AOIC

Release 1.0 beta

June 11, 2015

CHAPTER ONE

OVERVIEW

AOIC (ANOPP2 OpenMDAO Interface Code) is an interface code written in Python developed at NASA (National Aeronautics and Space Administration) LARC (Langley Research Center) that offers an ability to use ANOPP2 (Aircraft NOise Prediction Program 2) to optimize designs using OPENMDAO (Open Multi-disciplinary Design Analysis and Optimization). The AOIC contains two classes: the first interacts with ANOPP2 for setting the problem to be optimized and the second specifies the conditions for optimization. The AOIC developed to solve a simple case of finding the location of maximum noise along a sideline parallel to the flight path during an aircraft takeoff is provided and explained.

CHAPTER

TWO

INTRODUCTION

Aircraft design has traditionally been based on performance characteristics, carrying capacity, and operational efficiency. With FAA's increasingly stringent specifications on aircraft noise, aircraft designers are increasing the importance of aircraft noise in their designs. Design engineers can use OpenMDAO to arrive at a combination of various design variables that optimizes several performance characteristics. ANOPP2 is a tool available to aircraft design engineers for predicting noise from the aircraft at various observer locations. AOIC is an interface code, written in Python, between ANOPP2 and OpenMDAO for the purpose of minimizing aircraft noise. Aircraft design engineers can include noise as one of the parameters to be optimized by using AOIC to bring aircraft noise, predicted by ANOPP2, into OpenMDAO. A demonstration case is provided that predicts sideline noise during takeoff for the purpose of demonstrating the functionality of the AOIC. During takeoff, one of the certification microphone locations is on the sideline, measured in terms of EPNL (Effective Perceived Noise Level). The sideline microphone location should be at the maximum noise location; this sample case demonstrates the use of AOIC to determine this location.

CHAPTER THREE

REQUIREMENTS

AOIC requires the installation of OpenMDAO in addition to Python 2.7 along with the numerical and scientific packages NumPy and SciPy. AOIC is a Python plugin in OpenMDAO that the user will have to create to use ANOPP2 for noise optimization. AOIC also requires library, `libANOPP2.so`, and the ANOPP2 API, `ANOPP2_api.py`.

CHAPTER FOUR

USAGE GUIDE

The AOIC contains two classes: *Anopp2Component* and *Anopp2Optimize*. The *Anopp2Component* class contains the essence of all interactions with ANOPP2. Details of the *Anopp2Component* class used in this specific example are provided herein. Users desiring to use ANOPP2 for optimization through OpenMDAO are required to develop similar codes with appropriate ANOPP2 function calls and other necessary associated files (see ANOPP2 Manuals for more information). This example code is tested through a test code, *test_anopp2.py*.

In this case, the noise generated during the takeoff of a *Tube and Wing* aircraft is modeled through ANOPP2 and the noise along a sideline parallel to the takeoff direction on the ground is predicted. The location along the sideline at which the noise (measured through the acoustic metric EPNL) reaches its maximum is determined through OpenMDAO. The details of this AOIC are explained here.

4.1 Anopp2Component

4.1.1 Initialization

Step 1. Initialize ANOPP2 API: The ANOPP2 Command Executive API [*LOPES2014A*] is initialized first through the following command: `ANOPP2.a2py_exec_init_api ().`

Step 2. Initialize local variables: Initialize the Atmosphere and Flight Path tags.

Step 3. Create an Observer: An observer [*LOPES2014B*] is defined as a point cloud (a collection of several points in space, also referred to as *nodes*) along the sideline defined by its x (along the direction of the flight on the ground), y (perpendicular to the direction of flight on the ground), and z (perpendicular to the ground) axes. The observer is defined in a configuration file, *observer.config* placed in the current working folder. This configuration file is provided below:

```
!-------------------------------------------------------------------------------
! This is the namelist that defines the ground observer positions that will be passed
! to ANOPP during execution.  This configuration is for several observer positions
! along the sideline of a takeoff flightpath.
!-------------------------------------------------------------------------------
&ObserverPointCloudNamelist

  ! Since our microphone is stationary, this parameter is 0, meaning the microphone
  ! is not moving within the ground frame of reference.
  nFrameChanges = 0

/
  !-----------------------------------------------------------------------------
  ! The point will be anywhere along the sideline of the takeoff maneuver.  The line
  ! is 1476 feet (or 449.8848 meters) to the side of the flight path.  The OpenMDAO
  ! is expected to locate the the observer position at which maximum EPNdB is found.
```

```
!-------------------------------------------------------------------------------
&PointCloudNamelist

  ! This is the number of nodes.
  nNodes = 1

/
2410.0, 449.8848, 0.0
```

The observer is created through the following ANOPP2 command:

```
intSuccess =             \
  ANOPP2.a2py_obs_create \
   (pointer(self.intObserverTag), create_string_buffer (b"observer.config"))
```

4.1.2 Execution

The *Anopp2.py* performs the following steps as part of its execution. Each of these steps are executed for each iteration to determine the noise (EPNL) corresponding to that observer location.

