi-looking can be done in PolSARPro with a single routine, `uavsar_convert_grd_MLK_T3.exe`

Let's look at the routine's usage by typing
[10] `uavsar_convert_grd_MLK_T3.exe`

The arguments are:

- `HeaderFile` = UAVSAR annotation file (*.ann)
- `in_dir` = Input directory
- `out_dir` = Output directory
- `Off_lig` = Offset rows
- `Off_col` = Offset columns
- `Nligfin` = Number of rows in user-defined ROI
- `Ncolfin` = Number of columns in user-defined ROI
- `Nlook_col` = Multilook factor for columns
- `Nlook_lig` = Multilook factor for rows

Two directories have been already created to save the outputs for this step:
`/home/ubuntu/data/lab10/fire/output_09010/T3`
`/home/ubuntu/data/lab10/fire/output_09072/T3`

The command below imports, crops and multi-looks the pre-fire images.
[11] `cd /home/ubuntu/data/lab10/fire`
[12] `uavsar_convert_grd_MLK_T3.exe flight_09010/*ann flight_09010 output_09010/T3 2500 21000 1800 4000 2 2`

This means that starting at pixel position (2500, 21000), we define an area that is 1800 rows x 4000 columns and and average 2x2 pixels in 1 pixel. PolSARPro will crop all polarimetric images contained in the input directory, provided that their names match the provided annotation file. This includes all polarimetric bands and the local incidence angle image.

List output files by typing:
[13] `ls output_09010/T3`
You should see:
`> T11.bin  T12_imag.bin  T12_real.bin  T13_imag.bin  T13_real.bin T22.bin  T23_imag.bin  T23_real.bin  T33.bin  config.txt  dem.bin`

Do it yourself: call `uavsar_convert_grd_MLK.exe` once again to crop the post-fire images from flight 09072. Use the same Region Of Interest as the example above.

The routine `uavsar_convert_grd_MLK_T3.exe` creates the file `config.txt` as well as 9 cropped data (*bin) files. File names reflect the convention used in the polarimetry literature.

File	Description
T11.bin	$\|HH+VV\|^2/2$
T22.bin	$\|HH-VV\|^2/2$
T33.bin	$2*HV^2$
T12_real.bin T12_imag.bin	Complex correlation (HH+VV)(HH-VV)*/2
T13_real.bin T13_imag.bin	Complex correlation (HH+VV)HV*
T23_real.bin T23_imag.bin	Complex correlation (HH-VV)HV*

The “T3” term refers to the coherency matrix used to represent polarimetric data. We could have also used the covariance “C3” matrix.

The output file dem.bin is actually the cropped incidence angle file (this will be fixed in future releases of PolSARPro), so let’s rename it:

```
[14] cd /home/ubuntu/data/lab10/fire/output_09010/T3/
[15] mv dem.bin inc.bin
[16] cd /home/ubuntu/data/lab10/fire/output_09072/T3/
[17] mv dem.bin inc.bin
```

The size of the new images is recorded in the file `config.txt`. Type:

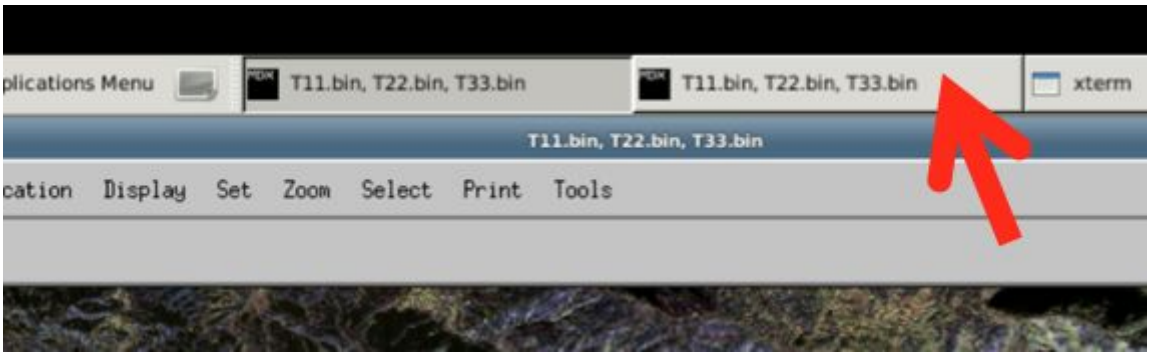
```
[18] cat /home/ubuntu/data/lab10/fire/output_09010/T3/config.txt
[19] cat /home/ubuntu/data/lab10/fire/output_09072/T3/config.txt
```

Make a note of the number of columns and rows (you will need them for the next steps). They should match for the pre-fire and post-fire images.

Let’s look at the cropped backscatter images by launching the Remote Desktop interface. From the Remote Desktop, the command below displays a false color composite **HH+VV 2HV HH-VV** of the pre-fire image:

```
[20-RD] cd /data/lab10
[21-RD] sh mdx_polsar.sh fire_cropped
```

You can switch between pre-fire and post-fire by clicking on the mdx window tab.


plications Menu
T11.bin, T22.bin, T33.bin
T11.bin, T22.bin, T33.bin
xterm
T11.bin, T22.bin, T33.bin
cation Display Set Zoom Select Print Tools

5.Compensation for Polarimetric Orientation Angle

Now we will correct for possible polarimetric distortions induced by azimuth terrain slopes.

We start from this directory:
[22] `cd /home/ubuntu/data/lab10/fire/`

We already have two directories where we'll save the corrected images:
`output_09010/T3_poa`
`output_09072/T3_poa`

Copy the config.txt file into the output `T3_poa` directories:
[23] `cp output_09010/T3/config.txt output_09010/T3_poa`
[24] `cp output_09072/T3/config.txt output_09072/T3_poa`

Check the usage of the orientation angle correction routine:
[25] `orientation_estimation_T3.exe`
`> orientation_estimation_T3 in_dir out_dir offset_lig offset_col sub_nlig sub_ncol`

Apply orientation angle correction to pre-fire image:
[26] `orientation_estimation_T3.exe output_09010/T3 output_09010/T3_poa 0 0 900 2000`

Do it yourself: call `orientation_estimation_T3.exe` to apply the polarimetric orientation angle compensation to post-fire images from flight 09072.

Go to the Remote Desktop and display the pre-fire image before and after the compensation for azimuth distortion.
[27-RD] `sh mdx_polsar.sh fire_poa`

6.Eigenvalue-based Decomposition: H/A/Alpha Decomposition

Let's now apply the H/A/alpha decomposition to the cropped data. First check out the usage:
`> h_a_alpha_decomposition_T3.exe`

These are the input arguments:

`in_dir` = Input directory containing the T3 matrix produced with `orientation_estimation_T3.exe`
`out_dir` = Output directory
`Nwin` = Window size in pixels
`offset_lig` = Offset rows
`offset_col` = Offset columns
`sub_nlig` = Number of rows in input images from the config.txt file
`sub_ncol` = Number of columns in input images from the config.txt file
`alpbetdelgam` = Alpha, Beta, Delta, Gamma, Lambda [1=yes, 0=no]
`lambda` = Lambda component [1=yes, 0=no]
`alpha` = Alpha component [1=yes, 0=no]
`entropy` = Entropy estimate [1=yes, 0=no]
`anisotropy` = Anisotropy [1=yes, 0=no]
The options below refer to combinations of Entropy and Anisotropy components:
`CombHA` = HA [1=yes, 0=no]
`CombH1mA` = H(1-A) [1=yes, 0=no]
`Comb1mHA` = (1-H)A [1=yes, 0=no]
`Comb1mH1mA` = (1-H)(1-A) [1=yes, 0=no]

Then make sure you are in the Station Fire directory:
[28] `cd /home/ubuntu/data/lab10/fire/`

Call the function of the H/A/alpha decomposition on the pre-fire dataset. Note that we only set three output options to 1 in order to get the alpha, entropy, and anisotropy components.
[29] `h_a_alpha_decomposition_T3.exe output_09010/T3_poa output_09010/T3_poa 1 0 0 900 2000 0 0 1 1 1 0 0 0 0`

Do it yourself: apply the H/A/Alpha decomposition to the post-fire dataset.

Go in the Remote Desktop. Let's see what the entropy images look like. We can mask out extreme local incidence angle values < 0.2 radians (11.5 degrees) and > 1.4 radians (80 degrees). Entropy values range between 0 and 1.
[30-RD] `sh mdx_polsar.sh fire_entropy`

Close the entropy mdx images and display the alpha angle images. What differences do you

expect to see when comparing pre and post fire conditions?
[31-RD] `sh mdx_polsar.sh fire_alpha`

Notice results are not influenced by terrain slope, although we still see geometric topographic effects. Alpha values range between 0 deg and 90 deg:

If alpha is close to..	The dominant scattering mechanism is..
0 deg	surface
45 deg	volume
90 deg	double bounce

7.Model-based Decomposition: Van Zyl Decomposition

We will now run the Van Zyl polarimetric decomposition to separate the scattering contribution from the surface, the double bounce and the volume.

Pre-fire:
[32] `cd /home/ubuntu/data/lab10/fire/output_09010`
[33] `vanzyl_3components_decomposition_T3.exe T3_poa/ T3_poa/ 1 0 0 900 2000`

Post-fire:
[34] `cd /home/ubuntu/data/lab10/fire/output_09072`
[35] `vanzyl_3components_decomposition_T3.exe T3_poa/ T3_poa/ 1 0 0 900 2000`

Instead of displaying the RGB images with mdx we compare the histograms of volume component for the fre-fire and the post fire (values in decibels using 100 bins):

Pre-fire:
[36] `cd /home/ubuntu/data/lab10/fire/output_09010/T3_poa`
[37] `echo $[2000*900] > npts.txt`
[38] `statistics_histogram.exe VanZyl3_Vol.bin VanZyl3_Vol_hist.txt npts.txt float db10 100 1 0 0`

Post-fire:
[39] `cd /home/ubuntu/data/lab10/fire/output_09072/T3_poa`
[40] `echo $[2000*900] > npts.txt`
[41] `statistics_histogram.exe VanZyl3_Vol.bin VanZyl3_Vol_hist.txt npts.txt float db10 100 1 0 0`

Now let's plot the histograms of the volume Van Zyl scattering component:
[42-RD] `sh mdx_polsar.sh fire_volhist`

8. Polarimetric Classification

For this exercise we will use an image acquired in Monterey Bay, California. We'll repeat many steps from previous sessions to make an H/A/Alpha decomposition, which will be used here as input to a polarimetric image partitioner.

