Rotorcraft Optimization Tools: Incorporating Rotorcraft Design Codes into Multi-Disciplinary Design, Analysis, and Optimization

Larry A. Meyn
Aerospace Engineer
NASA Ames Research Center
Moffett Field, CA

Abstract

One of the goals of NASA's Revolutionary Vertical Lift Technology Project (RVLT) is to provide validated tools for multi-disciplinary design, analysis and optimization (MDAO) of vertical lift vehicles. As part of this effort, the software package, RotorCraft Optimization Tools (RCOTOOLS), is being developed to facilitate incorporating key rotorcraft conceptual design codes into optimizations using the OpenMDAO multi-disciplinary optimization framework written in Python. RCOTOOLS, also written in Python, currently supports the incorporation of the NASA Design and Analysis of RotorCraft (NDARC) vehicle sizing tool and the Comprehensive Analytical Model of Rotorcraft Aerodynamics and Dynamics II (CAMRAD II) analysis tool into OpenMDAO-driven optimizations. Both of these tools use detailed, file-based inputs and outputs, so RCOTOOLS provides software wrappers to update input files with new design variable values, execute these codes and then extract specific response variable values from the file outputs. These wrappers are designed to be flexible and easy to use. RCOTOOLS also provides several utilities to aid in optimization model development, including Graphical User Interface (GUI) tools for browsing input and output files in order to identify text strings that are used to identify specific variables as optimization input and response variables. This paper provides an overview of RCOTOOLS and its use.

Introduction

The ability to hover, as well as takeoff and land vertically is highly desirable for numerous air vehicle use cases. These include human and cargo transportation systems that are independent of airports and runways, as well as surveillance and inspection missions that require loitering capabilities. To support the development of new vehicles with vertical lift capabilities, the National Aeronautics and Space Administration (NASA) created the Revolutionary Vertical Lift Technology (RVLT) project. Part of the RVLT project's goals is to support the development of a multidisciplinary design and optimization (MDAO) process for the design of vertical lift aircraft. This involves the coupling of codes from multiple disciplines, required for the design of a new vertical lift vehicle, into a coordinated optimization process.

Presented at the AHS Technical Conference on Aeromechanics Design for Transformative Vertical Flight, San Francisco, CA, January 16-19, 2018.

Two rotorcraft design codes that NASA researchers use in this process are NDARC and CAMRAD II. The NASA Design and Analysis of RotorCraft (NDARC) is a conceptual aircraft design software tool with a primary focus on rotorcraft.[1] The Comprehensive Analytical Model of Rotorcraft Aerodynamics and Dynamics II (CAMRAD II) is a comprehensive rotorcraft analysis tool.[2,3] An example of integrating CAMRADII and NDARC for the design and performance analysis of compound helicopters is presented in Ref. 4, in which the design process was basically a manual integration where code inputs were manually altered based on the results of previous iterations of the codes. Manual processes such as this are time consuming, which highlights the need to automate the process using a tool such as OpenMDAO.[5]

OpenMDAO is an open-source computing platform for multi-disciplinary optimization, written in Python, that is currently under development at NASA Glenn Research Center and is supported by the RVLT project. OpenMDAO provides a framework for connecting "components" which provide design and analysis calculations, and then solving them in a tightly-coupled manner using a variety of both gradient-free and gradient-based optimization methods.

Examples of using NDARC and CAMRAD II in an OpenMDAO-based design study are provided in Refs. 6 and 7. These examples show both the feasibility and desirability of having NDARC and CAMRAD II available as components for use in OpenMDAO optimizations and analyses. Both NDARC and CAMRAD II use file-based inputs and outputs, so integrating them into an OpenMDAO optimization required the development of file-based application wrappers, so that they could be used as "components" in an OpenMDAO optimization. These wrappers would take design variable input values from the OpenMDAO framework, incorporate them into input files, execute the file-based application, parse the output files and then provide design variable output values to the OpenMDAO framework. The NDARC and CAMRAD II wrappers developed for OpenMDAO in these previous studies were not primarily developed for distribution to other designers and they used an earlier, deprecated, version of OpenMDAO. The goal of the RVLT sponsored effort described in this paper is to develop flexible, easy-to-use and well-documented OpenMDAO design application wrappers that are intended for distribution and use by rotorcraft designers.

Figure 1 shows a flowchart depicting the role of RCOTOOLS in an OpenMDAO optimization process for the design of a new vertical lift vehicle. The process starts with an initial, preliminary, aircraft design definition that includes new technologies and vehicle topologies. These are used to construct models for several design codes that have OpenMDAO wrappers. These may include aerodynamics and structures models for rotors and/or other vehicle components, as well as codes modeling geometry, handling qualities or engine performance. An OpenMDAO problem is set up to connect these models to CAMRAD II and NDARC for a defined computation sequence where design variable values are passed between the model components. RCOTOOLS handles the interface to and from CAMRAD II and NDARC within the optimization problem. OpenMDAO controls model execution in the optimization loop and adjusts design variable values as needed to meet the optimization criteria. For gradient-based optimizations, OpenMDAO may also execute some models individually to determine partial derivatives for some of the design variables via finite difference methods.

