Unofficial guide to using ISCE modules

ISCE developer team

Jet Propulsion Laboratory

September 18, 2014

Contents

Abstract

This document introduces various aspects of the ISCE software for users who are familiar with basic Python constructs. This document tries to explain the structure and usage of ISCE modules (same as Python classes), as stand-alone functional units. The document is meant to be a guide to help users build their own workflows using ISCE modules. Basic functional knowledge of the following concepts in Python are assumed - lists, dictionaries, iterators, classes, inheritance, property, constructors and pickle.

ISCE is a mix of Python / C / C++ / Fortran routines. All intensive computing is carried out in the C/C++/Fortran and Python provides an easy user interface to string together these routines. In developing workflows, users should never have to deal with the C/C++/Fortran directly. The basic assumption in ISCE is that if the information is wired correctly at the Python level then the C/C++/Fortran routines should work seamlessly. The Python level ISCE objects/ classes that are accessible to the users follow standard Python conventions and should be easy to understand for regular python users. This document is most useful for users who are not afraid of digging into Python source code.

The document introduces basic ISCE concepts first and then moves on to describe some example workflows. Remember that ISCE has an Application Framework (with extensive features) that should be used to design production ready workflows. This document is a guide to developing prototype workflows for testing or for personal use. These workflows should be migrated to the Application Framework, once they mature and are intended to be used in a production environment.

Learn Python

The authors of this document assume that the users are operating with basic functional knowledge of Python. ISCE processing modules are stand alone components and are typically included in the distribution as separate Python files (one per module). In our examples, we will not bother ourselves with the implementation of the various processing algorithms. This document focuses on interacting with the various Python classes in ISCE and assumes that users are able and willing to read the Python source code for the various modules. ISCE modules are reasonably well documented with doc strings and the source code is the best place to dig for answers.

1 Binary files in ISCE

ISCE uses an extensive C++-based image API (`components/iscesys/ImageApi`) to deal with passing image information between Python and C/Fortran. A simple interface to the C++ API is provided in `components/isceobj/Image/Image.py`. The basic attributes of an image object that need to be set by users are

1. Filename
2. Width
3. Datatype
4. Interleaving Scheme
5. Number of Bands
6. Access mode

1.1 Creating a simple image object

We describe the creation of an image object with an example. Suppose we have a 2-band, line interleaved image of size 1000 lines by 2000 pixels named "input.rmg" of float32 datatype. The following code snippet will allow use to create a corresponding ISCE image object, that reads in the data as float32 image:

```
#Standard import routines at top of file
import isce
import isceobj

obj=isceobj.createImage()
obj.setFilename('input.rmg')
obj.setWidth(1000)        #Width is required
#See components/iscesys/ImageApi/DataAccessor/bindings
```

```
obj.setDataType('FLOAT')
obj.setInterleavedScheme('BIL')      #BIL / BIP / BSQ
obj.setBands(2)                      # 2 Bands
obj.setAccessMode('read')            #read / Write
obj.createImage()                    #Now object is completely wired

###Use obj here for processing

obj.finalizeImage()                  #Close image after use
obj.renderHdr()                      #Create ISCE XML file if needed
```

1.2 Using casters: Advanced image object

Often, some of the code modules assume a certain data type for input data whereas the data is available in a different format. A typical example would be to provide support for use of short format DEMs and float32 format DEMs within the same module. To deal with such cases, the users can use a caster to make a short format image look like a float32 format image.

```
#Standard import routines at top of file
import isce
import isceobj
from isceobj.Image.Image import Image #Alternate import

#Does same thing as isceobj.createImage()
obj=Image()
obj.setFilename('short.dem')
obj.setWidth(1000)                   #Width is required


#See components/iscesys/ImageApi/DataAccessor/bindings
obj.setDataType('SHORT')
obj.setScheme('BIL')                 #BIL / BIP / BSQ
obj.setBands(1)                      # 2 Bands
obj.setAccessMode('read')            #read / Write
obj.setCaster('read','FLOAT')        #Mode needed in caster definition
obj.createImage()                    #Now object behaves like float image

###Use obj here for processing

obj.finalizeImage()                  #Close image after use
obj.renderHdr()                      #Create ISCE XML file if needed
```

1.3 Loading a file with an ISCE XML

If you already have an image with a corresponding ISCE XML file describing it, you can use

```
obj = Image()
obj.load('input.xml')
obj.setAccessMode('read')
#obj.setCaster('read', 'FLOAT') #if needed
obj.createImage()
```

1.4 Short hand

Alternately, if you know all the information about a particular image and want to initialize it in a single line

```
obj.initImage(filename,mode,width,dataType)
#obj.setCaster(mode,'FLOAT')  #if needed
obj.createImage()
```

There are a number of helper image functions that are available. These functions use a predetermined scheme and number of bands. They only need the filename, widths and casters to create Image objects.