Go to data directory and create two new directories to save outputs:
[43] `cd /home/ubuntu/data/lab10/monterey`

Let's read in the image and multilook it by a factor of 2. This time we'll read the entire image with no cropping.
[44] `uavsar_convert_grd_MLK_T3.exe input_23025/SanAnd_23025_12030_010_120521_L090_CX_03.ann input_23025/T3 0 0 0 0 2 2`

Compensate for polarimetric orientation angle. The values 7282 and 10033 correspond to the number of rows and columns taken from the config.txt file. Copy the config file to the T3_poa directory.
[45] `orientation_estimation_T3.exe /home/ubuntu/data/lab10/monterey/T3 /home/ubuntu/data/lab10/monterey/T3_poa/ 0 0 7282 10033`
[46] `cp /home/ubuntu/data/lab10/monterey/T3/config.txt /home/ubuntu/data/lab10/monterey/T3_poa`

The H/A/Alpha decomposition serves as input for the classification:
[47] `h_a_alpha_decomposition_T3.exe T3_poa T3_poa 1 0 0 7282 10033 0 0 1 1 1 0 0 0 0`

To partition the image according to the H/Alpha plane, type:
[48] `h_a_alpha_planes_classifier.exe T3_poa T3_poa 0 0 7282 10033 1 0 0 /home/ubuntu/install/polsarpro/Config/Planes_H_A_Alpha_ColorMap9.pal`

This produces a binary classified image H_alpha_class.bmp containing 9 classes, and two quicklook *bmp files: H_alpha_class.bmp and H_alpha_occurence_plane.bmp. Let's open the two *bmp files.

[49-RD] `sh mdx_polsar.sh monterey_class`

CHAPTER 12

Post-Processing UAVSAR Stacks With isceApp.py

1. Post-Processing UAVSAR Stack data with isceApp.py

In this lab, you will learn how to process UAVSAR Stack data, while learning about the ISCE application `isceApp.py`. Previous labs have used the application `insarApp.py` to process pairs of raw or single look complex data acquired on two different dates using spaceborne sensors into interferograms and geocoded products. In this lab we will work with the application `isceApp.py` to post-process several data sets from the same flight track acquired at different times with the UAVSAR radar flown on an airplane. The data downloaded from the UAVSAR website have already been processed to single look complex images by the UAVSAR team. Post-processing includes the following steps: forming interferograms from the slc data, removing topographic phase, filtering, unwrapping, and geocoding.

To get started change directory to the /data/lab11 directory (click the “Launch” button if you haven’t already done so),

```
> cd /data/lab11
```

Take a look at the directory contents with the “ls -l” command,

```
> ls -l
demLat_N38_N39_Lon_W123_W121.dem.wgs84.xml
incoming -> /data/sites/Napa_uavsar_stack/incoming
isceApp.xml
precooked -> /data/sites/Napa_uavsar_stack/
SanAnd_05510_01_BC.dop -> incoming/SanAnd_05510_01_BC.dop
SanAnd_05510_12128_000_121105_L090HH_01_BC.ann ->
incoming/SanAnd_05510_12128_000_121105_L090HH_01_BC.ann
SanAnd_05510_12128_000_121105_L090HH_01_BC_s1_1x1.slc ->
incoming/SanAnd_05510_12128_000_121105_L090HH_01_BC_s1_1x1.slc
SanAnd_05510_13089_001_130508_L090HH_01_BC.ann ->
incoming/SanAnd_05510_13089_001_130508_L090HH_01_BC.ann
SanAnd_05510_13089_001_130508_L090HH_01_BC_s1_1x1.slc ->
incoming/SanAnd_05510_13089_001_130508_L090HH_01_BC_s1_1x1.slc
SanAnd_05510_13165_004_131031_L090HH_01_BC.ann ->
incoming/SanAnd_05510_13165_004_131031_L090HH_01_BC.ann
SanAnd_05510_13165_004_131031_L090HH_01_BC_s1_1x1.slc ->
incoming/SanAnd_05510_14068_000_140529_L090HH_01_BC_s1_1x1.slc
SanAnd_05510_14068_000_140529_L090HH_01_BC.ann ->
incoming/SanAnd_05510_14068_000_140529_L090HH_01_BC.ann
SanAnd_05510_14068_000_140529_L090HH_01_BC_s1_1x1.slc
SanAnd_05510_14128_003_140829_L090HH_01_BC.ann ->
incoming/SanAnd_05510_14128_003_140829_L090HH_01_BC.ann
SanAnd_05510_14128_003_140829_L090HH_01_BC_s1_1x1.slc ->
```

```
incoming/SanAnd_05510_14128_003_140829_L090HH_01_BC_s1_1x1.slc
```

You see that we have prepared this directory with some files and some symbolic links to data. We have provided a symbolic link named "`precooked`" to a directory (indicated with the `->` symbol following its name) where we have previously post-processed a stack of 12 SLCs from the San Andreas fault in the Napa area of California. In the interest of time, in this lab we will illustrate how to post-process these data using a short stack of only a few of those SLCs just before and after the recent earthquake in that area. The full stack will be used in Lab 12 on using GIAnT to create a time series of the deformation. The symbolic links in the current directory ending in "`.ann`" and "`.slc`" contain the meta data and the single look complex data downloaded from the UAVSAR website that we will use in the current lab.

We will start the processing of these data now while going on with the tutorial exposition because it will take a while for the data to process. You can keep an eye on the processing in the terminal pane while continuing to read the tutorial notes in this pane of your web browser window. If you haven't already done so, you can start the remote desktop now to have access to another terminal window and to display images.

To start the processing, you simply enter the following command,

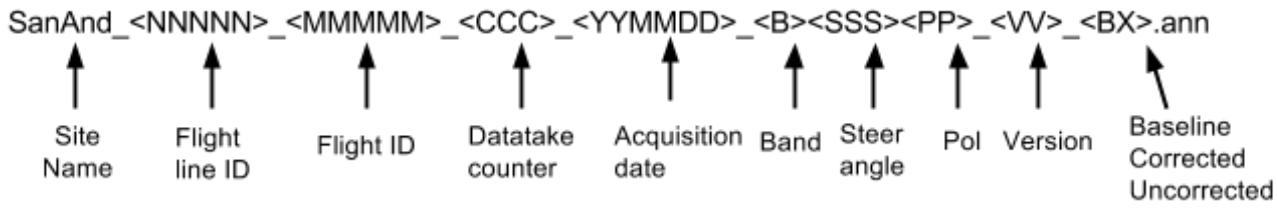
```
> isceApp.py
```

After you launch this command you should see a stream of information going to the terminal window.

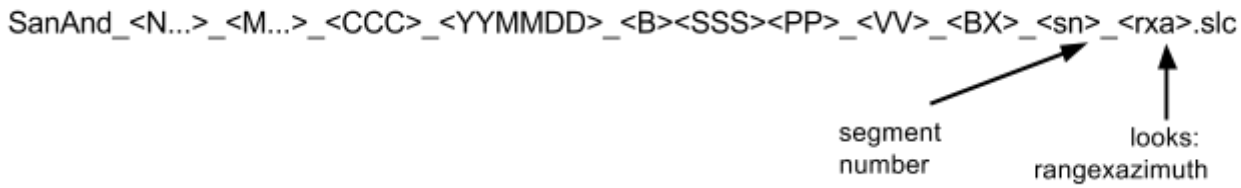
The remaining sections in this lab will explain how to understand the names of the downloaded UAVSAR files, the input files read by ISCE when you entered the command `isceApp.py`, and will lead you through exploring the data products with `mdx.py`.

2. Understanding UAVSAR Data Set Names

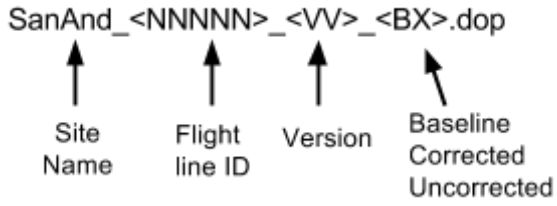
Annotation file:



Single look complex image file:



Doppler file:



The graphic above shows the standard naming conventions for UAVSAR data files. UAVSAR data come with a metadata file called an annotation file with extension "`.ann`", a set of single look complex files with extensions "`.slc`", and a file containing Doppler values as a function of range location with extension "`.dop`".

There is one annotation file for each Flight ID or acquisition date (except in the rare case of multiple flights on one date when only the Flight ID will be different). The SLC data are cut into multiple along track segments within each Flight ID. In this lab we are only working with segment 1 for each Flight ID. The annotation file for each Flight ID contains information on all of the SLC segments. There is one Doppler file for each Flight line. The UAVSAR team processes all SLCs in a Flight ID stack to the same Doppler and the same coordinate system (described by the Peg point contained in the annotation files). All the SLC images in a stack are

either baseline corrected (BC) or baseline uncorrected (BU), depending on whether the residual baseline correction was applied.