The NDARC and CAMRAD II application wrappers provided in RCOTOOLS are intended to be "light-weight," with minimal computational overhead. The wrappers are not intended to serve as full-blown application front-ends, such as the AIDEN application is for NDARC.[8] The RCOTOOLS

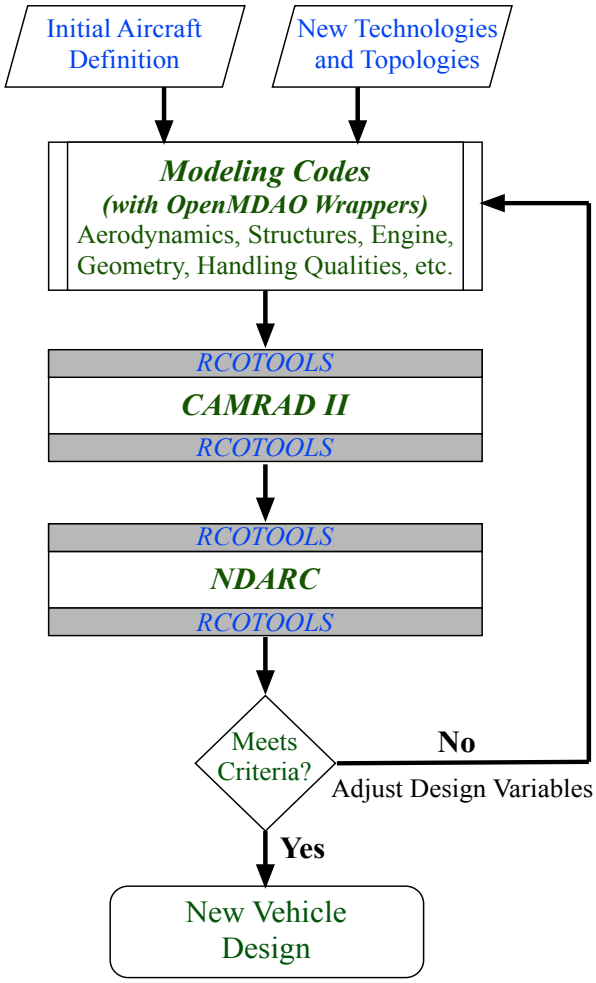

Figure 1: A flowchart showing how RCOTOOLS could be used in an OpenMDAO optimization for the design of a new vertical lift vehicle.

wrappers do not validate inputs beyond simple namelist format validation and any auxiliary input files, specified in the input deck, are not read and parsed. NDARC and CAMRAD II input files should be validated and tested before use with the RCOTOOLS wrappers. The wrappers in RCOTOOLS consist of core, generic Python wrappers that read, modify and execute program input files and read and extract data from solution files. These core wrappers do not rely on or use OpenMDAO software. Separate wrappers are provided that interface with OpenMDAO. These interface wrappers inherit from OpenMDAO component objects and are built on top of the core wrapper functions and objects. These OpenMDAO wrappers were designed to be flexible and work out-of-the-box for most needs. However, one of the reasons the core wrappers were developed, was to allow for easier development of custom OpenMDAO wrappers if the ones provided do not provide needed functionality. The core wrappers also facilitate adaption to any future changes

to the OpenMDAO application programming interface (API.)

An effort was made to use common code and similar APIs in the development of both the NDARC and CAMRAD II wrappers in RCOTOOLS. When new design code wrappers are added to RCOTOOLS, the goal is to maintain similar commonalities where possible. Much of the common code and functionality provided by RCOTOOLS resides in the `rcotools.utils` module which contains utility functions used by both the NDARC and CAMRAD II wrappers. An overview of these utilities will be presented next, followed by overviews for the NDARC and CAMRAD II wrappers.

Utilities

RCOTOOLS provides several utility functions that are used by multiple application wrappers. Convenience functions are also included that may prove useful to RCOTOOLS users. These utilities are located in the `rcotools.utils` module. Among the utilities provided are a Fortran namelist wrapper for reading, writing and manipulating namelists and applications for graphically browsing input and output files used by NDARC and CAMRAD II. An overview of these utilities follows.

Fortran Namelist Wrapper

Many science and engineering codes are written in Fortran and often use Fortran namelist files for input. This is true of both CAMRAD II and NDARC. NDARC also has an optional namelist representation of the output solution. An example of namelist input for NDARC is presented in Listing 1. The namelist format separates inputs into named groups and each group has a list of variable names and their assigned values. Variable names within a group are unique, so when multiple assignments are made within a group, the last assignment is the one that is used. The beginning of a group starts with a "`&`*`group_name`*" tag and the group ends with a "`/`", "`&END`" or "`$END`" tag. Complex variable assignments are being used, such as "`loc_rotor(2)%XoL = 0.00`," which includes both Fortran array and structure assignments. Of note, the group names '`DEFN`' and '`VALUE`' are being used repeatedly in this example. Although OpenMDAO includes a module for reading and writing Fortran namelists, the more complicated namelist assignments used by both NDARC and CAMRAD II are not supported. No third-party namelist parsers for Python that were capable of handling NDARC and CAMRAD II inputs were found, so a new namelist module was developed for RCOTOOLS.

Listing 1: Example namelist based on NDARC input.

```
&JOB open_status=1,&END
&DEFN action='ident',created='today',title='standard input',&END
!#############################################################################
&DEFN action='read file',file='gen1000.list',&END
&DEFN action='read file',file='airplane.list',&END
!=============================================================================
&DEFN quant='Geometry',&END
&VALUE
  ! fuselage reference
    loc_fuselage%FIX_geom='xyz',
  ! scaled geometry (INPUT_geom=2); x +aft, y +right, z +up
    loc_cg%XoL          =0.01,loc_cg%YoL          =0.00,loc_cg%ZoL          = 0.00,
    loc_fuselage%XoL    =0.00,loc_fuselage%YoL    =0.00,loc_fuselage%ZoL    = 0.00,
    loc_gear%XoL        =0.00,loc_gear%YoL        =0.00,loc_gear%ZoL        =-0.30,
    loc_rotor(1)%XoL    =0.00,loc_rotor(1)%YoL    =0.00,loc_rotor(1)%ZoL    = 0.20,
 &END
&DEFN quant='Cases',&END
&VALUE
   title='Airplane',
   TASK_size=1,TASK_mission=1,TASK_perf=1,
   OUT_aircraft=0,OUT_solution=1,
&END
&DEFN quant='Size',&END
&VALUE title='design',nFltCond=1,nMission=1,&END
&DEFN action='endofcase',&END
&DEFN action='endofjob',&END
```