1. `isceobj.createSlcImage()` *#CFLOAT, 1 band*
2. `isceobj.createDemImage()` *#SHORT, 1 band*
3. `isceobj.createIntImage()` *#CFLOAT, 1 band*
4. `isceobj.createAmpImage()` *#FLOAT, 2 band, BIP*
5. `isceobj.createRawImage()` *#BYTE, 1 band*

2 Understanding ISCE components

This section briefly describes the configurable framework that is used for ISCE components. Every processing module is derived from the base class `Component` that can be found in `components/iscesys/Component/Component.py`.

The following parts of an ISCE component are important for designing workflows:

1. Constructor
2. Parameters
3. Configure method
4. Ports
5. Main processing method

2.1 Constructor

The `__init__.py` method of the Python class is called its constructor. All constructors of ISCE components take in the keyword "name" as an optional input. When creating an instance of a particular ISCE class, the value of this "name" field ties it to an external XML file that can optionally be used to control its values. Here is an example:

```
import isce
from mroipac.Ampcor.Ampcor import Ampcor

#Object with public name "myexample"
obj = Ampcor(name='myexample')
```

In this case, "myexample.xml" will be scanned, whenever available, for setting up default parameters when this instance of Ampcor class is first created. Note that loading of information from XML files is done only during construction of the instance/ object. If the user does not wish the instance to be controlled by users, do not provide the keyword "name" as an input to the constructor. In general, the constructor is supposed to initialize the private variables of the ISCE `Component` that are not meant to be accessed directly by the users.

2.2 Parameters

Parameters represent the input information needed by an ISCE Component / Class. One will often see a long list of parameters defined for each Component. Here is an example:

```
#Example from mroipac/ampcor/Ampcor.py
WINDOW_SIZE_WIDTH = Component.Parameter('windowSizeWidth',
        public_name='WINDOW_SIZE_WIDTH',
        default=64,
        type=int,
        mandatory = False,
        doc = 'Window width of the reference data window for correlation.')
```

This code snippet defines a parameter that will be part of an ISCE Component and will be available as the member `self.windowSizeWidth` of the class after construction. The default value for this member is an integer of value 64. The "mandatory" keyword is set to False, to indicate that the user need not always specify this value. A default value is typically provided / determined by the workflow itself.

The `public_name` of a parameter specifies the tag of the property key to control its value using an XML file. If we create an instance `obj` with the public name "myexample", then an XML file with the name "myexample.xml" and following structure will modify the default value of `obj.windowSizeWidth` of the instance during construction.

```
##File 1
obj = Ampcor(name='myexample')

###myexample.xml
<myexample>
    <component name="myexample">
        <property name="WINDOW_SIZE_WIDTH">
            <value>32</value>
        </property>
    </component>
</myexample>
```

An ISCE component typically consists of more than one configurable parameter. Each ISCE component is also provided with an unique `family` name and `logging_name`. These are for use by advanced users and are beyond the scope of this document. The list of all configurable parameters of a Component are always listed in the `self.parameter_list` member of the Class. Shown below is the list of parameters for Ampcor

```
class Ampcor(Component):

    family = 'ampcor'
    logging_name = 'isce.mroipac.ampcor'

    parameter_list = (WINDOW_SIZE_WIDTH,
                      WINDOW_SIZE_HEIGHT,
                      SEARCH_WINDOW_SIZE_WIDTH,
                      SEARCH_WINDOW_SIZE_HEIGHT,
                      ZOOM_WINDOW_SIZE,
                      OVERSAMPLING_FACTOR,
                      ACROSS_GROSS_OFFSET,
                      DOWN_GROSS_OFFSET,
                      ACROSS_LOOKS,
                      DOWN_LOOKS,
                      NUMBER_WINDOWS_ACROSS,
                      NUMBER_WINDOWS_DOWN,
                      SKIP_SAMPLE_ACROSS,
                      SKIP_SAMPLE_DOWN,
                      DOWN_SPACING_PRF1,
                      DOWN_SPACING_PRF2,
                      ACROSS_SPACING1,
                      ACROSS_SPACING2,
                      FIRST_SAMPLE_ACROSS,
                      LAST_SAMPLE_ACROSS,
                      FIRST_SAMPLE_DOWN,
```

```
                    LAST_SAMPLE_DOWN,
                    IMAGE_DATATYPE1,
                    IMAGE_DATATYPE2,
                    SNR_THRESHOLD,
                    COV_THRESHOLD,
                    BAND1,
                    BAND2,
                    MARGIN,
                    DEBUG_FLAG,
                    DISPLAY_FLAG)
```

The key thing to remember when dealing with configurability is that the user defined XML file is only used for instance initializtion. If the any of the parameters are redefined in the workflow, the user-defined values are lost.