3. Understanding the ISCE xml input files for stack processing

When you issued the `isceApp.py` command earlier in Section 1 you may have wondered how `isceApp.py` received input information on what to process and how to set processing options. ISCE will read configuration data from appropriately named files in the local directory. For the application, `isceApp.py` the name can either be `isce.xml` or `isceApp.xml`. In fact you could have two files using both of these names with either complimentary or conflicting information. In the case of complimentary information the union of the information in the two files will be used. In the case of conflicting information the information in the file named `isceApp.xml` will win. It is also possible to name an input file on the command line with any name desired (as was done in some of the earlier labs); the file on the command line will win in the case of conflicting information amongst the different configuration files. You can issue the following two commands to see that we have placed a file with name `isceApp.xml` in the directory, but not one with name `isce.xml`.

```
> ls isceApp.xml
isceApp.xml
> ls isce.xml
ls: cannot access isce.xml: No such file or directory
```

When we run `isceApp.py` ISCE loads the contents of the `isceApp.xml` file found in the local directory and configures the application.

Let's look at the contents of the input file,

```
> cat isceApp.xml
<?xml version="1.0" encoding="UTF-8"?>

<isceApp>
  <component name="isce">
      <property name="sensor name">UAVSAR_Stack</property>
      <property name="resamp range looks">   6</property>
      <property name="resamp azimuth looks">16</property>
      <property name="do unwrap">True</property>
      <property name="unwrapper name">icu</property>
      <property name="output directory">.</property>

      <component name="stack">
<!--
      <component name="Scene1">
```

```
<property name="id">uav1</property>
<property name="hh">
            SanAnd_05510_09006_011_090218_L090HH_01_BC.ann
      </property>
</component>

<component name="Scene2">
<property name="id">uav2</property>
<property name="hh">
         SanAnd_05510_09091_005_091117_L090HH_01_BC.ann
      </property>
</component>

<component name="Scene3">
<property name="id">uav3</property>
<property name="hh">
          SanAnd_05510_10037_009_100511_L090HH_01_BC.ann
      </property>
</component>

<component name="Scene4">
<property name="id">uav4</property>
<property name="hh">
         SanAnd_05510_10077_010_101028_L090HH_01_BC.ann
      </property>
</component>

<component name="Scene5">
<property name="id">uav5</property>
<property name="hh">
          SanAnd_05510_11049_008_110713_L090HH_01_BC.ann
      </property>
</component>

<component name="Scene6">
<property name="id">uav6</property>
<property name="hh">
          SanAnd_05510_11071_012_111103_L090HH_01_BC.ann
      </property>
</component>

<component name="Scene7">
<property name="id">uav7</property>
```

```
      <property name="hh">
                  SanAnd_05510_12017_007_120418_L090HH_01_BC.ann
            </property>
      </component>

      <component name="Scene8">
      <property name="id">uav8</property>
      <property name="hh">
                SanAnd_05510_12128_000_121105_L090HH_01_BC.ann
            </property>
      </component>
-->

      <component name="Scene9">
      <property name="id">uav9</property>
      <property name="hh">
                SanAnd_05510_13089_001_130508_L090HH_01_BC.ann
            </property>
      </component>

      <component name="Scene10">
      <property name="id">uav10</property>
      <property name="hh">
               SanAnd_05510_13165_004_131031_L090HH_01_BC.ann
           </property>
      </component>

      <component name="Scene11">
      <property name="id">uav11</property>
      <property name="hh">
                SanAnd_05510_14068_000_140529_L090HH_01_BC.ann
            </property>
      </component>

      <component name="Scene12">
      <property name="id">uav12</property>
      <property name="hh">
               SanAnd_05510_14128_003_140829_L090HH_01_BC.ann
            </property>
      </component>
      </component>

      <property name="selectPols">hh</property>
```

```
      <property
name="selectPairs">uav10-uav12,uav9/uav11,uav9/uav12</property>

      <property name="coregistration strategy">single
reference</property>
            <property name="reference scene">uav9</property>
      <property name="reference polarization">hh</property>

      <property name="geocode list">
          ["filt_topophase.flat", "filt_topophase.unw"]
      </property>

      <component name="dem">
          <catalog name="dem">
                    demLat_N38_N39_Lon_W123_W121.dem.wgs84.xml
               </catalog>
      </component>

  </component>
</isceApp>
```

This is an edited version of the file that was used in post-processing the data in the “precooked” directory. The file lets isceApp.py know that the “sensor name” is UAVSAR_Stack so that the appropriate code for handling the sensor meta data and image data will be used. It sets a few processing options such as the number of range and azimuth looks and the name of the unwrapper to use. Then it has a “component” section giving the names of the input annotation files for each image in the stack. In the precooked directory we used 12 input images. In this directory we have commented out (xml comment begins with “<!--” and ends with “-->”) all of the scenes except 9-12, the ones just before and after the recent Napa earthquake. After the scenes list in the stack, the reference scene is selected and the pairs to be post-processed are selected. The specification of the pairs using a “/” symbol simply means to form a single pair from the two named scenes. So, `uav9/uav11` means form an interferogram from the SLC labelled `uav9` and the SLC labelled `uav11` in the definition of the stack. The “-” symbol between two scenes is shorthand for all possible unique combinations between the first and last scene. So, `uav10-uav12` would expand into `uav10/uav11, uav10/uav12, uav11/uav12`.

The `isceApp.xml` gives the name of the annotation files provided by the UAVSAR project. The annotation file contains the names of the other inputs files used by ISCE, namely the single look complex (SLC) file and the doppler file. The annotation file is an rdf (radar data format) file, which is a text file consisting of lines of the form “keyword (units) = value”. Some example lines used in ISCE from one of the annotation files in this directory follows (`;` starts a comment,

& indicates a string value, – indicates a dimensionless entry),

```
slc_1_1x1 (&) = SanAnd_05510_11071_012_111103_L090HH_01_BC_s1_1x1.slc
dop       (&) = SanAnd_05510_01_BC.dop
slc_1_1x1 Columns   (pixels)  = 9900   ;samples in SLC 1x1 segment 1
slc_1_1x1 Rows      (pixels)  = 66664  ;lines in SLC 1x1 segment 1

1x1 SLC Range Pixel Spacing        (m) = 1.66551366
1x1 SLC Azimuth Pixel Spacing      (m) = 0.6
Global Average Altitude            (m) = 12495.755
Global Average Terrain Height      (m) = 4.61314579
Average Pulse Repetition Interval (ms) = 2.24897112
Peg Latitude                     (deg) =   38.2206738
Peg Longitude                    (deg) = -121.928086
Peg Heading                      (deg) =   55.2680562
Ellipsoid Semi-major Axis          (m) = 6378137.0
Ellipsoid Eccentricity Squared     (-) = 0.00669438
Segment 1 Data Starting Azimuth    (m) = -32011.8
Image Starting Slant Range        (km) = 13.4489379 ;for SLC 1x1 data
Average Along Track Velocity     (m/s) = 246.759825
Minimum Look Angle               (deg) = 21.54218775
Maximum Look Angle               (deg) = 65.29939711
Look Direction                     (&) = Left
Antenna Length                     (m) = 1.5
Polarization                       (&) = HH
Center Wavelength                 (cm) = 23.8403545
Bandwidth                        (MHz) = 80.0
Pulse Length                (microsec) = 40.0
Start Time of Acquisition          (&) =  3-Nov-2011 22:39:41 UTC
Stop Time of Acquisition           (&) =  3-Nov-2011 22:44:22 UTC
```

4. Understanding the output products

After isceApp.py has finished running, take a look at the list of directories and files in the current directory,

```
> ls
catalog
demLat_N38_N39_Lon_W123_W121.dem.wgs84.xml
dop.txt
incoming
isceApp.xml
isce.log
isceProc_20141017013039.xml
precooked
SanAnd_05510_01_BC.dop
SanAnd_05510_12128_000_121105_L090HH_01_BC.ann
SanAnd_05510_12128_000_121105_L090HH_01_BC_s1_1x1.slc
SanAnd_05510_13089_001_130508_L090HH_01_BC.ann
SanAnd_05510_13089_001_130508_L090HH_01_BC_s1_1x1.slc
SanAnd_05510_13089_001_130508_L090HH_01_BC_s1_1x1.slc.xml
SanAnd_05510_13165_004_131031_L090HH_01_BC.ann
SanAnd_05510_13165_004_131031_L090HH_01_BC_s1_1x1.slc
SanAnd_05510_13165_004_131031_L090HH_01_BC_s1_1x1.slc.xml
SanAnd_05510_14068_000_140529_L090HH_01_BC.ann
SanAnd_05510_14068_000_140529_L090HH_01_BC_s1_1x1.slc
SanAnd_05510_14068_000_140529_L090HH_01_BC_s1_1x1.slc.xml
SanAnd_05510_14128_003_140829_L090HH_01_BC.ann
SanAnd_05510_14128_003_140829_L090HH_01_BC_s1_1x1.slc
SanAnd_05510_14128_003_140829_L090HH_01_BC_s1_1x1.slc.xml
uav10
uav10__uav11
uav10__uav12
uav11
uav11__uav12
uav12
uav9
uav9__uav10
uav9__uav11
uav9__uav12
```

Starting from the top of this listing we see a new directory named `catalog`, which contains several text files with data indicating the state of several objects such as the orbit during the processing flow. These may be useful in debugging if something doesn't seem to work properly.

The next new file is the `dop.txt` file, which is a text file containing the input doppler samples from the UAVSAR input products and samples from a polynomial fit to these data done by `isceApp.py`.

The next new file is `isce.log`, which is a text file with logging information from the run of `isceApp.py`. Then there is the file `isceProc_<date-time>.xml` that contains detailed information relevant to the information used in post-processing the data (provenance), including the version of ISCE that was run, the input parameters, and much more. You can use list the contents of the file using the commands `'more'` or `'less'` or `'cat'`.

Next you see a block of files that were originally in the directory with new `slc.xml` files created by `isceApp.py` interspersed in the listing. These files contain useful metadata for the slc files that can be used in displaying those images with the command, `mdx.py`.