Listing 2: Example code to read and modify a namelist.

```
1  from rcotools.utils import Namelist
2
3  # Create empty Namelist and then read in namelist file
4  mynml = Namelist()
5  mynml.read('namelist_xmpl.txt')  # use nowarnings=True to supressing warnings
6
7  # Get variable representing group at namelist index 7
8  geometry = mynml[7][1]
9
10 # Modify value in the existing group
11 geometry['loc_rotor'][1]['ZoL'] = 0.25
12
13 # Insert new groups before the group containing 'endofcase' (index = -2)
14 mynml.insert_group(-2, 'DEFN', {'quant': 'Solution'})
15 mynml.insert_group(-2, 'VALUE', {'trace_size': 1, 'trace_case': 0})
16
17 # Print out the modified namelist
18 print(mynml.writestr())
```

The `rcotools.utils.namelist` module has two major classes, `Namelist` and `NamelistGroup`. The `Namelist` class represents namelists as a list of (*name*, *value*) tuples[†], where *name* represents the group name and *value* is either a Python dictionary or a `NamelistGroup`. To support the inclusion of comments and non-namelist data, the tuples can also contain a pair stings. If *name* string begins with a "`%`", then *value* must be a string. Currently the following *name* strings beginning with "`%`," are supported: "`%comment_line`," "`%blank_line`" and "`%data_line`." When one of these *names* are encountered, then the *value* string contains the full text for one line of the file. Note that namelist group *names* may appear multiple times within a `Namelist`. In addition to special methods for reading, writing and manipulating namelists, `Namelist` objects implement all standard list methods for list manipulation. A simple `Namelist` list could look like the following:

```
[('%comment_line',     'Comment string'),
 ('GROUP1',            NamelistGroup   ),
 ('GROUP2',            NamelistGroup   ),
 ('%blank_line',       ''              ),
 ('GROUP3',            NamelistGroup   )]
```

The `NamelistGroup` class represents namelist groups as a case-insensitive, ordered dictionaries. In addition to namelist variables, they can also store namelist comments, comment lines and blank lines using special dictionary keys that begin with "%." All other keys represent namelist variable names. In addition to special methods for reading, writing and manipulating namelist groups, `NamelistGroup` objects implement all standard dictionary methods for dictionary manipulation.

For typical use, only the `Namelist` class needs to be imported, using either:

```
From rcotools.utils.namelist import Namelist
```

or:

```
from rcotools.utils import Namelist
```

Listing 2 is an example of using the RCOTOOLS' `Namelist` class to read, modify, and print the namelist presented in Listing 1. Importing the Namelist class is done in line 1. Line 4 creates a new Namelist object, `mynml`, and then in line 5, the "`read()`" method is used to read the namelist from a file. The resulting Namelist has 14 (*name*, *value*) tuples, which includes tuples for two comment lines. Line 8 assigns the NamelistGroup from the eighth (*name*, *value*) tuple in mynml to the variable, geometry. Since Namelist objects, like Python lists, have an initial index of 0, the index of the eighth tuple is 7 and since the NamelistGroup is the second item in the tuple, its index is 1. In line 11, the value for "loc_rotor(1)%ZoL" in the "`VALUE`" group is being set to 0.25 using the statement "`geometry['loc_rotor'][1]['ZoL'] = 0.25`". The "`loc_rotor`" index identifies the variable within the group. This variable is an array represented by a special list-like object in RCOTOOLS that, following Fortran convention, has the first item accessed using the index 1. The "%" symbol in the original namelist file indicates that this array contains a structure which is represented as a dictionary in the NamelistGroup and "`ZoL`" is the key identifying which item in the structure is to be accessed.

[†] Tuples are a Python structure representing an immutable sequence of items. These are defined in code as comma separated values and variables between parentheses.

Listing 3: NDARC namelist input modified using RCOTOOLS namelist utilities. Changes from listing 1 are highlighted.

```
&JOB open_status=1, &END
&DEFN action='ident', created='today', title='standard input', &END
 !##############################################################################
&DEFN action='read file', file='gen1000.list', &END
&DEFN action='read file', file='airplane.list', &END
 !==============================================================================
&DEFN quant='Geometry', &END
&VALUE
   ! fuselage reference
    loc_fuselage%FIX_geom='xyz', loc_fuselage%XoL=0.0, loc_fuselage%YoL=0.0,
    loc_fuselage%ZoL=0.0,
   ! scaled geometry (INPUT_geom=2); x +aft, y +right, z +up
    loc_cg%XoL=0.01, loc_cg%YoL=0.0, loc_cg%ZoL=0.0, loc_gear%XoL=0.0, loc_gear%YoL=0.0,
    loc_gear%ZoL=-0.3,            loc_rotor(1)%XoL=0.0,            loc_rotor(1)%YoL=0.0,
loc_rotor(1)%ZoL=0.25,
&END
&DEFN quant='Cases', &END
&VALUE
    title='Airplane',    TASK_size=1,    TASK_mission=1,    TASK_perf=1,    OUT_design=0,
OUT_perf=0,
    OUT_geometry=0, OUT_aircraft=0, OUT_solution=1,
&END
&DEFN quant='Size', &END
&VALUE title='design', nFltCond=1, nMission=1, &END
&DEFN quant='Solution', &END
&VALUE trace_size=1, trace_case=0, &END
&DEFN action='endofcase', &END
&DEFN action='endofjob', &END
```

The lines 14 and 15 insert a `DEFN` and a `VALUE` group, typical of NDARC input, into mynml `Namelist` using the "`inset_group`" mehthod. Finally, line 18 prints out the mynml `Namelist` as a string, which is shown in Listing 3. Changes to the original namelist, other than formatting, are highlighted in yellow.