2.3 Configure method

Every ISCE Component has a *configure* method. This method is responsible for setting up the default parameter values for the class after initialization. If a Component is not configured after it is initialized, the class members corresponding to the defined parameter list may not be initialized. One will often see configure method immediately following the creation of a class instance like this

```
#Class instance with specific public name
ampcorObj = Ampcor(name='my_ampcor')

#Setting up of default values correctly
ampcorObj.configure()
```

2.4 Ports

Ports provide a mechanism to configure an ISCE component with relevant information in a quick and efficient way. For example, if a particular component needs the PRF, range sampling rate and velocity; it might be easier or relevant to pass the metadata of a SAR raw / SLC object with all this information through a port in one call. Instead of making three separate function calls to add the three parameters individually, adding a port can help us localize the setting up of these parameters to a single function and makes the code more readable.

Note that the ports are not the only way of setting up ISCE components. They only provide a simple mechanism to efficiently do so in most cases. We will illustrate this with an example.

Ports are always defined using the `createPorts` method of the ISCE class. We will use `components/stdproc/rectify/geocode/Geocode.py` to describe port definitions. There are multiple equivalent ways of defining ports.

```
def createPorts:
    #Method 1
    slcPort = Port(name='masterSlc', method=self.addMasterSlc)
    self._inputPorts.add(slcPort)

    #Method 2
    #self.inputPorts['masterSlc'] = self.addMasterSlc
```

Both code snippets above define an input port named `masterSlc`, that automatically invokes a class method `self.addMasterSlc`. In this case, the function `addMasterSlc` is defined as follows:

```
def addMasterSlc(self):
    formslc = self._inputPorts.getPort(name='masterslc').getObject()
    if(formslc):
        try:
            self.rangeFirstSample = formslc.startingRange
        except AttributeError as strerr:
            self.logger.error(strerr)
            raise AttributeError
```

Note that if the input port `masterSlc` is not wired up, no changes are made to the instance / object. Alternately, users can directly set the rangeFirstSample without using the ports mechanism with a call as follows:

```
obj.rangeFirstSample = 850000.
```

Ports is a mechanism to use localize the wiring up of relevant information in different sections of the code, and helps in the logical organization of the ISCE components.

In ISCE every port is accompanied by an `addPortName` method. The method takes one python object as input. The exact manner in which these ports are used will become clear in the next subsection. There are two equivalent ways of wiring up input ports:

```
#Method 1. Explicit wiring.
#Example in isceobj/InsarProc/runTopo.py
objTopo.wireInputPort(name='peg', object=self.insar.peg)

#Method 2. When making the main processing call.
#Example in isceobj/InsarProc/runGeocode.py
objGeocode(peg=self.insar.peg,
           frame=self.insar.masterFrame, ...)
```

2.5 Main Processing Method

In ISCE, component/ class names use camel case, e.g, `DenseOffsets, Ampcor` etc. The main processing method of each of these components is the lowercase string corresponding to the name of the Class. For example, the main method for class `Ampcor` is called —ampcor— and that of class `DenseOffsets` is called `denseoffsets`. The class instances themselves are also callable. Making a function call with the class instance results in automatic wiring of input ports and execution of the main processing method. Typical use of an ISCE component follows this pattern:

```
###Example from runPreprocessor.py in isceobj/components/InsarProc
###1. Initialize a component
baseObj = Baseline()
baseObj.configure()

###2. Wire input ports
baseObj.wireInputPort(name='masterFrame',
                      object=self._insar.getMasterFrame())
baseObj.witeInputPort(name='slaveFrame',
                      object=self._insar.getSlaveFrame())

###3. Set any other values that you would like
###This particular module does not need other parameters
###Say using ampcor, you might set values if you have
###computed offsets using orbit information
###obj.setGrossAcrossOffset(coarseAcross)
###obj.setGrossDownOffset(coarseDown)

####4. Call the main process
baseObj.baseline()

###The instance itself is callable. Could have done.
#baseObj(masterFrame=self._insar.getMasterFrame(),
#        slaveFrame=self._insar.getSlaveFrame())

####5. Get any values out of the processing as needed
horz_baseline_top = baseObj.hBaselineTop
vert_baseline_top = baseObj.vBaselineTop
```