Then there are several directories named for the labels of the scenes (`uav9` for example) and the pairs indicated in the input file (*eg.*, `uav9__uav12`), `isceApp.xml`. Use the `'ls'` command to view the contents of those directories.

The reference scene (defined in the xml file) directory,

```
> ls uav9
uav9_hh.raw     uav9.lon.rdr    uav9.los.rdr.xml      uav9.z.rdr
uav9.zsch.rdr.xml
uav9.lat.rdr    uav9.lon.rdr.xml  uav9.simamp.rdr     uav9.z.rdr.xml
uav9.lat.rdr.xml  uav9.los.rdr        uav9.simamp.rdr.xml
uav9.zsch.rdr
```

The file with the extension `.raw` is not actually a raw unfocused file. It is a placeholder for a file expected in the normal flow of `isceApp.py`, which is actually just a symbolic link to the slc file. The `.rdr` files contain the digital elevation model (DEM) lattitude, longitude, and heights resampled in the radar coordinate system, a computed line of sight file to each output pixel in `los.rdr`, and a simulated ampitude file from the DEM in `simamp.rdr`.

One of the pair directories,

```
> ls uav9__uav12
uav9__uav12_hh.dem.crop
uav9__uav12_hh.resampImage.amp
uav9__uav12_hh.dem.crop.xml
uav9__uav12_hh.resampImage.amp.xml
```

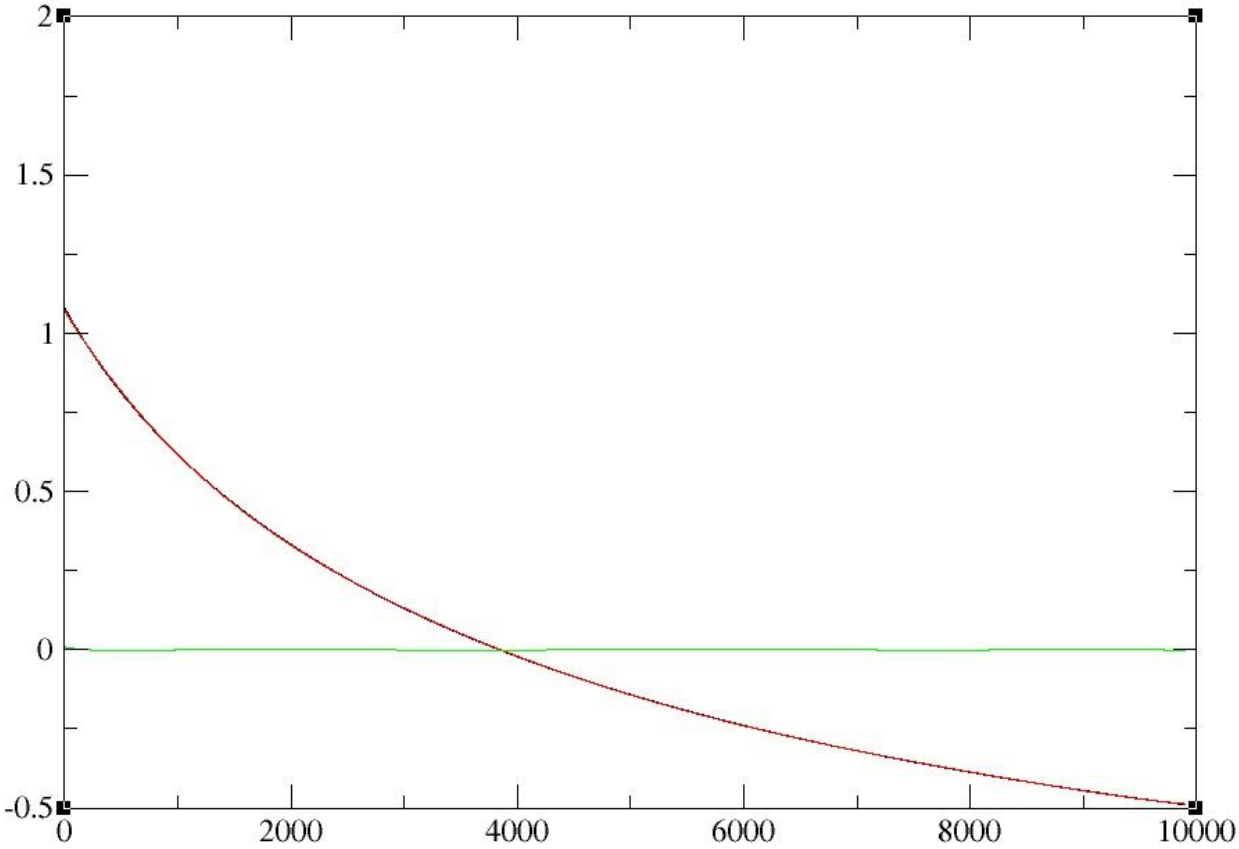
```
uav9__uav12_hh.filt_topophase.conncomp
uav9__uav12_hh.resampImage.int
uav9__uav12_hh.filt_topophase.conncomp.xml
uav9__uav12_hh.resampImage.int.xml
uav9__uav12_hh.filt_topophase.flat
uav9__uav12_hh.resampOnlyImage.amp
uav9__uav12_hh.filt_topophase.flat.geo
uav9__uav12_hh.resampOnlyImage.amp.xml
uav9__uav12_hh.filt_topophase.flat.geo.xml
uav9__uav12_hh.resampOnlyImage.int
uav9__uav12_hh.filt_topophase.flat.xml
uav9__uav12_hh.resampOnlyImage.int.xml
uav9__uav12_hh.filt_topophase.unw
uav9__uav12_hh.topophase.cor
uav9__uav12_hh.filt_topophase.unw.geo
uav9__uav12_hh.topophase.cor.xml
uav9__uav12_hh.filt_topophase.unw.geo.xml
uav9__uav12_hh.topophase.flat
uav9__uav12_hh.filt_topophase.unw.xml
uav9__uav12_hh.topophase.flat.xml
uav9__uav12_hh.geo.log
uav9__uav12_hh.topophase.mph
uav9__uav12_hh.phsig.cor
uav9__uav12_hh.topophase.mph.xml
uav9__uav12_hh.phsig.cor.xml
>
```

5. Visualizing the output products

You can look at a plot of the Doppler centroid as a function of range across the UAVSAR swath and the polynomial fit to it used in ISCE by using the following command in your remote desktop,

```
> cd /data/lab11
> xmgrace -nxy dop.txt
```

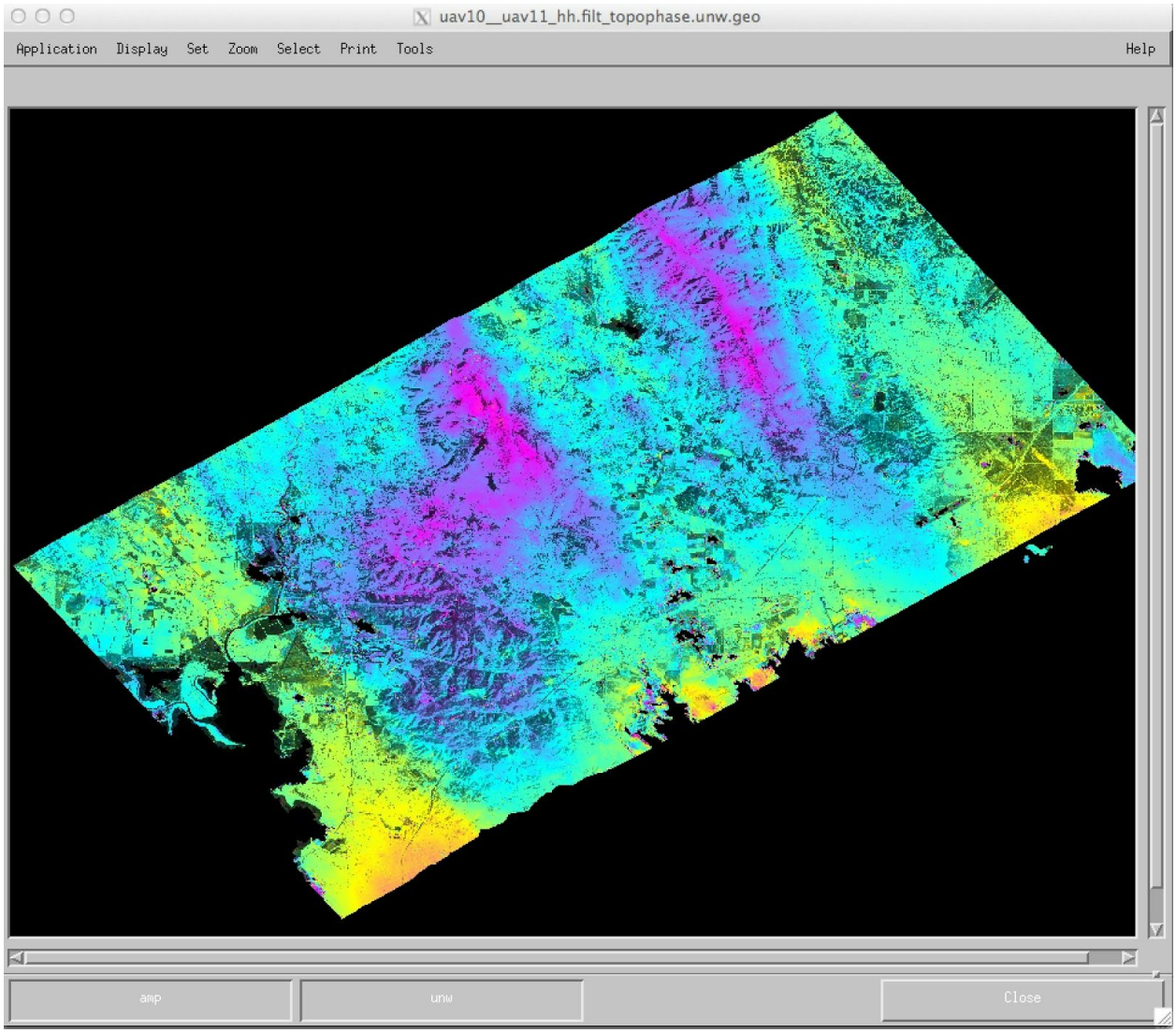
The abcissa is the range bin and the ordinate is the Doppler value normalized by the PRF.