Input and Output File Viewers

The application wrappers in RCOTOOLS allow specification of input and output variables using Fortran namelist-style strings. These strings specify the sequence of dictionary keys and list indices that identify the variables in input and output data structures. Although these strings are constructed using a definitive logical sequence, generating them manually is error prone and time consuming. To address this problem, a set of Graphical User Interface (GUI) applications are provided that allow users to visually view and browse through existing input and output data files to find variables and display namelist-style access strings that are used by the RCOTOOLS wrappers. The currently available viewer applications and the commands used to call them are listed in Table 1.

Table 1: List of file viewers in RCOTOOLS.

Command	Viewer Name
niview	NDARC Input Viewer
nsview	NDARC Solution Viewer
ciview	CAMRAD II Input Viewer
csview	CAMRAD II Solution Viewer

These viewers have nearly identical form and function, the primary difference between them is the type of data file that they read and display. Because of their similarities, the use of only one of these applications, the NDARC Input Viewer, is described below.

If RCOTOOLS has been installed using either the terminal command "`python setup.py install`" or "`python setup.py develop`," then the NDARC Input Viewer can be launched from a terminal window using the `niview` command. `niview` has an optional argument, ***`filepath`***, that is the path of the NDARC input file to be viewed, as shown below:

niview [-h] [-i *filepath*]

Figure 2 shows a typical NDARC Input Viewer window after a solution has been loaded. Annotation (1) shows the button used to load a new NDARC input file name. Clicking this button brings up a file browser which is used to identify an NDARC input text file to be loaded. If the GUI is launched without a designated *`filepath`*, then this button is used to load one. Once a solution file is loaded, its contents can be browsed using a tree-view of the NDARC data structures. Clicking on a data structure item will reveal all of its sub-structures. If the data structure contains a scalar value or an array of scalar values, that data is displayed. Individual array items can also be viewed as a sub-structure of an array. A structure variable is highlighted when selected in the GUI, as shown in annotation (2). The namelist string used to access this structure variable by RCOTOOLS is then displayed in the "Namelist String" field of the GUI window, as shown in annotation (3). This string is pre-selected and can be copied to the clipboard using the "Copy" menu item

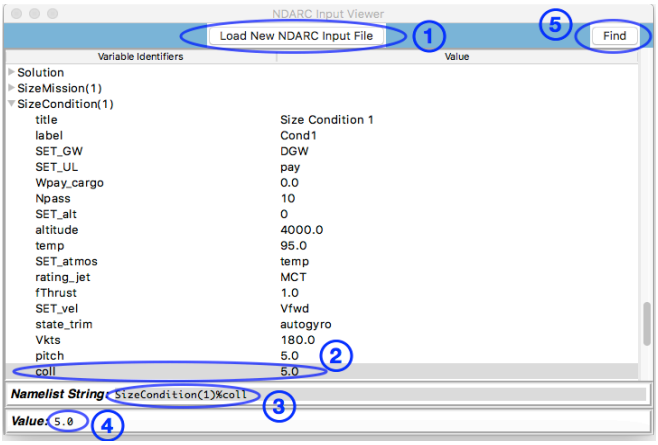

Figure 2: A typical niview window with annotations for several elements.

in the "File" menu or using the standard copy commands for the operating system being used. If this structure contains scalar data or an array of scalar data, then that data (or array) is displayed in the "Value" field as shown in annotation (4). To save time browsing through a data structure looking for specific items, a find dialog can be used, which is activated by clicking the "Find" button shown in annotation (5).

Figure 3 shows a typical "Find" dialog window for the viewer applications. This example is for an NDARC input data structure viewed with `niview`. This dialog will search for text in the key names for items in the data structure. The text is entered in the "Find Key Text" field as shown in annotation (1). The search is case-insensitive, so capitalization does not matter. The search parameters can be modified using the "Matching" checkboxes as shown in annotation (2). Checking "Whole word" will only find keys where the search text matches entire words within the key string. Checking "Values too" will extend the search to include searches of the string equivalent of data structure values, numerical values included. A search is initiated by pressing either the "Find Next" or "Find All" buttons shown in annotation (3). If any matches are found, they are displayed in the panel below. Both the namelist access string and the data structure value is displayed. Selecting a match, as shown in annotation (4) will display the namelist access string and the data structure value in text fields as shown in annotation (5) and (6). The selected item will also be displayed and selected in the main viewer window, as shown in Fig. 2.

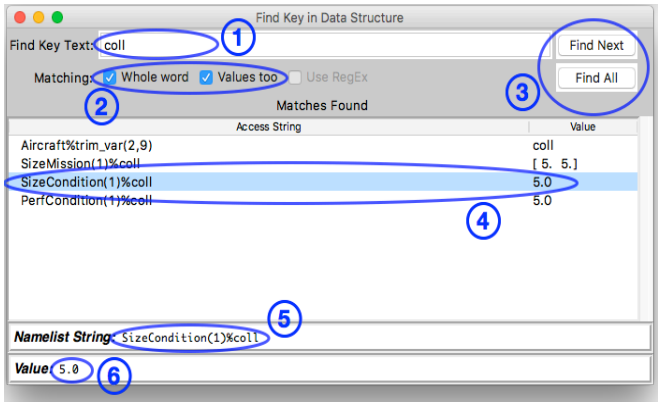

Figure 3: A typical niview "Find" dialog window with annotations for several elements.