3 Example 0: Automated generation of input XML files

XML is a great format for automating large workflows and organizing data in a manner that is machine friendly. However, construction of XML files by hand can be a tedious task. ISCE

includes utilities that help users convert standard Python dictionaries into meaningful XML files.

```
import isce
from isceobj.XmlUtil import FastXML as xml

if __name__ == '__main__':
    '''
    Example demonstrating automated generation of insarApp.xml for
    COSMO SkyMed raw data.
    '''

    #####Initialize a component named insar
    insar = xml.Component('insar')

    ####Python dictionaries become components
    ####Master info
    master = {}
    master['hdf5'] = 'master.h5'
    master['output'] = 'master.raw'

    ####Slave info
    slave = {}
    slave['hdf5'] = 'slave.h5'
    slave['output'] = 'slave.raw'

    #####Set sub-component
    insar['master'] = master
    insar['slave'] = slave

    ####Set properties
    insar['doppler method'] = 'useDEFAULT'
    insar['sensor name'] = 'COSMO_SKYMED'
    insar['range looks'] = 4
    insar['azimuth looks'] = 4

    #####Catalog example
    insar['dem'] = xml.Catalog('dem.xml')

    ####Components include a writeXML method
    insar.writeXML('insarApp.xml', root='insarApp')
```

This produces an output file named "insarApp.xml" that looks like this

```
<insarApp>
    <component name="insar">
        <component name="master">
            <property name="hdf5">
                <value>master.h5</value>
            </property>
            <property name="output">
                <value>master.raw</value>
            </property>
        </component>
        <component name="slave">
            <property name="hdf5">
                <value>slave.h5</value>
            </property>
            <property name="output">
                <value>slave.raw</value>
            </property>
        </component>
        <property name="doppler method">
            <value>useDEFAULT</value>
        </property>
        <property name="sensor name">
            <value>COSMO_SKYMED</value>
        </property>
        <property name="range looks">
            <value>4</value>
        </property>
        <property name="azimuth looks">
            <value>4</value>
        </property>
        <component name="dem">
            <catalog>dem.xml</catalog>
        </component>
    </component>
</insarApp>
```

FastXML can be easily used to quickly generated input files for ISCE applications like make_raw.py, insarApp.py and isceApp.py.

4 Advanced example 1: Using ampcor for estimating offset field

Here is a complete example that lets one use `Ampcor` on any two arbitrary files:

```
#!/usr/bin/env python

import isce
import logging
import isceobj
import mroipac
import argparse
from mroipac.ampcor.Ampcor import Ampcor
import numpy as np

def cmdLineParser():
    parser = argparse.ArgumentParser(description='Simple ampcor driver')
    parser.add_argument('-m', dest='master', type=str,
            help='Master image with ISCE XML file', required=True)
    parser.add_argument('-b1', dest='band1', type=int,
            help='Band number of master image', default=0)
    parser.add_argument('-s', dest='slave', type=str,
            help='Slave image with ISCE XML file', required=True)
    parser.add_argument('-b2', dest='band2', type=int,
            help='Band number of slave image', default=0)
    parser.add_argument('-o', dest='outfile', default= 'offsets.txt',
            type=str, help='Output ASCII file')
    return parser.parse_args()


#Start of the main program
if __name__ == '__main__':

    logging.info("Calculate offset between two using ampcor")

    #Parse command line
    inps = cmdLineParser()

    ####Create master image object
    masterImg = isceobj.createImage()    #Empty image
    masterImg.load(inps.master +'.xml') #Load from XML file
    masterImg.setAccessMode('read')     #Set it up for reading
    masterImg.createImage()             #Create File

    #####Create slave image object
    slaveImg = isceobj.createImage()     #Empty image
    slaveImg.load(inps.slave +'.xml')    #Load it from XML file
    slaveImg.setAccessMode('read')       #Set it up for reading
    slaveImg.createImage()               #Create File
```

```
####Stage 1: Initialize
objAmpcor = Ampcor(name='my_ampcor')
objAmpcor.configure()

####Defautl values used if not provided in my_ampcor
coarseAcross = 0
coarseDown = 0

####Get file types
if masterImg.getDataType().upper().startswith('C'):
    objAmpcor.setImageDataType1('complex')
else:
    objAmpcor.setImageDataType1('real')

if slaveImg.getDataType().upper().startswith('C'):
    objAmpcor.setImageDataType2('complex')
else:
    objAmpcor.setImageDataType2('real')

#####Stage 2: No ports for ampcor
### Any parameters can be controlled through my_ampcor.xml

### Stage 3: Set values as needed
####Only set these values if user does not define it in my_ampcor.xml
if objAmpcor.acrossGrossOffset is None:
    objAmpcor.acrossGrossOffset = coarseAcross

if objAmpcor.downGrossOffset is None:
    objAmpcor.downGrossOffset = coarseDown

logging.info('Across Gross Offset = %d'%(objAmpcor.acrossGrossOffset))
logging.info('Down Gross Offset = %d'%(objAmpcor.downGrossOffset))

####Stage 4: Call the main method
objAmpcor.ampcor(masterImg,slaveImg)

###Close ununsed images
masterImg.finalizeImage()
slaveImg.finalizeImage()
```

```
######Stage 5: Get required data out of the processing run
offField = objAmpcor.getOffsetField()
logging.info('Number of returned offsets : %d'%(len(offField._offsets)))

####Write output to an ascii file
field = np.array(offField.unpackOffsets())
np.savetxt(inps.outfile, field, delimiter="    ", format='%5.6f')
```