There is a black curve, which is the Doppler data from the UAVSAR project input file, `SanAnd_05510_01_BC.dop`, which is hidden behind the red curve, which is the polynomial fit. The green curve is the residuals from the fit.

For those who are familiar with satellite SAR data, notice that there is a strong variation of the Doppler centroid across the UAVSAR swath caused by the large average yaw or squint of this

stack (-1.9 degrees). You can check the yaw of the stack provided in the annotation file with this command:




```
grep Yaw SanAnd_05510_13089_001_130508_L090HH_01_BC.ann
Global Average Yaw                              (deg) = -1.90762926
```

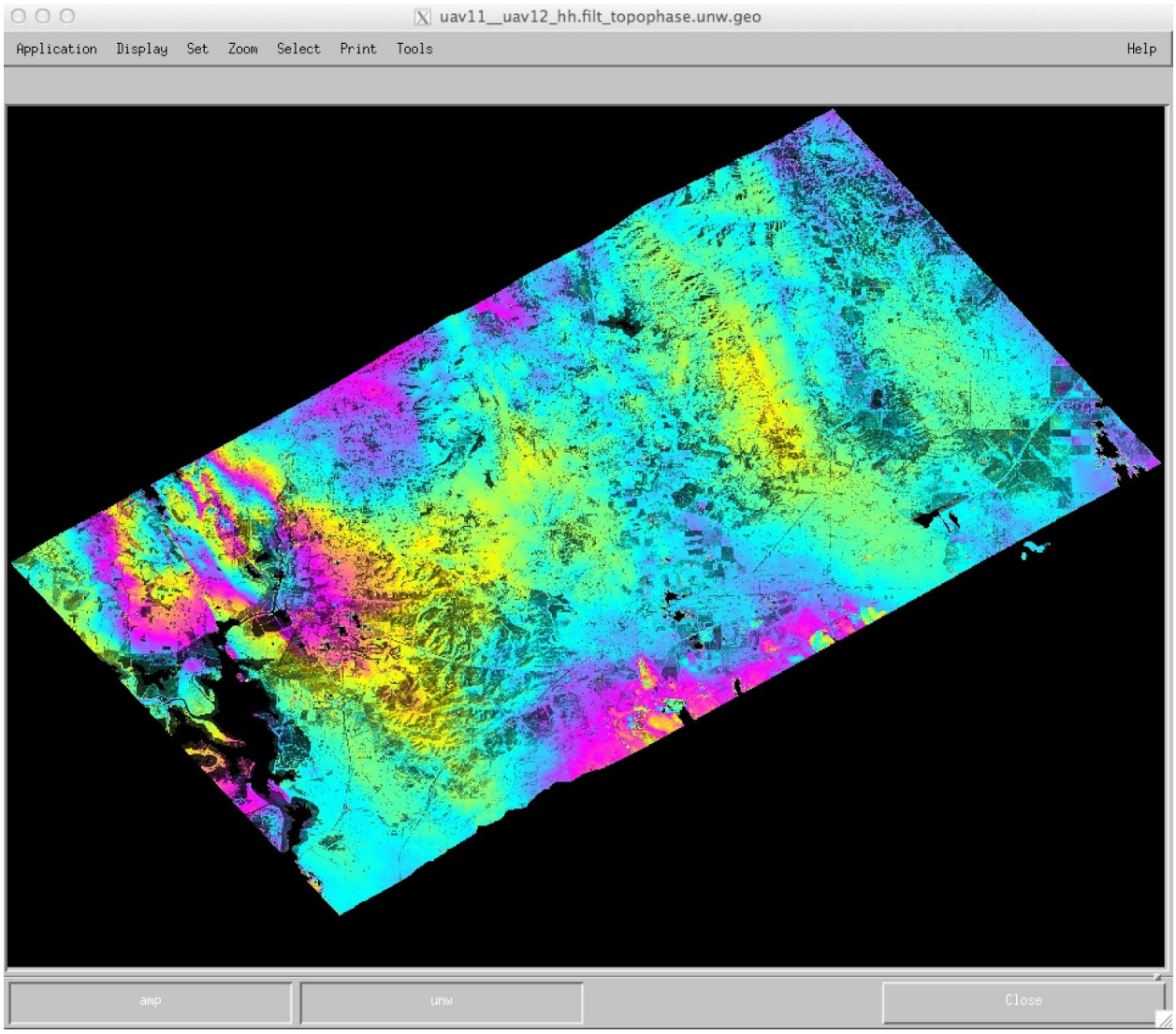
You can use `mdx.py` to visualize any of the output images. For example, the filtered, unwrapped, topophase corrected, geocoded interferogram for a pair before the earthquake,

```
> mdx.py uav10__uav11/uav10__uav11_hh.filt_topophase.unw.geo -z -2
```

Note that some of the phase in this interferogram before the earthquake (waves running across the swath) is likely due to aircraft motion that was not fully corrected, because this stack was processed without the residual baseline correction.

Similarly for a pair spanning the earthquake,

```
> mdx.py uav11__uav12/uav11__uav12_hh.filt_topophase.unw.geo -z -2
```



The August 24, 2014 M6.0 South Napa Earthquake caused the strong fringes and the sharp phase discontinuity near the west end of this UAVSAR interferogram where the fault ruptured the surface. The default phase unwrapping method was not able to estimate the large offset across the surface rupture, which is about 20 cm or nearly two fringes.

6. Notes about component configurability

Lab 7 illustrated a few of the component configurability options for controlling the processing of COSMO-SkyMed data. In the current lab we did not tell you earlier but we have used a feature of component configurability to provide additional input information to the processsing job. We have set certain global preferences for processing the UAVSAR data with `isceApp.py` so that the input `isceApp.xml` file we showed you in Section 3 was as simple and as specific to the current data as possible. ISCE uses an environment variable $ISCEDB to locate these global preferences. If you type the following commands you will see the name and contents of the directory that ISCE looks in for these global preferences,

```
> echo $ISCEDB
/home/ubuntu/.iscedb

>ls $ISCEDB
insar.xml  isce.xml
```

When you run an application the ISCE framework looks in this directory to see if any file is named appropriately for configuring an ISCE component or applications. The `isceApp.py` application can be configured with a file named either `isce.xml` or `isceApp.xml` (or both with the one named `isceApp.xml` having higher priority if there are conflicting information for any properties contatined in the two files). When configuring isceApp.py the global preferences can be overridden by settings in the input files in the processing directory or on the cammand line.

Let's look at the contents of the global preferences file for `isceApp.py`,

```
> cat ~/.iscedb/isce.xml
<?xml version="1.0" encoding="UTF-8"?>

<!-- NOTE: tag/attribute names must be in lower case -->
<isceApp>
  <component name="isce">
      <component name="stack">
      <component name="Raster1">
          <property name="ncol">1500</property>
          <property name="nlin">4000</property>
          <property name="datatype">float</property>
      </component>
      </component>
      <property name="doppler method">usedefault</property>
      <!-- Processors to run: True/False -->
```

```
      <property name="do preprocess">True</property>
      <property name="do verifyDEM">True</property>
      <property name="do pulsetiming">True</property>
      <property name="do estimateheights">True</property>
      <property name="do mocomppath">True</property>
      <property name="do orbit2sch">True</property>
      <property name="do updatepreprocinfo">True</property>
      <property name="do formslc">True</property>
      <property name="do multilookslc">True</property>
      <property name="do filterslc">False</property>
      <property name="do filter interferogram">True</property>
      <property name="do polarimetric correction">False</property>
      <property name="do calculate FR">False</property>
      <property name="do FR to TEC">False</property>
      <property name="do TEC to phase">False</property>
      <property name="do offsetprf">False</property>-->
      <property name="do outliers1">False</property>-->
      <property name="do prepareresamps">True</property>-->
      <property name="do resamp">False</property>-->
      <property name="do resamp image">False</property>-->
      <property name="do crossmul">True</property>
      <property name="do mocomp baseline">True</property>
      <property name="do set topoint1">True</property>
      <property name="do topo">True</property>
      <property name="do shadecpx2rg">True</property>
      <property name="do rgoffset">False</property>
      <property name="do rg outliers2">True</property>
      <property name="do resamp only">True</property>
      <property name="do set topoint2">True</property>
      <property name="do correct">True</property>
      <property name="do coherence">True</property>
      <property name="do unwrap">False</property>
      <property name="do geocode">True</property>
  </component>
</isceApp>
```

Most of the contents of this file tell isceApp.py whether to do a particular step in its flow. The False settings in this file are not really optional for processing UAVSAR stack data. The isceApp.py application is general enough to work with many different types of sensors, so it is necessary that it knows which steps to do for this particular sensor. The UAVSAR stack processing is relatively new and in future updates to ISCE would automatically set these options as the defaults when processing data from this sensor.

CHAPTER 13

GIAnT with UAVSAR Stacks

1. Intro - Preparing the ISCE stack for analysis

Note: Execute all the commands in this lab session on a terminal in the Guacamole interface.

GIAnT is designed to work with outputs from multiple SAR/InSAR processors -e.g, ISCE, ROI_PAC etc. The very first stage of processing with GIAnT transforms InSAR products from their native formats (e.g, ISCE's binary files, GMTSAR's grd files etc) to an internally consistent Hierarchical Data Format 5 (HDF5) format.