NDARC Wrappers

The NDARC core wrapper is defined in the `rcotools.nadarc.ndarc_core` module. Two versions of the OpenMDAO wrappers for NDARC are currently provided, one for OpenMDAO version 1.7+ and preliminary one for OpenMDAO version 2+. The OpenMDAO 1.7+ wrapper is defined in the `rcotools.nadarc.ndarc_mdao1` module, while the OpenMDAO 2+ wrapper is defined in the `rcotools.nadarc.ndarc_mdao2` module.

NDARC Core Wrappers

NDARC uses internal data structures that are organized into a single hierarchical structure that is represented in outline form in Fig. 4. The structure variable names presented in this figure are the names used by NDARC and are intended to be descriptive of the structure content. Indentation level is used to show which structures are contained in a higher-level structure within the hierarchy. NDARC uses namelist formatted input files to initialize the values in these structures and the NDARC solution output file is a

representation of the final value of these data structures in a namelist format. However, the group names and the sequencing of the groups in the solution file namelist is very different from that of the input file namelists. One of the goals of the RCOTOOLS NDARC core wrappers is to hide the namelist formatting details as much as possible and allow the user to reference the data in both input and output files as members of a hierarchical data structure similar to what is presented in Fig. 4. This mapping to the NDARC internal data structures is not exact, especially for NDARC input, which has some variations to handle specialized input features. However, the file viewers `niview` and `nsview`, previously described, can be used to visualize and interrogate the RCOTOOLS data structures for NDARC input and output files.

The `rcotools.nadarc.ndarc_core` module defines two classes, `NdarcJob` and `NdarcSolution`. The `NdarcJob` class is used to read in an NDARC job namelist, modify the namelist and then execute the namelist. The `NdarcSolution` class is used to read in and parse an NDARC solution file (which is in namelist format). After parsing, the solution file variables can then be accessed as an ordered dictionary of NDARC data structures.

NdarcJob

The simplest use of `NdarcJob` would be as follows:

```
import rcotools.ndarc.ndarc_core as nc

# initialize NDARC job
njob = nc.NdarcJob(namelist, execution_directory,
                   NDARC_executable_path, **args)

#    *** make namelist modifications ***

# execute NDARC job
exitonerror = njob.execute(output_file_path)
```

The initialization argument, `namelist`, can be a `Namelist` object, a file object, a string containing namelist input, a string representing a file path or the Python value, `None`. NDARC execution scripts can be used as input, since the parser will ignore lines of text that do not contain namelist information. If the default value of `None` is provided, then the namelist can be read in later using the `NdarcJob.read()` method. The `Namelist` object is accessible using the `namelist` attribute of an `NdarcJob` object.

- Design
  - Cases
  - Size
    - SizeParam
    - FltCond(nfltmax)
    - FltState(nfltmax)
    - Mission(nmissmax)
      - MissParam
      - MissSeg(nsegmax)
      - FltState(nsegmax)
  - OffDesign
    - OffParam
    - Mission(nmissmax)
      - MissParam
      - MissSeg(nsegmax)
      - FltState(nsegmax)
  - Performance
    - PerfParam
    - FltCond(nfltmax)
    - FltState(nfltmax)
  - MapEngine
  - MapAero
  - Solution
  - Cost
    - CostCTM
  - Emissions
  - Aircraft
    - [Location]loc_cg
    - Weight
    - XAircraft
  - Systems
    - Weight
    - WFltCont
    - WDelce
- Fuselage
  - [Location]loc_fuselage
  - AFuse
  - Weight
  - WFuse
- LandingGear
  - [Location]loc_gear
  - AGear
  - Weight
  - WGear
- Rotor(nrotormax)
  - [Location]loc_rotor
  - [Location]loc_pylon
  - [Location]loc_pivot
  - [Location]loc_nac
  - PRotorInd
  - PRotorPro
  - PRotorTab
  - IRotor
  - DRotor
  - Weight
  - WRotor
- Wing(nwingmax)
  - [Location]loc_wing
  - AWing
  - Weight
  - WWing
  - WWingTR
- Tail(ntailmax)
  - [Location]loc_tail
  - ATail
  - Weight
  - WTail
- FuelTank(ntankmax)
  - [Location]loc_auxtank(nauxtankmax)
  - Weight
  - WTank
- Propulsion(npropmax)
  - Weight
  - WDrive
- EngineGroup(nengmax)
  - [Location]loc_engine
  - DEngSys
  - Weight
  - WEngSys
- JetGroup(njetmax)
  - [Location]loc_jet
  - DJetSys
  - Weight
  - WJetSys
- ChargeGroup(nchrgmax)
  - [Location]loc_charger
  - DChrgSys
  - Weight
  - WChrgSys
- EngineModel(nengmax)
- EngineParamN(nengpmax)
- EngineTable(nengmax)
- RecipModel(nengmax)
- CompressorModel(nengmax)
- MotorModel(nengmax)
- JetModel(njetmax)
- FuelCellModel(nchrgmax)
- SolarCellModel(nchrgmax)
- BatteryModel(ntankmax)
- FltState(nfltmax)
  - FltAircraft
  - FltFuse
  - FltGear
  - FltRotor(nrotormax)
  - FltWing(nwingmax)
  - FltTail(ntailmax)
  - FltTank(ntankmax)
  - FltProp(npropmax)
  - FltEngn(nengmax)
  - FltJet(njetmax)
  - FltChrg(nchrgmax)

Figure 4: Hierarchical grouping of data structures used by NDARC. (From NDARC 1.12 documentation.)