One can control the `ampcor` parameters using a simple example file in the same directory as shown below:

```
<!--my_ampcor.xml-->
<my_ampcor>
    <component name="my_ampcor">
        <property name="ACROSS_GROSS_OFFSET">60</property>
        <property name="DOWN_GROSS_OFFSET">-164</property>
    </component>
</my_ampcor>
```

5 Plugging into insarApp workflow

Currently, `insarApp.py` is the most used and maintained application. It is designed to produce one differential interferogram from a pair of SAR acquisitions. ISCE applications are designed to work with check-pointing capability - i.e, the state of the processing variables are periodically stored to a pickle file. This allows users to experiment with various intermediate products and metadata.

The entire state of the processing workflow for `insarApp` is stored in an object of type `InsarProc` defined in `components/isceobj/InsarProc/InsarProc.py`. Any intermediate product of interest can be accessed from a pickled copy of this object.

5.1 Loading a pickle

Loading a pickle object from an insarApp run is straightforward. It is done as follows:

```
###These import lines are important. If the class definitions from these are
###unavailable, pickle will not be able to unpack binary data in pickle files
###to the correct objects
import isce
import isceobj

def load_pickle(fname='PICKLE/formslc'):
    import cPickle #Optimized version of pickle

    insarObj = cPickle.load( open(fname, 'rb'))
    return insarObj

if __name__ == '__main__':
    '''
    Dummy testing.
    '''
    insar = load_pickle()
```

```
    vel, hgt = insar.vh()

    print 'Velocity: ', vel
    print 'Height: ', hgt

    planet = insar.masterFrame._instrument._platform._planet
    print 'Planet :', planet
```

Once the object is loaded, users can play around with the values stored in the `InsarProc` object.

5.2 Saving a pickle

In the extreme cases, where the users would like to modify the contents of the pickle files, they would only need to overwrite the corresponding pickle file and insarApp will use the new state for further processing.

```
import isce
import isceobj

def save_pickle(insar, fname='PICKLE/formslc'):
    import cPickle

    fid = open(fname, 'wb')
    cPickle.dump(insar, fid)
    fid.close()
```

Currently, this method is used by the `updateBbox.py` script in `isce/calimap` directory to updated bounding boxes for multiple interferograms to enable geocoding on to a common grid.