In this tutorial, we will describe the steps involved in transforming all the input data (described in the previous tutorial) into a HDF5 format needed by GIAnT. Again, we start with our test dataset located in the directory "synthetic":

```
> cd /home/ubuntu/data/giant/napa/GIAnT
> ls
example.rsc  map.json        prepsbasxml.py
ifg.list     prepdataxml.py  userfn.py
```

From amongst the various python scripts in the directory - "userfn.py" and "prepdataxml.py" are needed for preparing our data stack for analysis. The other python scripts are related to the actual time-series analysis and will be discussed in Lab 12.2.

2. userfn.py - Translating pair information to actual files on disk

As described in the previous tutorial, “ifg.list” is a four column text file that describes our interferogram network in a simple fashion.

```
> less ifg.list
20090218   20091117   0.0 UAVSAR
20090218   20100511   0.0 UAVSAR
20090218   20101028   0.0 UAVSAR
20091117   20100511   0.0 UAVSAR
20091117   20101028   0.0 UAVSAR
...
```

We also mentioned that we stored our unwrapped phase and coherence files in individual sub-directories in a directory named “insar”. But we never provided the exact mapping between each line of “ifg.list” and the corresponding files in “insar”. This is accomplished through userfn.py .

```
>less userfn.py
...
def makefnames(dates1, dates2, sensor):
    dirname = '../insar'
    key1 = date2key(dates1)
    key2 = date2key(dates2)

    pre = key1+'__'+key2+'_hh.'
    root = os.path.join(dirname, key1+'__'+key2)
    iname = os.path.join(root, pre+'filt_topophase.unw')
    cname = os.path.join(root, pre+'phsig.cor')
    return iname, cname
```

“userfn.py” should define a function named “makefnames” that takes the the master date, slave date and sensor name as inputs and returns two strings that represent the path to the unwrapped phase file and the coherence file. “userfn.py” should be located in your working directory.

This particular mechanism was devised to allow users to store InSAR outputs using their preferred directory and file name structure. Note that “userfn.py” should be considered as an user input, and each stack should be accompanied by its own “userfn.py”.

3. userfn.py and map.json

GIAnT only insists on the existence a function named “makefnames” within the script “userfn.py”. This function can in turn invoke other functions from other python libraries or look up databases to construct filenames. In this case, it invokes the “date2key” function which has also been defined in the same script.

```
> less userfn.py
...
def date2key(indate):
    fid = open('map.json', 'r')
    keymap = json.load(fid)
    fid.close()

    instr = str(indate)

    for key,value in keymap.items():
        if instr in value:
            return key

    raise Exception('Date not found')
    return
…
```

This is a simple python function that loads “map.json” and accepts a datestring in yyyymmdd format to return the corresponding user-defined ISCE key.

4. “prepdataxml.py” - setting up data properties.

“prepdataxml.py” is responsible for generating the input file “data.xml” which describes the characteristics of the dataset like dimensions, looks, formats etc.

```
> cd /home/ubuntu/data/giant/napa/GIAnT
> less prepdataxml.py
#!/usr/bin/env python

import tsinsar as ts
import argparse
import numpy as np

if __name__ == '__main__':

    ######Prepare the data.xml
    g = ts.TSXML('data')
    g.prepare_data_xml('example.rsc',
                       xlim=[0,1650], ylim=[0, 4165],
                       rxlim = [745,755], rylim=[3595,3605],
                       latfile='', lonfile='', hgtfile='',
                       inc = 21., cohth=0.4, chgendian='False',
                       unwfmt='RMG', corfmt='FLT')
    g.writexml('data.xml')
```

We set up some basic parameters for processing our stack using “prepdataxml.py”. The complete list of all configurable parameters can be found in the GIAnT user manual. We describe the parameters that we have set up using prepdataxml.py below:

example.rsc	ROI_PAC style resource file with minimum common metadata.
xlim	X limits for cropping the image (Python convention). We use the full image here.
ylim	Y limits for cropping the image (Python convention). We use the full image here.
rxlim	X limits of reference region. Pixel 30-49 in range. (zero index)
rylim	Y limits of referenec region. Line 50-69 in azimuth. (zero

	index)
latfile, lonfile, hgtfile	Files for lat, lon, height in radar coordinates. This information is needed for atmospheric corrections, which are currently not used. These are described in the tutorial on advanced topics.
inc	Incidence angle (constant or file). Again only use for atmospheric corrections and GPS comparison. Not used in this tutorial.
cohth	Coherence threshold. All phase measurements with coherence less than this value are considered invalid.
chgendian	To the input files are in a different format than the native machine format.
unwfmt	FLT/RMG to indicate that the input is one or two channel file.
corfmt	FLT/RMG to indicate that the input is one or two channel file.

The default data type for all files is float32. See GIAnT user manual for complete list of options and default values.

We will then generate our “data.xml” script as follows:

```
> python prepdataxml.py
```

To view the generated “data.xml” file,

```
> less data.xml
```

```
<data>
  <proc>
    <value>RPAC</value>
    <type>STR</type>
    <help>Processor used for generating the interferograms.</help>
  </proc>
  <master>
    <width>
      <value>1650</value>
      <type>INT</type>
      <help>WIDTH of the IFGs to be read in.</help>
```

```
    </width>
    <file_length>
      <value>4165</value>
      <type>INT</type>
      <help>FILE_LENGTH of the IFGS to be read in.</help>
    </file_length>
    <wavelength>
      <value>0.238403545</value>
      <type>FLOAT</type>
      <help>WAVELENGTH of the Stack. If combining sensors,ensure that
they are all converted to same units.</help>
    </wavelength>
...
```

Note that the generated XML file can be modified in a text editor, and we include a help string to describe each of the parameters in the file.

We are now ready to gather data into a HDF5 file readable by GIAnT.

4. PrepIgramStack.py - preparing the stack

From the GIAnT working directory, execute PrepIgramStack.py.

(NOTE: The PrepIgramStack.py command and many of the rest in this lab need to be run from the X11 windows in the Remote Desktop function of EarthKit.)

```
> cd /home/ubuntu/data/giant/napa/GIAnT

> PrepIgramStack.py
<module> - INFO - Number of interferograms = 30
<module> - INFO - Number of unique SAR scenes = 12
<module> - INFO - Number of connected components in network: 1
<module> - INFO - No common mask defined
<module> - INFO - Output h5file: Stack/RAW-STACK.h5
<module> - INFO - PNG preview dir: Figs/Igrams
<module> - INFO - Reading in IFGs
[========================= 59% =>                    ]    134s / 93s
```

As indicated by the screen output, the program generates a file named “Stack/RAW-STACK.h5” in the Stack directory and another directory called “Figs/Igrams”.

```
> ls
data.xml     Figs      prepdataxml.py  Stack      userfn.pyc
example.xml  ifg.list  prepsbasxml.py  userfn.py

> ls Stack
RAW-STACK.h5

> ls Figs
Igrams

> ls Figs/Igrams

...
```

HDF5 outputs of all GIAnT programs are stored in the “Stack” directory and associated PNG previews are generated in a directory named “Figs”.

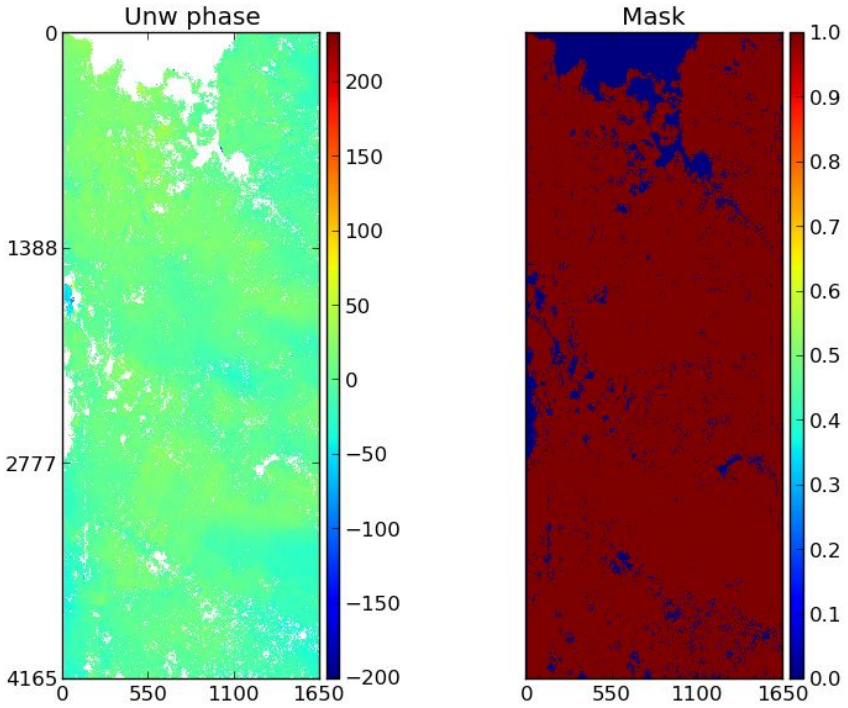
5. PNG previews - What does our data look like?

The directory Figs/Igrams contains PNG previews of all unwrapped interferograms listed in ifg.list . The PNG files are numbered in sequence. The PNG preview corresponding to the 80th interferogram in our test data set.

To preview the PNG files, run the following command: (NOTE: you will need to run this image preview command from the Remote Desktop as it is graphical in nature)

```
> cd Figs/Igrams
> eog *.png
```

Notice that a coherence threshold has been applied to the interferograms depending on the user inputs in data.xml. The unwrapped phase has been converted to mm at this stage.



6. Listing contents of RAW-STACK.h5

In this section, we will try to understand the structure of the HDF5 file `Stack/RAW-STACK.h5` created by “PrepIgramStack.py”. We can summarize the contents of this file using `h5ls`

```
> cd /home/ubuntu/data/giant/napa/GIAnT
> h5ls Stack/RAW-STACK.h5
Jmat                     Dataset {30, 12}
bperp                    Dataset {30}
cmask                    Dataset {4165, 1650}
dates                    Dataset {12}
igram                    Dataset {30, 4165, 1650}
tims                     Dataset {12}
usat                     Dataset {12}
```

This lists the various arrays stored in the HDF5 file and their corresponding sizes.