Step 4. Create a New Observer Node: A new node is introduced into the observer at a location whose `x` coordinate is provided by the optimizer and the `y` and `z` coordinates are those of the previous node. So, the number of nodes in the observer is first obtained through the following function call:

```
intSuccess = ANOPP2.a2py_obs_number_of_nodes (self.intObserverTag, self.nNodes)
```

The position of the last node in the observer is obtained through the following function call:

```
intSuccess =               \
  ANOPP2.a2py_obs_position \
    (self.intObserverTag, self.nNodes, 0.0, a2_global, self.fltPosition)
```

where, `self.fltPosition` is an array containing the `x`, `y`, and `z` coordinates of the position of the last node in the observer.

The first value of the Position array (x coordinate) is replaced with the value provided by the optimizer: `self.fltPosition[0] = self.x`. A new node is added in the observer point cloud at the location corresponding to the values of the `fltPosition` array through the following function call:

```
intSuccess = ANOPP2.a2py_obs_new_node (self.intObserverTag, self.fltPosition)
```

Step 5. Create an AnoppComplete Functional Module: In this case, ANOPP2's AnoppComplete Functional Module [LOPES2014C] is used to predict the EPNL. This functional module is invoked in ANOPP2 through the following routine call:

```
intSuccess =                                                                \
  ANOPP2.a2py_exec_create_functional_module                                 \
   (pointer(self.intAnoppCompleteTag),                                      \
    create_string_buffer (b"AnoppComplete.config"), self.nInputs, self.intInputTags, \
    pointer(self.intObserverTag), pointer(self.nResults), pointer(self.intResultTags))
```

The settings and details required for using this Functional Module are provided in the Configuration file, *AnoppComplete.config*. The contents of this file is provided here.

```
!-------------------------------------------------------------------------------
! This is the input file for the ANOPP Complete Prediction using ANOPP modules to
! predict noise. The output of this prediction method is what is with the ANOPP
! Prediction module.
```

```
!-------------------------------------------------------------------------------------
&AnoppCompleteNamelist

  ! Specify the ANOPP executable to be used including the path.  The ANOPP executable
  ! should be located in the current directory.
  strExecutableFileName = "anopp.exe"

  ! This is the input deck that contains the sound sources and the propagation to the
  ! observer positions.  This includes all noise sources and their propagation (a
  ! complete prediction).
  strInputDeck = "Takeoff-Complete.inp"

  ! ANOPP Output File name.  Once ANOPP is executed, the output of ANOPP is copied to
  ! this file name.  A user may peruse this filename if they wish to see how ANOPP
  ! was executed and any additional output it may provide.
  strOutputFileName = "TubeAndWing-Simple.out"

  ! Specify whether you want the ANOPP Input Deck to be created by ANOPP2 (.TRUE.) or
  ! if the input deck is already available and you want to use it (.FALSE.). This is
  ! helpful if you just want ANOPP to execute an already existing deck (useful for
  ! debugging).
  blnOverwriteInputDeck = .TRUE.

  ! This is a list of strings that are the names of th files to be parse after ANOPP
  ! is executed.  These names must correspond to the names of the LEVOUT parameter
  ! in the ANOPP input deck.  These are the names of the results to be taken out of
  ! the ANOPP run. The order matters.
  strLevFilenames = "TOTAL.OUT", "ENGINE.OUT", "AIRFRAME.OUT"

  ! This is the flag that turns on metadata for this module.  Metadata for the ANOPP
  ! Complete prediction module includes the geometric emission angles as a function
  ! of time.  This is written out by ANOPP if the IPRINT parameter is set to 3 right
  ! before the EXECUTE GEO command in the input deck.  If that is set and this flag
  ! is set to true, a Metadata folder will be created, and a file containing the
  ! emission angles as a function of reception time will be written out.
  blnWriteMetadata = .TRUE.

  ! This is the name of the metadata file being written out.
  strMetadataFileName = 'EmissionAngles.txt'

/
```

The name of an ANOPP input deck template, *Takeoff-Complete.inp*, is specified in the Configuration file, *AnoppComplete.config*. This input deck template is required for executing ANOPP as part of this functional module. This input deck template contains all the specifications of the aircraft frame and engine, as well as the ANOPP functional modules to be executed to obtain noise. The template contains the marker, `$$$ A2_GROUND_OBSERVER` that enables the AnoppComplete functional module to insert the current observer in the ANOPP input deck. The ground effects are turned off through the statement, `PARAM GROUND = .FALSE. $` in the template. The input deck template instructs ANOPP to execute and obtain noise from jet (JET), treated inlet (INLETT), treated aft fan (AFTFNT), GE Core (GECOR), gear (GEAR), flap (FLAP), and trailing edge (TRAL). It also instructs ANOPP to add the noise from all the sources and provide that as the Total noise, add JET, INLETT, AFTFNT, and GECOR as Engine noise, and add GEAR, FLAP, and TRAL as Airframe noise. The AnoppComplete functional module inserts the *Total*, *Engine*, and *Airframe* noise into the ANOPP2 Observer as the first, second, and the third result, respectively.