6 Advanced example 2: Projecting a deformation model in radar coordinates

In this example, we will demonstarte the ENU2LOS modules to ISCE to project ground deformation into radar coordinates and use the information to correct a wrapped interferogram.

```
#!/usr/bin/env python

###Our usual import statements
import numpy as np
import isce
import isceobj
from stdproc.model.enu2los.ENU2LOS import ENU2LOS
import argparse

####Method to load pickle information
####from an insarApp run
def load_pickle(step='topo'):
    '''Loads the pickle from correct as default.'''
    import cPickle

    insarObj = cPickle.load(open('PICKLE/{0}'.format(step),'rb'))
    return insarObj


###Create dummy model file if needed
###Use this for simple testing
###Modify values as per your test dataset

def createDummyModel():
    '''Creates a model image.'''
    wid = 401
    lgt = 401
    startLat = 20.0
    deltaLat = -0.025
    startLon = -156.0
    deltaLon = 0.025

    data = np.zeros((lgt,3*wid), dtype=np.float32)
    ###East only
#    data[:,0::3] = 1.0
    ###North only
#    data[:,1::3] = 1.0
    ###Up only
    data[:,2::3] = 1.0

    data.tofile('model.enu')

    print('Creating model object')
    objModel = isceobj.createDemImage()
```

```
    objModel.setFilename('model.enu')
    objModel.setWidth(wid)
    objModel.scheme = 'BIP'
    objModel.setAccessMode('read')
    objModel.imageType='bip'
    objModel.dataType='FLOAT'
    objModel.bands = 3
    dictProp = {'REFERENCE':'WGS84','Coordinate1': \
                    {'size':wid,'startingValue':startLon,'delta':deltaLon}, \
                    'Coordinate2':{'size':lgt,'startingValue':startLat, \
                    'delta':deltaLat},'FILE_NAME':'model.enu'}
    objModel.init(dictProp)
    objModel.renderHdr()
```

```
###cmd Line Parser
def cmdLineParser():
    parser = argparse.ArgumentParser(description="Project ENU deformation to LOS in rada
    parser.add_argument('-m','--model', dest='model', type=str,
        required=True,
        help='Input 3 channel FLOAT model file with DEM like info')
    parser.add_argument('-o','--output', dest='output', type=str,
        default='enu2los.rdr', help='Output 1 channel LOS file')

    return parser.parse_args()

###The main program
if __name__ == '__main__':

    ###Parse command line
    inps = cmdLineParser()

    ###For testing only
#     createDummyModel()

    ####Load model image
    print('Creating model image')
    modelImg = isceobj.createDemImage()
    modelImg.load(inps.model +'.xml')  ##From cmd line

    if (modelImg.bands !=3 ):
```

```
    raise Exception('Model input file should be a 3 band image.')

modelImg.setAccessMode('read')
modelImg.createImage()


####Get geocoded information
startLon = modelImg.coord1.coordStart
deltaLon = modelImg.coord1.coordDelta
startLat = modelImg.coord2.coordStart
deltaLat = modelImg.coord2.coordDelta

####Load geometry information from pickle file.
iObj = load_pickle()
topo = iObj.getTopo()   #Get info for the dem in radar coords

####Get the wavelength information.
###This is available in multiple locations within insarProc
#wvl = iObj.getMasterFrame().getInstrument().getRadarWavelength()
wvl = topo.radarWavelength


####Pixel-by-pixel Latitude image
print('Creating lat image')
objLat = isceobj.createImage()
objLat.load(topo.latFilename+'.xml')
objLat.setAccessMode('read')
objLat.createImage()

####Pixel-by-pixel Longitude image
print('Creating lon image')
objLon = isceobj.createImage()
objLon.load(topo.lonFilename+'.xml')
objLon.setAccessMode('read')
objLon.createImage()

#####Pixel-by-pixel LOS information
print('Creating LOS image')
objLos = isceobj.createImage()
objLos.load(topo.losFilename +'.xml')
objLos.setAccessMode('read')
objLos.createImage()

###Check if dimensions are the same
```

```
for img in (objLon, objLos):
    if (img.width != objLat.width) or (img.length != objLat.length):
        raise Exception('Lat, Lon and LOS files are not of the same size.')


####Create an output object
print ('Creating output image')
objOut = isceobj.createImage()
objOut.initImage(inps.output, 'write', objLat.width, type='FLOAT')
objOut.createImage()


print('Actual processing')
####The actual processing
#Stage 1: Construction
converter = ENU2LOS()
converter.configure()

#Stage 2: No ports for enu2los
#Stage 3: Set values
converter.setWidth(objLat.width)    ###Radar coords width
converter.setNumberLines(objLat.length) ###Radar coords length
converter.setGeoWidth(modelImg.width) ###Geo coords width
converter.setGeoNumberLines(modelImg.length) ###Geo coords length

###Set up geo information
converter.setStartLatitude(startLat)
converter.setStartLongitude(startLon)
converter.setDeltaLatitude(deltaLat)
converter.setDeltaLongitude(deltaLon)

####Set up output scaling
converter.setScaleFactor(1.0)    ###Change if ENU not in meters
converter.setWavelength(4*np.pi)     ###Wavelength for conversion to radians

converter.enu2los(modelImage = modelImg,
                  latImage = objLat,
                  lonImage = objLon,
                  losImage = objLos,
                  outImage = objOut)

#Step 4: Close the images
modelImg.finalizeImage()
objLat.finalizeImage()
```

```
    objLon.finalizeImage()
    objLos.finalizeImage()
    objOut.finalizeImage()
    objOut.renderHdr()    ###Create output XML file
```