HDF5 datasets are compatible with “gdal” and you can use gdalinfo to display help information about each dataset.

```
> gdalinfo Stack/RAW-STACK.h5
Driver: HDF5/Hierarchical Data Format Release 5
Files: RAW-STACK.h5
Size is 512, 512
Coordinate System is `'
Metadata:
  bperp_help=Array of baseline values.
  cmask_help=Common mask for pixels.
  dates_help=Ordinal values of SAR acquisition dates.
  help=All the raw data read from individual interferograms into a
single location for fast access.
  igram_help=Unwrapped IFGs read straight from files.
  Jmat_help=Connectivity matrix [-1,1,0]
  tims_help= Array of SAR acquisition times.
Subdatasets:
  SUBDATASET_1_NAME=HDF5:"RAW-STACK.h5"://Jmat
```

```
  SUBDATASET_1_DESC=[30x12] //Jmat (64-bit floating-point)
  SUBDATASET_2_NAME=HDF5:"RAW-STACK.h5"://cmask
  SUBDATASET_2_DESC=[4165x1650] //cmask (64-bit floating-point)
  SUBDATASET_3_NAME=HDF5:"RAW-STACK.h5"://igram
  SUBDATASET_3_DESC=[30x4165x1650] //igram (32-bit floating-point)
Corner Coordinates:
Upper Left  (    0.0,    0.0)
Lower Left  (    0.0,  512.0)
Upper Right (  512.0,    0.0)
Lower Right (  512.0,  512.0)
Center      (  256.0,  256.0)
```

Every HDF5 dataset created by GIAnT includes a self-explanatory “help” attribute which is listed in the “Metadata” section of the output from the `gdalinfo` command.

RAW-STACK.h5 has all the data we need to proceed to the next stage of time-series processing, stored in a convenient and easily accessible format.

1. Quick recap

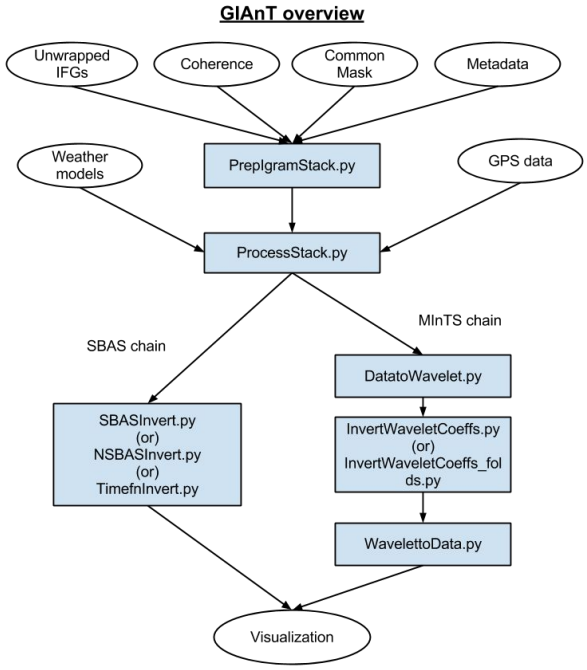
So far, we have gathered all the required network, unwrapped phase and coherence information into a HDF5 file using “PrepIgramStack.py” in the previous tutorial.

```
> cd /home/ubuntu/data/giant/napa/GIAnT
> h5ls Stack/RAW-STACK.h5
Jmat                     Dataset {30, 12}
bperp                    Dataset {30}
cmask                    Dataset {4165, 1650}
dates                    Dataset {12}
igram                    Dataset {30, 4165, 1650}
tims                     Dataset {12}
usat                     Dataset {12}
```

In this tutorial, we will apply optional corrections to the ingested stack and estimate the deformation time-series using the SBAS technique. We will also teach users to interactively visualize some of the time-series results.

2. Setting up the processing parameters

In the previous tutorial, we described how the dataset parameters are controlled using “data.xml”. In this tutorial, we will learn to set up the processing parameters using a similar XML file - “sbas.xml”. This processing file is specific for the SBAS set of time-series inversions (See figure below).



In the example dataset directory, you will find a script named “prepsbasxml.py” .

```
> cd /home/ubuntu/data/giant/napa/GIAnT
> less prepsbasxml.py
#!/usr/bin/env python

import tsinsar as ts
import argparse
import numpy as np

if __name__ == '__main__':
    g = ts.TSXML('params')
    g. prepare_sbas_xml(nvalid = 20, netramp=True, atmos='',
                        demerr = False, filt=0.25)

    g.writexml('sbas.xml')
```

The complete list of all configurable parameters in “sbas.xml” can be found in the GIAnT user manual. We describe the parameters that we have set up using prepsbasxml.py below:

nvalid	Used for NSBAS inversion. Determines the minimum number of interferograms that a pixel should be coherent to be considered for inversion.
netramp	Boolean parameter controlling the deramping of interferograms. In this case, the applied ramp corrections are consistent over the entire network.
atmos	ProcessStack.py can download weather model data and use that for stratified trophospheric phase delay correction. This is beyond the scope of this tutorial. We use a empty string to indicate that no weather model corrections are to be applied.
demerr	Boolean parameter indicating if a DEM error term needs to be estimated. The baseline information from ifg.list is used for DEM error estimation.
filt	Width of the Gaussian filter to applied to the raw time-series to obtain the smoothed estimates. The value of this parameter is in years.

See GIAnT user manual for complete list of options and default values.

```
> cd /home/ubuntu/data/giant/napa/GIAnT
> python prepsbasxml.py
```

To view the generated “sbas.xml” file,

```
> less sbas.xml
<params>
  <proc>
    <nvalid>
      <value>20</value>
      <type>INT</type>
      <help>Minimum number of coherent IFGs for a single pixel. If
zero, pixel should be coherent in all IFGs.</help>
    </nvalid>
    <uwcheck>
      <value>False</value>
```

```
      <type>BOOL</type>
    </uwcheck>
    <netramp>
      <value>True</value>
      <type>BOOL</type>
      <help>Network deramp. Remove ramps from IFGs in a network
sense.</help>
    </netramp>
    <gpsramp>
      <value>False</value>
      <type>BOOL</type>
      <help>GPS deramping. Use GPS network information to correct
ramps.</help>
    </gpsramp>
    <stnlist>
      <value></value>
      <type>STR</type>
      <help>Station list for position of GPS stations.</help>
    </stnlist>
…....
</params>
```

Remember that the generated XML file can be modified in a text editor, and we again include a help string to describe each of the parameters in the file.

We are now ready to process our stack from the HDF5 file.

3. ProcessStack.py - Applying corrections

The first stage of processing, in which the data supplied by the users is modified, is accomplished using “ProcessStack.py”. The aim of this step is to

1. Correct for stratified troposphere artifacts, either
    a. Empirically by looking at relationship between InSAR phase and DEM
    b. Using weather models through the PyAPS package
2. Estimate ramps introduced due to orbital errors, either
    a. Either empirically by fitting a predefined orbit error function to data
    b. Using dense GPS observations

All the corrections are applied consistently across the interferogram network.

For this tutorial, we only choose to empirically deramping of interferograms. Details regarding other options and the associated fields in “sbas.xml” can be found in the GIAnT user manual.

Run “ProcessStack.py”

(NOTE: The ProcessStack.py command needs to be run from the X11 windows in the Remote Desktop function of EarthKit.)

```
> pwd
/home/ubuntu/data/giant/napa/GIAnT

> ProcessStack.py
logger - INFO - GIANT Toolbox - v 1.0
logger - INFO - --------------------
<module> - INFO - Input h5file: Stack/RAW-STACK.h5
<module> - INFO - Deleting previous Stack/PROC-STACK.h5
<module> - INFO - Output h5file: Stack/PROC-STACK.h5
deramp - INFO - PROGRESS: Estimating individual ramps.
[====================== 98% ==================> ]     17s /      0s
deramp - INFO - PROGRESS: Network deramp of IFGs.
[====================== 98% ==================> ]     27s /      0s
<module> - INFO - PNG preview of Deramped images: Figs/Ramp
[======>                11%                     ]     42s /    342s
```

Outputs of “ProcessStack.py” include - a processed stack file “Stack/PROC-STACK.h5” and a directory of PNG previews of deramped interferograms “Figs/Ramp”.

```
> ls Stack
PROC-STACK.h5 RAW-STACK.h5
```

To preview the contents of the new stack file

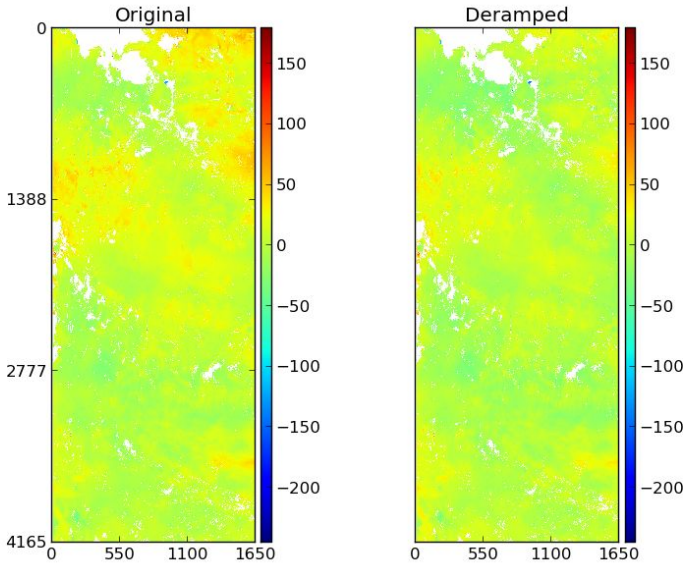
```
>h5ls Stack/PROC-STACK.h5
Jmat                     Dataset {30, 12}
bperp                    Dataset {30}
cmask                    Dataset {4165, 1650}
dates                    Dataset {12}
figram                   Dataset {30, 4165, 1650}
ramp                     Dataset {30, 3}
tims                     Dataset {12}
```

To view the contents of the directory with the PNG previews.

```
> ls Figs/Ramp
I001.png  I006.png  I011.png  I016.png  I021.png  I026.png
I002.png  I007.png  I012.png  I017.png  I022.png  I027.png
.......
```

To see the effect of deramping on the 7th interferogram in the Stack: (NOTE: you need to run this from Remote Desktop for the graphical viewer)

```
> eog Figs/Ramp/I007.png
```



Our stack is now deramped and ready for the final time-series inversion.