The `execution_directory` parameter is the directory in which the namelist NDARC will be executed. Before job execution, the current working directory will be saved, the working directory will then be changed to `execution_directory` for job execution. Following job execution, the working directory will be changed back to previous working directory. If no `execution_directory` is provided, then the parameter is set to the current working directory.

The `NDARC_executable_path` is the path to the NDARC executable. If the path is not provided, then the executable path is set to variable `rcotools.NDARC_CMD`, which is set in the `rcotools/rcotools.ini` configuration file.

Once a namelist is input, the `NdarcJob` object has several methods for interrogating and modifying the internal `Namelist` object before execution. Two methods are provided to modify NDARC job namelist, `updatecaseparams()` and `amendcase()`.

The `updatecaseparams()` method will update the last "Cases" namelist group in an NDARC input case using the dictionary `updatedict`. The targeted case is the last one in the input namelist, unless a `caseindex` is provided. Its usage is "`njob.updatecaseparams(updatedict, caseindex=-1)`".

The `amendcase()` method inserts namelist groups, `addgroups`, at the end of a NDARC job input case. The targeted case is the last one in the input namelist, unless a `caseindex` is provided. Its usage is "`njob.amendcase(addgroups, caseindex=-1)`".

To facilitate defining the namelist groups for use as the `addgroups` variable, `ndarc_core` provides the `getNdarcUpdate()` function. This function takes a dictionary of *key*, *value* pairs, where *key* is a namelist variable string and *value* is the desired corresponding value. This function converts these (the pairs??) to a `Namelist` for updating an `NdarcJob` using `amendcase()`. The following code shows how an example input dictionary is converted to namelist groups. (Note that an `OrderedDict` is not necessary, it (what?) just preserves the variable order in the resulting namelist.)

```
updatedict = {'Geometry%loc_tail(1)%XoL': 1.10,
              'Geometry%loc_tail(1)%YoL': 0.00,
              'Geometry%loc_tail(1)%ZoL': -0.25,
              'Rotor(1)%MODEL_int': 0,
              'Rotor(1)%SET_limit_rs': 2,
              'Rotor(1)%fPlimit_rs': 1.,
              'Rotor(1)%incid_hub': 0.}

updates = getNdarcUpdate(updatedict,
                         asString=True)
print(updates)
```

This would print the following namelist string.

```
&DEFN quant='Geometry', &END
&VALUE loc_tail(1)%XoL=1.1, loc_tail(1)%YoL=0.0,
 loc_tail(1)%ZoL=-0.25, &END
&DEFN quant='Rotor 1', &END
&VALUE MODEL_int=0, SET_limit_rs=2,
 fPlimit_rs=1.0, incid_hub=0.0, &END
```

The namelist variable strings provided to the `getNdarcUpdate()` function use the "`quant`" value for the DEFN/VALUE input pair as the first argument of the namelist variable string, preceding the first "`%`" symbol. If the quant value includes a number, such as "`Rotor 1`", then the number is converted to a namelist array index, for example: "`Rotor(1)`". The `niview` viewer utility can be used to identify these namelist strings from an NDARC input file. If the input file contains valid paths to imported files, the viewer will try to parse them and include their variables in the data structure.

According to NDARC input rules, `Namelist` definitions in the `addgroups` value passed to `amendcase()` will override previous definitions. Given this rule, the following two cautions are noted. First, for NDARC data structures like "`Rotor`", amendments to the first instance should be referred as "`Rotor(1)`" not just "`Rotor`", as this will actually add a new rotor. Second, if the copy command is used to create several objects, like "`Rotor`", then the amendment must specify updates to each copy for which the update is desired.

As a final note, if the `NdarcJob` methods do not fully provide the wanted functionality, one can work directly with the `namelist` attribute of an `NdarcJob` object using `Namelist` class method.

NdarcSolution

The `NdarcSolution` class reads and converts an NDARC solution file to an ordered dictionary representing data for NDARC data structures as shown in Fig. 4. `NdarcSolution` parses the namelist data in an NDARC solution file in two passes: the first pass identifies major sections in the solution file and the second pass performs detailed namelist parsing of these sections. To save execution time, detailed namelist parsing is only performed on requested sections. This process is referred to as "lazy" evaluation. Loading an NDARC solution file for lazy evaluation would be done using the code "`NdarcSolution(nmlpath, dolazy=True)`."

To properly parse an NDARC solution file, the parser needs information on the top-level data structures in the solution. The data structures Size, OffDesign, Performance and FltState have specific parsing procedures, but the other top-level data structures are parsed in a generic fashion. The currently defined additional top-level data structures are provided in the dictionary, `_toplevel`, which is defined in the `ndarc_core` module and is presented in Listing 4. This (what?) is a dictionary of *key*, *value* pairs, where the *key* is the data structure name and *value* is a bool(boolean?) indicating if the structure can have multiple instances and should be represented as an array. As new NDARC data structures are developed, they can be simply added to this dictionary in the source code.