7 Advanced Example 3: WGS84 orbits vs SCH orbits

In this example, we demonstrate the use of `Orbit` class in ISCE to deal with different types of orbits. The example compares two different orbit interpolation schemes

1. WGS84 raw state vectors (to) WGS84 line-by-line vectors using Hermite polynomials (to) SCH line-by-line vectors
2. WGS84 raw state vectors (to) SCH raw state vectors (to) SCH line-by-line vectors using linear interpolation

```
#!/usr/bin/env python
import numpy as np
import isce
import isceobj
import stdproc
import copy
from iscesys.StdOEL.StdOELPy import create_writer
from isceobj.Orbit.Orbit import Orbit

###Load data from an insarApp run
###Load orbit2sch by default
def load_pickle(step='orbit2sch'):
    import cPickle

    insarObj = cPickle.load(open('PICKLE/{0}'.format(step), 'rb'))
    return insarObj

if __name__ == '__main__':
    ##### Load insarProc object
    print('Loading original and interpolated WGS84 state vectors')
    iObj = load_pickle(step='mocompath')

    ####Make a copy of the peg point data
    peg = copy.copy(iObj.peg)


    #####Copy the original state vectors
    #####These are the 10-15 vectors provided
    #####with the sensor data in WGS84 coords
    origOrbit = copy.copy(iObj.masterFrame.getOrbit())
    print('From Original Metadata - WGS84')
    print('Number of state vectors: %d'%len(origOrbit._stateVectors))
    print('Time interval: %s %s'%(str(origOrbit._minTime),
    str(origOrbit._maxTime)))


    #####Line-by-line WGS84 interpolated orbit
    #####This was done using Hermite polynomials
    xyzOrbit =  copy.copy(iObj.masterOrbit)
    print('Line-by-Line XYZ interpolated')
    print('Number of state vectors: %d'%len(xyzOrbit._stateVectors))
    print('Time interval: %s %s'%(str(xyzOrbit._minTime),
    str(xyzOrbit._maxTime)))
```

```
####Delete the insarProc object from "mocomppath"
del iObj

####Note:
####insarApp converts WGS84 orbits to SCH orbits
####during the orbit2sch step


######Line-by-line SCH orbit
######These were generated by converting
######Line-by-Line WGS84 orbits
print('Loading interpolated SCH orbits')
iObj = load_pickle('orbit2sch')

####Copy the peg information needed for conversion
pegHavg = copy.copy(iObj.averageHeight)
planet = copy.copy(iObj.planet)

###Copy the orbits
schOrbit = copy.copy(iObj.masterOrbit)
del iObj
print('Line-by-Line SCH interpolated')
print('Number of state vectors: %d'%len(schOrbit._stateVectors))
print('Time interval: %s %s'%(str(schOrbit._minTime),
    str(schOrbit._maxTime)))


######Now convert the original state vectors to SCH coordinates
###stdWriter logging mechanism for some fortran modules
stdWriter = create_writer("log","",True,filename='orb.log')

print('**********************')
orbSch = stdproc.createOrbit2sch(averageHeight=pegHavg)
orbSch.setStdWriter(stdWriter)
orbSch(planet=planet, orbit=origOrbit, peg=peg)
print('**********************')

schOrigOrbit = copy.copy(orbSch.orbit)
del orbSch
print('Original WGS84 vectors to SCH')
print('Number of state vectors: %d'%len(schOrigOrbit._stateVectors))
print('Time interval: %s %s'%(str(schOrigOrbit._minTime),
    str(schOrigOrbit._maxTime)))
print(str(schOrigOrbit._stateVectors[0]))
```

```
####Line-by-line interpolation of SCH orbits
####Using SCH orbits as inputs
pulseOrbit = Orbit()
pulseOrbit.configure()

#######Loop over and compare against interpolated SCH
for svOld in xyzOrbit._stateVectors:
    ####Get time from Line-by-Line WGS84
    ####And interpolate SCH orbit at those epochs
    ####SCH intepolation using simple linear interpolation
    ####WGS84 interpolation would use keyword method="hermite"
    svNew = schOrigOrbit.interpolate(svOld.getTime())
    pulseOrbit.addStateVector(svNew)


####Clear some variables
del xyzOrbit
del origOrbit
del schOrigOrbit

#####We compare the two interpolation schemes
####Orig WGS84 -> Line-by-line WGS84 -> Line-by-line SCH
####Orig WGS84 -> Orig SCH -> Line-by-line SCH

###Get the orbit information into Arrays
(told,xold,vold,relold) = schOrbit._unpackOrbit()
(tnew,xnew,vnew,relnew) = pulseOrbit._unpackOrbit()


xdiff = np.array(xold) - np.array(xnew)
vdiff = np.array(vold) - np.array(vnew)

print('Position Difference stats')
print('L1 mean in meters')
print(np.mean(np.abs(xdiff), axis=0))
print('')
print('RMS in meters')
print(np.sqrt(np.mean(xdiff*xdiff, axis=0)))

print('Velocity Difference stats')
print('L1 mean in meters/sec')
```

```
print(np.mean(np.abs(vdiff), axis=0))
print(' ')
print('RMS in meters/sec')
print(np.sqrt(np.mean(vdiff*vdiff, axis=0)))
```