4. SBASInvert.py - Final inversion

In this tutorial, we demonstrate the simplest time-series inversion algorithm implemented in GIAnT - the SBAS algorithm. GIAnT also implements two other algorithms in the SBAS chain - NSBASInvert.py and TimefnInvert.py. The detailed discussion on the differences between these approaches can be found in the GIAnT user manual.

The SBAS algorithm estimates the differential displacement between one SAR acquisition and the next using a simple least squares approach. Our implementation of the algorithm estimates the time-series only for the pixels that are considered coherent in all the interferograms in the entire stack.

```
> SBASInvert.py
logger - INFO - GIANT Toolbox - v 1.0
logger - INFO - ---------------------
<module> - INFO - Number of interferograms = 30
<module> - INFO - Number of unique SAR scenes = 12
<module> - INFO - Number of connected components in network: 1
Timefn - INFO - Adding 12 linear pieces (SBAS)
<module> - INFO - Output h5file: Stack/LS-PARAMS.h5
[====================== 99% ==================> ]     800s /      0s
```

“SBASInvert.py” stores the inversion results in “Stack/LS-PARAMS.h5”.

```
> h5ls Stack/LS-PARAMS.h5
bperp                    Dataset {46}
cmask                    Dataset {1920, 2118}
dates                    Dataset {17}
gamma                    Dataset {SCALAR}
ifgcnt                   Dataset {1920, 2118}
mName                    Dataset {3}
masterind                Dataset {SCALAR}
parms                    Dataset {1920, 2118, 3}
rawts                    Dataset {17, 1920, 2118}
recons                   Dataset {17, 1920, 2118}
regF                     Dataset {3}
tims                     Dataset {17}
```

Note that the HDF5 file contains the raw time-series estimates (rawts) as well as the filtered time-series estimates (recons). In the next couple of sections, we will describe the visualization tools that are included with GIAnT.

5. plotts.py - Interactive visualization

GIAnT includes a script called “plotts.py” for interactive visualization of the generated time-series products. “plotts.py” requires a graphical desktop to run. It may require you to adjust matplotlib settings to work with an X-windows environment.

Note: (Execute Only if you have trouble using the visualization scripts)
To set matplotlib to run successfully in X-windows, edit or create this file:

```
> nano ~/.matplotlib/matplotlibrc
```

and set this value:

```
backend : TkAgg
```

(or)
Execute this on the command line

```
> echo “backend : TkAgg” >> ~/.matplotlib/matplotlibrc
```

Now we are ready to run “plotts.py”. Running the script the “-h” option list all the input parameters that can be controlled from command line.

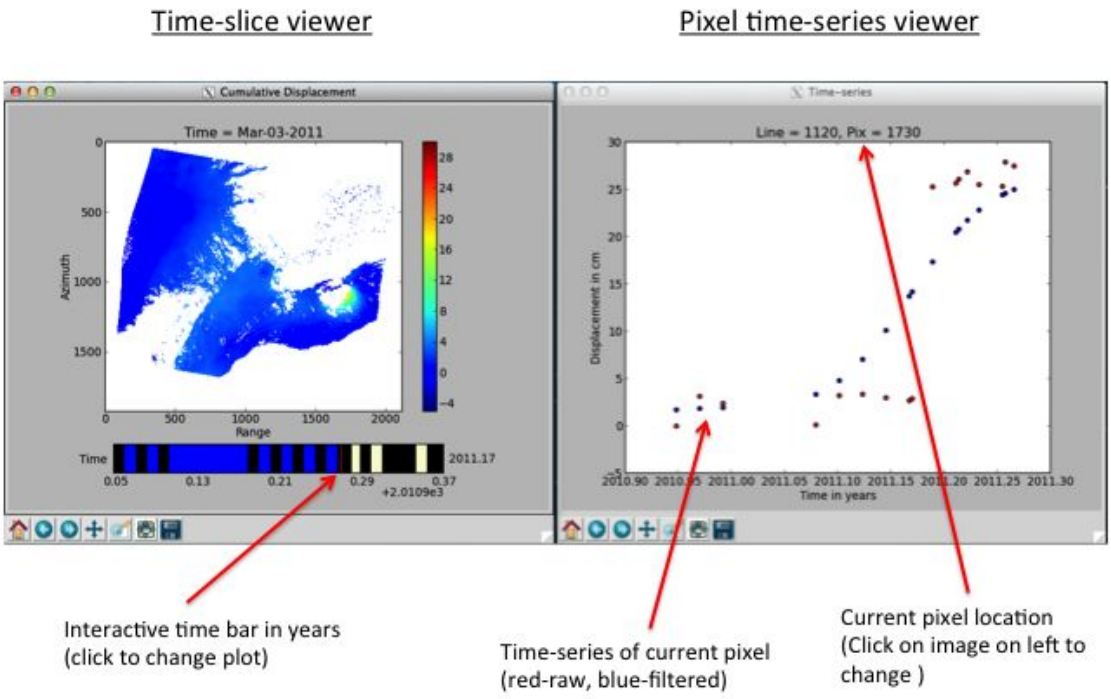
```
> plotts.py -h
logger - INFO - GIANT Toolbox - v 1.0
logger - INFO - ---------------------
usage: plotts.py [-h] [-e] [-f FNAME] [-i TIND] [-m MULT] [-y YLIM
YLIM]
                 [-ms MSIZE] [-raw] [-model] [-mask MASK MASK] [-zf]

Interactive SBAS time-series viewer

optional arguments:
  -h, --help       show this help message and exit
  -e               Display error bars if available. Default: False
  -f FNAME         Filename to use. Default: Stack/LS-PARAMS.h5
  -i TIND          Slice to display. Default: Middle index
  -m MULT          Scaling factor. Default: 0.1 for mm to cm
  -y YLIM YLIM     Y Limits for plotting. Default: [-25,25]
  -ms MSIZE        Marker size. Reduce if error bars are too small.
Default: 5
  -raw             Plot Un-Filtered Time Series as well, if available
  -model           Plot the individual model components as well. For
NSBAS, Timefn and MInTS.
```

```
  -mask MASK MASK  To mask out values. Need to provide 2 inputs -
Mask file in
                   float and xml file with dimensions. Default: None
  -zf              Changes time-origin to first acquisition for
showing time-
                   series.
```

To visualize the output from “SBASInvert.py”,

```
> plotts.py -f Stack/LS-PARAMS.h5 -y -5 30 -raw
```

This will open two plot windows - an interactive time-slice viewer and a pixel time-series viewer as shown below. The colorbar for the slice viewer ranges from -5 to 30 cm, and the raw time-series (red dots) is also shown along with the filtered time-series (blue); as requested using the command line flags -y and -raw.



Users can now view different time-slices by clicking on the time-bar and can view the time-series for different pixels by directly clicking on the pixel of interest in the image.

Some observations:

1. Our analysis clearly captures the offset associated with the Aug 24, 2014 Napa EQ.
2. GIAnT implements a simple Gaussian weighted moving average filter. Hence, the discrepancy between the raw displacement observations (red pixels) and the filtered observations (blue pixels).

3. Users can implement their own custom filtering with the raw time-series included in the HDF5 file.

Besides plotts.py, GIAnT can also export results as a movie through the “make_movie.py” script and as a Google Earth ready KML using “make_kml.py” scripts. For details and usage, refer to the GIAnT user manual. GIAnT also includes tools to export these datasets into GMT’s netcdf format and a GDAL compatible VRT. Users are strongly encouraged to use GDAL python bindings to export arrays from GIAnT’s HDF5 files to GIS-ready formats.

5. TimefnInvert.py - GPS-like time series modeling

In this section, we demonstrate the usage of “TimefnInvert.py” to perform a GPS-like functional form driven analysis of InSAR data . In this case, we define a functional form for the expected spatio-temporal evolution of the surface deformation over our area of interest based on *apriori knowledge* or observed deformation from a spatially sparse GPS network over the area of interest. For our example, we know than an earthquake occurred on Aug 24, 2014 in Napa and that our stack spans the event. We communicate a simple functional form for deformation - constant velocity + step function using a functionan named “timedict” in the “userfn.py” script as follows:

```
> less userfn.py
…
def timedict():
    rep = [['POLY',[1],[0.0]],
        ['STEP', [5.5113]]]
    return rep
```

Note that the time tags in the functional form is defined w.r.t to the first SAR acquisition in the stack (20090218) in fractional years.

```
>TimefnInvert.py
Timefn - INFO - Adding order 0 at T = 0.000000
Timefn - INFO - Adding order 1 at T = 0.000000
Timefn - INFO - Adding Step at T = 5.511300
<module> - INFO - Output h5file: Stack/TS-PARAMS.h5
[====================== 99% ==================> ]     800s /      0s
```

To visualize the results:

```
> plotts.py -f Stack/TS-PARAMS.h5 -y -15 15 -model
```

The model parameters are stored in a 3D matrix called “parms” in the output file “TS-PARAMS.h5”, and can be directly used for modeling. This should open three display windows instead of the usual two. The map of the model parameters is displayed in the third window (not linked to Pixel time-series viewer).

Note:
GIAnT user manual describes the convention for setting up complicated functional forms in detail.