Step 6. Creating an ANOPP2 Mission: An ANOPP2 Mission is created through the following routine call:

```
intSuccess =                                                                       \
  ANOPP2.a2py_exec_create_mission                                                  \
   (pointer(self.intMissionTag), create_string_buffer (b""), self.intAtmosphereTag, \
    self.intFlightPathTag, self.nSourceTimes,                                        \
    self.nMaximumSingleTimeFunctionalModules, self.nTimeSeriesFunctionalModules,     \
    self.fltSourceTimes, self.intSingleTimeFunctionalModuleTags,                     \
    self.intTimeSeriesFunctionalModulesTags)
```

This instructs ANOPP2 what functional module is to be executed and also provides the tags of all the inputs required for executing this mission.

Step 7. Execute an ANOPP2 Mission: The ANOPP2 mission is executed through the following routine call:

```
intSuccess = ANOPP2.a2py_exec_execute_mission (self.intMissionTag)
```

Upon execution, the acoustic data corresponding to the observer is calculated and placed in the Observer in terms of Octave Band SPL (Sound Pressure Level).

Step 8. Obtaining the Noise Data: The acoustic data EPNL is calculated from the SPL through the following routine call:

```
intSuccess =                                                                \
  ANOPP2.a2py_obs_calc_metric                                               \
    (self.intObserverTag, self.nResults.value, self.intResultTags, a2_aa_epnl, \
     a2_obs_complete)
```

The EPNL value corresponding to the last node added through Step 4 is obtained through the Observer API routine call as shown below. Because the intent is to find the location corresponding to maximum total noise, the EPNL corresponding to the first result, that is, the *Total* noise is obtained.

```
intSuccess =                                                                    \
  ANOPP2.a2py_obs_get_epnl                                                      \
     (self.intObserverTag, self.intResultTags[0], self.nNodes.value, self.fltEpnl, \
      self.fltD, self.fltTimeRange)
```

The EPNL, the duration, and the time range are obtained for the observer position.

Step 9. Getting the Optimizing Variable Value: The goal of this AOIC is to find the position of maximum noise. This is equivalent to minimizing the reciprocal of the EPNdB value predicted. To avoid very small fractions, the reciprocal was multiplied with 1000. In this AOIC, the variable `Epndb_inverse` was minimized.

```
self.Epndb_inverse = 1000.0/float(self.fltEpnl.value)
```

The EPNL values were exported to a Tecplot-friendly file, *Epnl.out.dat* through the following routine:

```
intSuccess =                                                               \
  ANOPP2.a2py_obs_export                                                   \
     (self.intObserverTag, self.intResultTags[0],                          \
      create_string_buffer(bytes(self.strOutputFile)), a2_aa_epnl, a2_global, \
      a2_formatted, a2_tecplot)
```

Finally, the result in the Observer is deleted because it is no longer needed.

```
intSuccess =                      \
  ANOPP2.a2py_obs_delete_results \
     (self.intObserverTag, self.nResults.value, self.intResultTags, 0)
```

4.2 Anopp2Optimize

The Anopp2Optimize code should contain a function, *configure* that specifies the optimzer, the component, the objective variable, the design variable, as well as the optimizing parameters.

The driver, *CONMINdriver* is used in this optimization through the following statement:

```
self.add('driver', CONMINdriver())
```

The component, *Anopp2Component* is introduced in the *configure* function as *anopp2*.

```
self.driver.workflow.add('anopp2')
```

The objective variable, *Epndb_inverse* and the design variable, *x* defined in *Anopp2Component* are accessed in this function through *anopp2.Epndb_inverse* and *anopp2.x*.

```
self.driver.add_objective('anopp2.Epndb_inverse')
self.driver.add_parameter('anopp2.x', low=2400., high=3500.0)
```

The CONMIN specific settings used in this optimization are as follows:

```
self.driver.itmax = 30
self.driver.fdch = 0.001
self.driver.fdchm = 0.0001
self.driver.ctlmin = 0.01
self.driver.delfun = 0.01
self.driver.conmin_diff = True
self.iIteration = 0
```

4.3 test_aoic.py

The Python test code that runs the optimizer and iterates to find the location along the sideline that has maximum noise is *test_aoic.py*. This test code is executed after launching the OpenMDAO framework through the following command:

```
python test_Anopp2_optimize.py
```

The Python code executes ANOPP2 iteratively with various values of the design variable (the observer x coordinate) and obtains the value of the objective variable to be minimzed. The optimizer stops on finding a location at which the noise is maximum.

4.4 Results

The EPNdB values at various locations along the sideline parallel to the flight path were independently obtained at intervals of 100m. An analysis of the EPNdB distribution along the sideline indicates that there exists a local maximum around the location where the aircraft takes off, at approximately 2500m from the aircraft starting location. The EPNdB values corresponding to observer positions along the sideline from 2400m to 2700m were independently obtained at intervals of 6m. The EPNdB values at various observer locations along the sideline were also obtained through this *AOIC*. These results are plotted as *EPNdB along a sideline*.

The observer locations and the corresponding EPNdB values as obtained through *AOIC* are also plotted in *EPNdB along a sideline* as a sequence of steps (red lines and dots). The location as found by OpenMDAO that corresponds to maximum EPNdB value along the sideline is at **2538.5m**.