8 Advanced Example 4: Working with GDAL, Landsat and ISCE

In this example, we demonstrate the interaction between numpy, gdal and ISCE libraries. We extract the Panchromatic band from a Landsat-8 archive obtained from Earth Explorer and create a corresponding ISCE XML file. This file can then readily be ingested with any ISCE module. An example would be dense pixel offset estimation on optical imagery.

```
#!/usr/bin/env python

import numpy as np
import argparse
from osgeo import gdal
import isce
import isceobj
import os

def cmdLineParse():
    '''
    Parse command line.
    '''
    parser = argparse.ArgumentParser(description='Convert GeoTiff to ISCE file')
    parser.add_argument('-i','--input', dest='infile', type=str,
            required=True, help='Input GeoTiff file. If tar file
            is also included, this will be output file extracted
            from the TAR archive.')
    parser.add_argument('-o','--output', dest='outfile', type=str,
            required=True, help='Output GeoTiff file')
    parser.add_argument('-t','--tar', dest='tarfile', type=str,
            default=None, help='Optional input tar archive. If provided,
            Band 8 is extracted to file name provided with input option.')

    return parser.parse_args()

def dumpTiff(infile, outfile):
    '''
    Read geotiff tags.
    '''
    ###Uses gdal bindings to read geotiff files
    data = {}
    ds = gdal.Open(infile)
    data['width'] = ds.RasterXSize
    data['length'] = ds.RasterYSize
    gt = ds.GetGeoTransform()

    data['minx'] = gt[0]
    data['miny'] = gt[3] + data['width'] * gt[4] + data['length']*gt[5]
    data['maxx'] = gt[0] + data['width'] * gt[1] + data['length']*gt[2]
    data['maxy'] = gt[3]
    data['deltax'] = gt[1]
    data['deltay'] = gt[5]
    data['reference'] = ds.GetProjectionRef()
```

```
    band = ds.GetRasterBand(1)
    inArr = band.ReadAsArray(0,0, data['width'], data['length'])
    inArr.astype(np.float32).tofile(outfile)

    return data

def extractBand8(intarfile, destfile):
    '''
    Extracts Band 8 of downloaded Tar file from EarthExplorer
    '''
    import tarfile
    import shutil

    fid = tarfile.open(intarfile)
    fileList = fid.getmembers()

    ###Find the band 8 file
    src = None
    for kk in fileList:
        if kk.name.endswith('B8.TIF'):
            src = kk

    if src is None:
        raise Exception('Band 8 TIF file not found in tar archive')

    print('Extracting: %s'%(src.name))

    ####Create source and target file Ids.
    srcid = fid.extractfile(src)
    destid = open(destfile,'wb')

    ##Copy content
    shutil.copyfileobj(srcid, destid)
    fid.close()
    destid.close()


if __name__ == '__main__':
    ####Parse cmd line

    inps = cmdLineParse()

    ####If input tar file is given
```

```
    if inps.tarfile is not None:
        extractBand8(inps.tarfile, inps.infile)

    print('Dumping image to file')
    meta = dumpTiff(inps.infile, inps.outfile)

#    print(meta)
    ####Create an ISCE XML header for the landsat image
    img = isceobj.createDemImage()
    img.setFilename(inps.outfile)
    img.setDataType('FLOAT')

    dictProp = {
                'REFERENCE' : meta['reference'],
                'Coordinate1': {
                                  'size': meta['width'],
                                  'startingValue' : meta['minx'],
                                  'delta': meta['deltax']
                               },
                'Coordinate2': {
                                   'size' : meta['length'],
                                   'startingValue' : meta['maxy'],
                                   'delta': meta['deltay']
                               },
                'FILE_NAME' : inps.outfile
              }
    img.init(dictProp)
    img.renderHdr()
```