Once a solution file is loaded, data structure values can be accessed using the `retrieve()` method, as follows:

```
import rcotools.ndarc.ndarc_core as nc
soln = nc.NdarcSolution(solution_path)
value = soln.retrieve(access_query)
```

The access query identifies a variable in the solution, where the variable is identified by either a namelist variable string or a list of (*key*, *index*) tuples. The (*key*, *index*) tuples are used to traverse the solution data structure to yield the requested variable value. An example access string would be:

```
Size%SizeMission(1)%MissParam%WFUEL_MISS(1)
```

The corresponding list of (*key*, *index*) tuples would be:

```
[('Size', None), ('SizeMission', 0),
 (MissParam, None), ('WFUEL_MISS', 0)]
```

For the `NdarcSolution` object, `soln`, this would be the same as:

```
soln['Size']['SizeMission'][0]['MissParam']
    ['WFUEL_MISS'][0]
```

Listing 4: Dictionary defining NDARC data structures, in rcotools.ndarc.ndarc_core.

```
# NDARC top level data structures
# Specifiy available NDARC data structures, (Name, isArray)
# Note: Size, OffDesign, Performance and FltState are handled seprately
_toplevel = Odict([('Cases', False),
                   ('MapEngine', False),
                   ('MapAero', False),
                   ('Solution', False),
                   ('Cost', False),
                   ('Emissions', False),
                   ('Aircraft', False),
                   ('Systems', False),
                   ('Fuselage', False),
                   ('LandingGear', False),
                   ('Rotor', True),
                   ('Wing', True),
                   ('Tail', True),
                   ('FuelTank', True),
                   ('Propulsion', True),
                   ('EngineGroup', True),
                   ('JetGroup', True),
                   ('ChargeGroup', True),
                   ('EngineModel', True),
                   ('EngineParam', True),
                   ('EngineTable', True),
                   ('RecipModel', True),
                   ('CompressorModel', True),
                   ('MotorModel', True),
                   ('JetModel', True),
                   ('FuelCellModel', True),
                   ('SolarCellModel', True),
                   ('BatteryModel', True),
                   ])
```

NDARC OpenMDAO Wrappers

Two versions of the OpenMDAO wrappers for NDARC are currently provided, one for OpenMDAO version 1.7+ and a preliminary one for OpenMDAO version 2+. The OpenMDAO 1.7+ wrapper is defined in the `rcotools.nadarc.ndarc_mdao1` module, while the OpenMDAO 2+ wrapper is defined in the `rcotools.nadarc.ndarc_mdao2` module. The OpenMDAO 2+ wrapper is still in the early stages of development, so only the OpenMDAO 1.7+ wrapper will be presented.

The `ndarc_mdao1` module defines the class `NdarcWrapper`, which is subclass of the OpenMDAO Component object and serves as a wrapper for NDARC. The `NdarcWrapper` initialization method reads in an NDARC namelist, makes user specified modifications and then sets up OpenMDAO inputs and outputs. The `NdarcWrapper.solve_nonlinear()` method modifies the NDARC job namelist based on OpenMDAO inputs, executes the NDARC job and then passes on selected outputs from the NDARC solution file to OpenMDAO. Most of the initialization parameters are stored as attributes in `NdarcWrapper.nw_` to avoid potential namespace conflicts with attributes or methods in future versions of the Component object.

The simplest initialization of an `NdarcWrapper` object would be as follows:

```
NdarcWrapper(jobfile, inputs, outputs)
```

The parameter `jobfile` is an NDARC job file path, `inputs` is a list of a OpenMDAO variable inputs and their mappings to NDARC input variables and `outputs` is a list of the OpenMDAO variable outputs and their mappings to NDARC solution variables. The parameters `inputs` and the `outputs` are lists of (*openmdao_variable_name*, *ndarc_access_string*) tuples or (*openmdao_variable_name*, *ndarc_access_string*, *meta_data_dict*) tuples.

The *ndarc_access_string* items in these tuples are specified as Fortran namelist variable names for values in the NDARC data structure. For data structures that have multiple instances, such as Rotor, they are referenced using an index in prerentheses. For example, for two rotors you would have a Rotor(1) and Rotor(2) data structures.

Attributes of the `NdarcWrapper` object can also be specified by *ndarc_access_string*. For these, the variable name is the attribute name (from `NdarcWrapper.nw_`) prepended by "`%`". So to get the `NdarcWrapper.nw_.success` flag, the *ndarc_access_string* would be "`%success`". Another example would be to use the *ndarc_access_string* would be "`%converged`" to get the state of the `NdarcWrapper.nw_.converged` flag.

The optional *meta_data_dict* items are the optional parameters passed to a call to `NdarcWrapper.add_param(`*openmdao_variable_name, val=<object object>, **kwargs*`)` for inputs or to `NdarcWrapper.add_output(`*openmdao_variable_name, val=<object object>, **kwargs*`)` for outputs, where *key* is the *ndarc_access_string*. If the *meta_data_dict* is not provided or if the *meta_data_dict* does not include the *key* "*val*" then `meta_data_dict['val'] = 0.0` is assumed. Note that while OpenMDAO variables can be mapped to more than one NDARC input value, there has to be a one-to-one mapping of OpenMDAO variables to NDARC output (or solution) variables.

An example showing how the `NdarcWrapper` can be used for constrained optimization using OpenMDAO 1.7 is presented in Listing 5. This example is based on an example provided in the RCOTOOLS distribution that models a 90-passenger, tandem, compound helicopter. The inputs to NDARC are wing loading and disk loading, and the outputs from NDARC are empty weight, mission fuel, engine power, rotor aspect ratio and rotor radius. The outputs for empty weight, mission fuel and engine power are passed to a utility function, `CostFunction`, that produces a single weighted value as a cost function. The values for rotor aspect ratio and radius are used as constraints on the solution. A line-by-line description of this example is provided in the RCOTOOLS documentation.