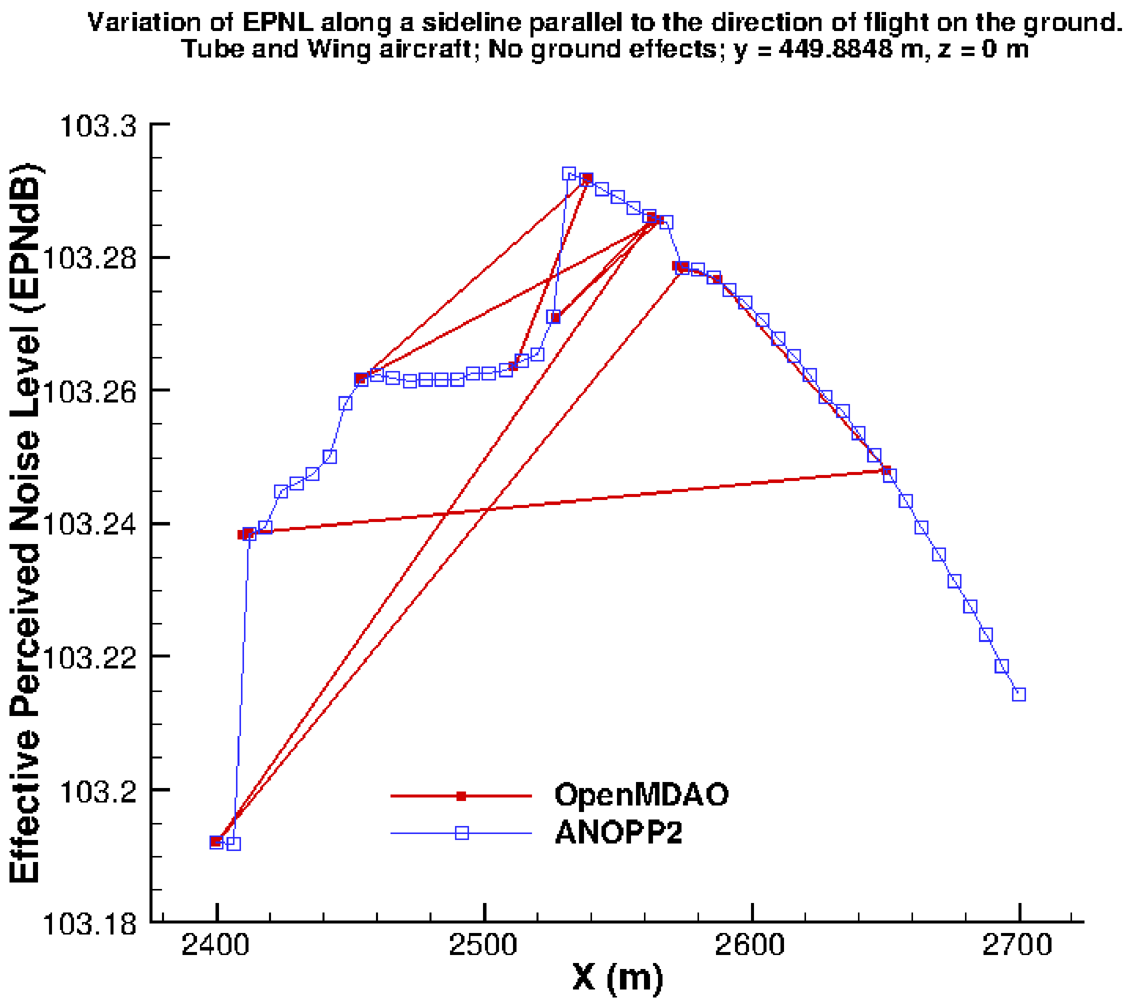

Fig. 4.1: Variation of EPNL along a sideline showing the sequence of steps followed by OpenMDAO in arriving at the location corresponding to maximum EPNdB

CHAPTER

FIVE

SOURCE DOCUMENTATION

The source code *AOIC.py* that uses ANOPP2 routines to identify the location along the sideline of maximum noise is provided as *AOIC.py*.

5.1 AOIC.py

```
__all__ = ['Anopp2']

# Import the Component class from the OpenMDAO Main API.
from openmdao.main.api import Component

# Import Float class from the Datatypes library in OpenMDAO.
from openmdao.lib.datatypes.api import Float

# Import the OpenMDAO library components. We need to import the External Code.
from openmdao.lib.components.external_code import ExternalCode

# Import the FileParser and InputFileGeneration functions residing in Filewrap utility
# in OpenMDAO
from openmdao.util.filewrap import FileParser, InputFileGenerator

# Import the Float datatype defined in OpenMDAO library required in AOIC.
from openmdao.lib.datatypes.api import Float

# Import Numpy as np
import numpy as np

# Import the ANOPP2 API python interface.
from ANOPP2_api import *

# Import the ctypes.
from ctypes import *

# Import the OS.
import os

from openmdao.main.api import Assembly
#from openmdao.lib.drivers.api import SLSQPdriver

# Import the Constrained Optimizer driver from the OpenMDAO Optimizers library.
from openmdao.lib.drivers.api import CONMINdriver


```

```

#===================================================================================
# This is the Anopp2Component class that receives the External code. This class is
# the crux of AOIC. This uses ANOPP2 function calls to perform various tasks for
# predicting noise.  Details of these functions, their syntax, purpose, and usage are
# explained in several ANOPP2 Manuals.  Users of ANOPP2 in OpenMDAO are required to
# develop this Anopp2Component specific to their problem.
#
# In this problem, the location of the Observer along a sideline parallel to the
# aircraft takeoff on the ground was determined at which the noise (Effective
# Perceived Noise Levels (EPNL)) is maximum. This determination is made using
# OpenMDAO by minimizing the reciprocal of the EPNL.
#===================================================================================
class Anopp2Component(ExternalCode):
  """
  All ANOPP2-capable components should be subclasses of Anopp2Component.

  By subclassing Anopp2Component, any component should have easy access to ANOPP2's
  subcomponents, such as Observer, FlightPath, etc.
  Refer to Documentation under Anopp2Component for additional information.
  """

  # This is the starting x location of the observer along the sideline, in terms of
  # meters. This is set as 2,410 m, the origin being the location at which the flight
  # originates.
  x = Float(2410.0, iotype='in', desc='The variable along the direction of flight')

  # This is the inverse of the EpndB. This is the variable that will be optimized or
  # minimized.  This variable will be returned by this class.
  Epndb_inverse = Float(iotype='out', desc='The inverse of the EPNdB')

  # Tags for the Atmosphere and Flight Path Data Structures. Although these Data
  # Structures will not be used in this demo, the variables are still required as
  # input for the routine that creates the mission.
  intAtmosphereTag = c_int(0)
  intFlightPathTag = c_int(0)

  #  A tag for the observer, which will hold the noise results parsed from the ANOPP
  #  output.
  intObserverTag = c_int(0)

  # A tag for the Functional Module.  For this demonstration program the Funtional
  # Module AnoppComplete will be used. A tag name must be declared as follows. This
  # tag is set by the a2py_exec_create_functional_module routine call.
  intAnoppCompleteTag = c_int(0)

  # This is the number of input Data Structure tags that are required for the
  # Functional Module.  In this case, for the AnoppComplete Functional Module there
  # will be 0 inputs.
  nInputs = 0

  # This is the array of input Data Structure tags, which again, will be of size 0.
  intInputTags = (c_int * nInputs)()

  # This is the number of results from the Functional Module stored in the Observer.
  nResults = c_int(0)

  # An array of tags used to access the results in the Observer Data Structure. A
  # pointer to this array is returned by the AnoppComplete Functional module.
  intResultTags = pointer(c_int(0))

  # ---------------------------------------------------------------------------
  # Integers and arrays for holding the Functional Module tags and passing them into
  # the Mission.  ANOPP2 has 2 types of Functional Modules: 'Time Series' or 'Single
  # Time'. The AnoppComplete Functional Module is of type 'Time Series'. Even if a
  # user is only utilizing one type of Functional Module, the definitions for arrays
  # and parameters need to be provided to the mission for both types.
  #----------------------------------------------------------------------------

  # A tag for the mission, which defines which Function Modules are executed.
  intMissionTag = c_int(0)

  # The number of Time Series Functional Modules to be called in the mission.
  # For this demonstration there will be 1 (AnoppComplete Functional Module)
  nTimeSeriesFunctionalModules = 1

  # The array of Time Series functional module tags.  This is allocated to size 1
  # because there is 1 Time Series Functional Module in this prediction: the
  # AnoppComplete Functional Module.
  intTimeSeriesFunctionalModulesTags = (c_int * nTimeSeriesFunctionalModules)()

  # Single Time definitions are set because the routine to create the mission requires
  # these arguments. This demonstration does not have any Single Time Functional
  # Modules, therefore all these are set to null or zero values.

  # The number of source times for the Single Time Functional Module (time in the
  # mission when specific functional modules are executed).  Since there is only a
  # Time Series Functional Module, this is not used and is set to 0.
  nSourceTimes = 0

  # The maximum number of Single Time Functional Modules executed at a single source
  # time. Since there are no Single Time Functional Modules defined in this
  # demonstration, this is set to zero.
  nMaximumSingleTimeFunctionalModules = 0

  # A 2-dimensional array of Single Time Functional Module tags.  The first dimension
  # is equal to the number of source times, the second is equal to the max number of
  # Functional Modules executed per source time.  These are both zero since this
  # demonstration does not have any Single Time Functional Modules
  intSingleTimeFunctionalModuleTags = pointer(c_int(0))

  # An array to store the source times.  These can be retrieved from the Flight Path
  # API after the Flight Path is created.  However, since there are no Single Time
  # Functional Modules, this is allocated to size zero (nSourceTimes is 0).
  fltSourceTimes = pointer(c_float(0))

  # ---------------------------------------------------------------------------
  # An integer success value.  This integer is returned from ANOPP2 API routine calls
  # to communicate the state of a given operation.  A return of 0 means success,
  # anything other than 0 indicates failure. Note the code execution does not depend
  # on this value, hence it is left to the user to check for success and take
  # appropriate action.
  # ---------------------------------------------------------------------------
  intSuccess = c_int(0)

  # This is the number of nodes existing in the Observer.
```

```
  nNodes = c_int(0)

  # This is the Epnl corresponding to this node.
  fltEpnl = c_float(-300.0)

  # This is the Duration factor determined while calculating EPNL.
  fltD = c_float(0)

  # This is the minimum and maximum time of the EPNL integration.
  fltTimeRange = (c_float * 2)(*[0.0, 0.0])

  # This is an array of the position of the last node in the Observer Point Cloud.
  fltPosition = (c_float * 3)(*[0.0, 0.0, 0.0])



  #==============================================================================
  # The Anopp2Component is initialized in this function. Several local variables are
  # also initialized. An Observer is created based on the details provided in
  # observer.config.
  #==============================================================================
  def __init__ (self):
    """ Initialize the Anopp2 class.
    """
    super(Anopp2Component, self).__init__()

    #============================================================================
    # Step 1.
    # The ANOPP2 API must be initialized before any of the routines can be executed
    # and should have false as it's argument.
    #============================================================================
    ANOPP2.a2py_exec_init_api ()

    #============================================================================
    # Step 2.
    # Create the necessary Data Structures required by the Anopp Complete Functional
    # Module. Set the tag numbers to zero for the Data Structures that ANOPP will
    # generate internally when it is executed.  This are necessary becuase the
    # variables are requried in the definition of the routine that creates the
    # mission. The Observer must be given to the Functional Module, so it must be
    # created from a configuration file.
    #============================================================================
    self.intAtmosphereTag = 0
    self.intFlightPathTag = 0

    #============================================================================
    # Step 3.
    # Construct an observer point (only a single point is used for serial execution)
    # from a configuration file.  This is done by using the create function of the
    # Observer API and passing a blank observer tag and the file name.  The necessary
    #  configuration file accompanies this demonstrator..
    #============================================================================
    intSuccess =             \
      ANOPP2.a2py_obs_create \
       (pointer(self.intObserverTag), create_string_buffer (b"observer.config"))
```

```
#==================================================================================
# The execute function gets executed by OpenMDAO repeatedly until the specified
# exit criteria are met. The steps followed in this function are:
#    1. Obtain the number of nodes available in the observer.
#    2. Obtain the position vector of the lsat node in the observer.
#    3. Replace the x-coordinate of this position vector with that provided by the
#       OpenMDAO optimizer.
#    4. Insert a new node in the observer with a position vector as defined through
#       steps 2 through 4.
#    5. Create an AnoppComplete Functional Module based on the config file,
#       "AnoppComplete.config".
#    6. Create a mission for running AnoppComplete functional module and executing
#       ANOPP to obtain noise due to Jet, Inlet, Aft Fan, Core, Gear, Flap, and
#       Trailing edge are all added to obtain the total noise. This noise is then
#       propagated to the ground observer.
#    7. Execute the mission to obtain the total noise on the ground observer.
#    8. Obtain the number of results in the Observer.
#    9. Obtain the number of nodes in the Observer.
#   10. Calculate the EPNL from the acoustic data predicted by the AnoppComplete
#       functional module.
#   11. Get the EPNL value of the last node.
#   12. If the EPNL value is valid, get its inverse. The inverse of the EPNL is
#       numerically small. Multiply it with 1000 and treat this as EpndB_inverse.
#   13. Export the results to a Tecplot-friendly file.
#   14. Delete the results because we do not need it any more.
#==================================================================================
def execute(self):
  """ Obtain a given Observer location, find the noise at that location. Use such
  noise values to optimize and locate the Observer location that has the maximum
  noise.
  """

  #================================================================================
  # Step 4. We need to introduce a new node at a location provided by the optimizer.
  # To do that, we need to first find the location of the last node in the observer,
  # replace the x coordinate of this node location with that provided by the
  # optimizer, and then introduce it as a new node in the point cloud.
  #================================================================================

  # Get the number of nodes in the Observer.
  intSuccess = ANOPP2.a2py_obs_number_of_nodes (self.intObserverTag, self.nNodes)

  # Get the coordinates of the last node.
  intSuccess =                \
    ANOPP2.a2py_obs_position \
      (self.intObserverTag, self.nNodes, 0.0, a2_global, self.fltPosition)

  # Replace the x coordinates of this array with that from the class. Rest of the
  # coordinates remain the same.
  self.fltPosition[0] = self.x

  # Insert a new node to the Point Cloud in the Observer.
  intSuccess = ANOPP2.a2py_obs_new_node (self.intObserverTag, self.fltPosition)

  #================================================================================
  # Step 5.
  # Create the AnoppComplete Functional Module.
  #================================================================================

  # Create the AnoppComplete Functional Module by passing an empty tag number (to be
  # filled), the configuration file name, the number of inputs, the list of Data
  # structure Tags, a pointer to the Observer tag, the number of results, and a
  # pointer to the array of result tags. The number of results and the result tags
  # are returned from the create function and can be used to later access the data
  # stored in the Observer Data Structure.
  intSuccess =                                                                   \
    ANOPP2.a2py_exec_create_functional_module                                    \
     (pointer(self.intAnoppCompleteTag),                                        \
      create_string_buffer (b"AnoppComplete.config"), self.nInputs,             \
      self.intInputTags, pointer(self.intObserverTag), pointer(self.nResults), \
      pointer(self.intResultTags))

  #==================================================================================
  # Step 6.
  # Create the Mission.  The Mission is the definition of what Functional Modules
  # will be executed. In this example, only the AnoppComplete Functional Module will
  # be executed for all time.
  #==================================================================================

  # Since the only Time Series Functional Module being used is AnoppComplete, assign
  # it to the first and only index of the array (if there was a second Time Series
  # Functional Module, it would be assigned to the second index of the array).
  self.intTimeSeriesFunctionalModulesTags [0] = self.intAnoppCompleteTag

  # To create a mission, each Functional Module tag must be added to the Mission
  # according to its type placed in arrays of the appropriate structure.  If Single
  # Time Functional Modules exist, then their tags would be placed into a two
  # dimensional array corresponding to waypoints and number of Single Time
  # Functional Modules. Time Series Functional Modules have their tags placed in a
  # one dimensional array. The routine to create a Mission also requires the
  # waypoint times, and the Atmosphere and Flight Path tags, regardless of  whether
  # they are being used. In this case they are not, so their values are set to zero.
  # Note: This demonstration program has only one Functional Module: AnoppComplete.
  # This is a Time Series Functional Module and therefore, all Single Time
  # Functional Module inputs are left empty.  The only reason they are needed is
  # because the a2py_exec_create_mission requires them.
  intSuccess =                                                                     \
    ANOPP2.a2py_exec_create_mission                                                \
     (pointer(self.intMissionTag), create_string_buffer (b""),                    \
      self.intAtmosphereTag, self.intFlightPathTag, self.nSourceTimes,            \
      self.nMaximumSingleTimeFunctionalModules, self.nTimeSeriesFunctionalModules, \
      self.fltSourceTimes, self.intSingleTimeFunctionalModuleTags,                \
      self.intTimeSeriesFunctionalModulesTags)

  #==================================================================================
  # Step 7.
  # Execute the Mission.  The Mission performs the noise prediction.  This call
  # tells the Mission to execute all Functional Modules at the time specified. This
  # will in turn execute ANOPP, producing a fort.4 that will be renamed to the
  # output specified in the configuration file
  #==================================================================================

  #  Call the routine that executes the Mission
  intSuccess = ANOPP2.a2py_exec_execute_mission (self.intMissionTag)

  # Get the number of results in the Observer.
  intSuccess = \
    ANOPP2.a2py_obs_number_of_results (self.intObserverTag, self.nResults)

  #==================================================================================
  # Step 8.
  # Calculate metrics and report the results. After the Mission is executed, the
  # Observer Data Structure contains the prediction results. These results can be
  # accessed to calculate derived metrics and reported to an external file using
  # Observer API routine calls.  To calculate metrics, the Observer API provides
  # specific acoustic analysis functionality that can be invoked via input
  # enumerators.  A list of the enumerators available for the Observer API routines
  # can be found in the Observer API User's Manual.
  #==================================================================================

  # Get the number of nodes in the Observer.
  intSuccess = ANOPP2.a2py_obs_number_of_nodes (self.intObserverTag, self.nNodes)

  # The first step is to calculate any derived metrics. For this demonstrator, EPNL
  # is calculated. The first argument in the call is the observer tag. The second
  # argument is the array containing the result tags associated with the observer
  # locations.  In this case there are three results, Engine, Airframe, and Total.
  # The third argument is an enumerator for what noise metric will be calculated and
  # returned by this call. For this demonstration the metric is EPNL. The last
  # argument is an enumerator that tells the Observer API that the EPNL has to be'
  # calculated based on the complete time history.  This is defined by the
  # enumerator, a2_obs_complete.
  intSuccess =                                                                   \
    ANOPP2.a2py_obs_calc_metric                                                  \
     (self.intObserverTag, self.nResults.value, self.intResultTags, a2_aa_epnl, \
      a2_obs_complete)

  # Get the EPNL value for the last node which is the node that was inserted into
  # the Observer.
  intSuccess =                                                         \
    ANOPP2.a2py_obs_get_epnl                                           \
      (self.intObserverTag, self.intResultTags[0], self.nNodes.value, \
       self.fltEpnl, self.fltD, self.fltTimeRange)

  # Ensure that the EPNL obtained from the Observer is greater than zero and is not
  # none.
  if (self.fltEpnl is not None) and (self.fltEpnl.value > 0.0):

    # Set the Epndb inverse as the reciprocal of the Epndb.
    self.Epndb_inverse = 1000.0/float(self.fltEpnl.value)

  # Print the Location and the EPNL values to the screen.
  print ("Location: ", self.fltPosition[0])
  print ("EPNL: ", self.fltEpnl.value)

  # Set the Output file name for exporting the EPNL to a file.
  self.strOutputFile = "Epnl.out.dat"

  # Export the result to a file.
  intSuccess =                                                                 \
    ANOPP2.a2py_obs_export                                                     \
      (self.intObserverTag, self.intResultTags[0],                             \
       create_string_buffer(bytes(self.strOutputFile)), a2_aa_epnl, a2_global, \
       a2_formatted, a2_tecplot)

  # Delete the results because we don't need them any more.
  intSuccess =                     \
    ANOPP2.a2py_obs_delete_results \
      (self.intObserverTag, self.nResults.value, self.intResultTags, 0)



#=====================================================================================
# This class contains the optimizer. It defines the design variable (the variable
# that has to be varied) and the parameter to be minimized (Epndb_inverse) and calls
# the Anopp2Component class repeatedly. It also specifies the driver to be used for
# optimization and the criteria for optimizing and exiting the optimization.
# This class performs the following steps:
#   1. It adds the Driver to be used and assigns a name to it. Two drivers have been
#      tried out: SLSQPdriver and CONMINdriver.
#   2. It adds the Anopp2Component class and assigns a name to it.
#   3. It adds this Anopp2Component to the Workflow of the driver.
#   4. It specifies the interval to display the optimization details.
#   5. It then adds the Objective variable and the parameter variable to be minimed.
#   6. Depending on the driver chosen it specifies the set of parameters for
#      optimization.
#=====================================================================================
class Anopp2Optimize(Assembly):
"""Unconstrained Optimization to locate the Observer location corresponding to
maximum EpndB"""



def configure(self):

  # Create an optimizer instance (Uncomment the following statement if SLSQP driver
  # is to be used, and comment the next statement.
#    self.add('driver', SLSQPdriver())
  self.add('driver', CONMINdriver())

  # Create Anopp2 component instance.
  self.add('anopp2', Anopp2Component())

  # Iteration hierarchy
  self.driver.workflow.add('anopp2')

  # SLSQP Flags
  self.driver.iprint = 1

  # Objective
  self.driver.add_objective('anopp2.Epndb_inverse')

  # Design variable
  self.driver.add_parameter('anopp2.x', low=2400., high=3500.0)

  # Set the SLSQP-specific settings. Uncomment the following if SLSQP Driver is to
  # be used.
#    self.driver.accuracy = 1.0e-08
#    self.driver.maxiter = 50

  # Set the CONMIN-specific settings. Comment the following if SLSQP Driver is to be
  # used.
  self.driver.itmax = 30
  self.driver.fdch = 0.001
  self.driver.fdchm = 0.0001
  self.driver.ctlmin = 0.01
  self.driver.delfun = 0.01
  self.driver.conmin_diff = True
  self.iIteration = 0
```

The code for testing this *AOIC.py* is provided as *test_aoic.py*.

5.2 test_aoic.py

```
#===================================================================================
# This program is a unit test that tests the Anopp2 optimization problem.
#===================================================================================

# Import the unittest module/class.
import unittest

# Import set_as_top from the OpenMDAO Main API.
from openmdao.main.api import set_as_top

# Import Anopp2Optimize class from the anopp2 folder.
from anopp2.anopp2 import Anopp2Optimize



#===================================================================================
# This class runs the unit tests that in turn tests the Anopp2 Optimization problem.
#===================================================================================
class Anopp2TestCase(unittest.TestCase):

  # There is nothing to setup here.
  def setUp(self):
    pass

  # There is nothing to teardown here.
  def tearDown(self):
    pass

  # This function tests the ANOPP2 component.
  def test_Anopp2(self):

    # Import the time module so we can assess the time taken to run this test.
    import time

    # This is the iteration number.
    iIteration = 0

    # This is the increment in x that we want to jump to determine a local maximum.
    fltdelta_x = 100.0

    # Set the current optimization problem as ANOPP2 Optimize.
    opt_problem = Anopp2Optimize()

    # Run the set_as_top function for the Anopp2 Optimize problem.
    set_as_top(opt_problem)

    # Set the time as tt.
    tt = time.time()

    # Execute the Anopp2Optimize
    opt_problem.run()

    # Write messages on the screen when a maximum noise is found.
    print "\n"
    print "Local Maximum found at (%f)" % opt_problem.anopp2.x
    print "EPNdB at this maximum (%f)" % opt_problem.anopp2.fltEpnl.value
    print "Elapsed time: ", time.time()-tt, "seconds"

if __name__ == "__main__":
    unittest.main()
```

CHAPTER

SIX

PACKAGE METADATA

- **classifier**:

```
Intended Audience :: Science/Research
Topic :: Scientific/Engineering
```

- **description-file:** README.txt
- **keywords:** openmdao
- **name:** AOIC
- **requires-dist:** openmdao.main
- **requires-python**:

```
>=2.7
<3.0
```

- **static_path:** ['_static']
- **version:** 1.0

[LOPES2014A] Lopes, L. V. and Burley, C. L., *ANOPP2 Command Executive API Reference Manual*, National Aeronautics and Space Administration, version 1.0.0, July 2014.

[LOPES2014B] Lopes, L. V. and Burley, C. L., *ANOPP2 Observer API Reference Manual*, National Aeronautics and Space Administration, version 1.0.0, July 2014.

[LOPES2014C] Lopes, L. V. and Burley, C. L., *ANOPP2 Functional Module Manual*, National Aeronautics and Space Administration, version 1.0.0, July 2014.