Listing 5: Example using NdarcWrapper under OpenMDAO 1.7 for constrained optimization.

```
from openmdao.api import IndepVarComp, Problem, Group
from openmdao.api import ScipyOptimizer
from rcotools.ndarc.ndarc_mdao1 import NdarcWrapper
from rcotools.utils.mdao1 import CostFunction

# Setup NdarcJob component
jobfile = '/path/to/tc90 size init.job'
inputs = [
          ('wingload', 'Wing(1)%wingload'),
          ('diskload', 'Rotor(1)%diskload'),
          ('diskload', 'Rotor(2)%diskload'),  # 'diskload' sets two NDARC inputs
         ]
outputs = [
           ('weightempty', 'Aircraft%Weight%WE'),
           ('miss_fuel', 'Size%SizeMission(1)%MissParam%WFUEL_MISS(1)'),
           ('engine_power', 'EngineGroup(1)%PENG'),
           ('aspect_ratio', 'Rotor(1)%AspectRatio'),
           ('radius', 'Rotor(1)%Radius'),
          ]
NdarcJob = NdarcWrapper(jobfile, inputs, outputs)

# Setup cost function component (name, weight) for each factor
factors = [('weightempty',   1.0),
           ('miss_fuel',     5.0),
           ('engine_power', 12.5),
           ]
cost = CostFunction(factors)

# Setup the MDAO Problem
top = Problem()
root = top.root = Group()

# Add components to the OpenMDAO problem
root.add('p1', IndepVarComp('wingload', 100.0))
root.add('p2', IndepVarComp('diskload', 11.0))
root.add('ndarc', NdarcJob)
root.add('ndarc_cost', cost)

# Make connections between components
root.connect('p1.wingload', 'ndarc.wingload')
root.connect('p2.diskload', 'ndarc.diskload')
root.connect('ndarc.weightempty', 'ndarc_cost.weightempty')
root.connect('ndarc.miss_fuel', 'ndarc_cost.miss_fuel')
root.connect('ndarc.engine_power', 'ndarc_cost.engine_power')

# Set up optimizer SLSQP or COBYLA
top.driver = ScipyOptimizer()
top.driver.options['optimizer'] = 'SLSQP'

# Set the design variables
top.driver.add_desvar('p1.wingload', lower=70.0, upper=120.0)
top.driver.add_desvar('p2.diskload', lower=6.0, upper=15.0)

# Set the objective to be optimized
top.driver.add_objective('ndarc_cost.cost')

# Set the design constraint variables
top.driver.add_constraint('ndarc.radius', upper=37.0)
top.driver.add_constraint('ndarc.aspect_ratio', upper=20.0)

# Setup and run the problem
top.setup()
top.run()
```

Monitoring NDARC Run Convergence

NDARC can potentially run successfully, but not return a converged solution. (RCOTOOLS considers an NDARC run to be successful if it doesn't generate an error and there are no "NaN" values in the solution.) The `NdarcWrapper` class has a flag, `NdarcWrapper.nw_.converged`, that determines convergence from NDARC output files for each run. The results from a job that did not converge may still be reasonable, but one should be aware about how they might affect results from an optimization. Figure 5 shows the optimization steps for the example optimization presented in Listing 5. In this figure, cost is a function of the design variables *wingload* and *diskload.* The background contour is based on a 20x20 grid of solutions, with circles plotted at

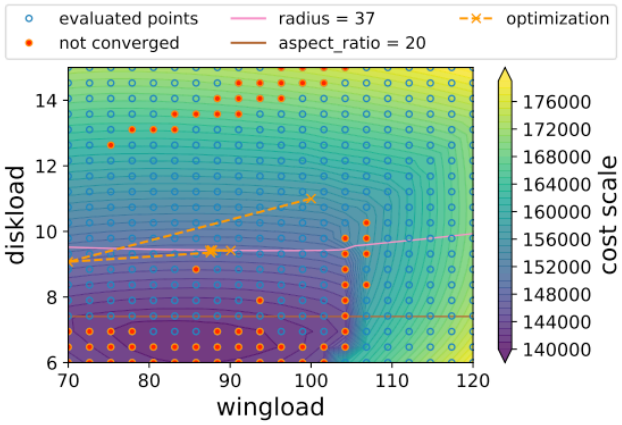

Figure 5: Optimization results provided for an NDARC job running under OpenMDAO superimposed on a contour plot of solution results over the design space. Filled red circles represent the wingload and diskload inputs resulting in non-converged solutions.

the *wingload* and *diskload* values used for each solution. The NDARC solutions that did not converge are represented by circles filled in red. The contour lines near solutions that did not converge are relatively smooth, indicating that their results are consistent with nearby solutions that did converge. Figure 5 implies that gradient-based optimization runs with evaluations that are only occasional near non-converged evaluations may still provide good results.

The orange dashed line shows the steps in an optimization that is constrained by having a solution with a radius of less than 37 feet. The solution space traversed during optimization is relatively free of non-converged solutions so the result should be good; the solution was confirmed by repeating the optimization with different starting conditions. On the other hand, an unconstrained optimization (see ndarc_tc90_unconstr.py in the RCOTOOLS distribution) probably would be suspect as the minimum cost function region contains numerous non-converged solutions. Repeated optimization runs using different starting conditions resulted in the number of evaluations required varying dramatically with the final results having significant variation. Often, convergence will be successful if some iteration input parameters changes, so such changes should be considered if convergence may be affecting optimization results.

CAMRAD Wrappers

CAMRAD II is a comprehensive helicopter analysis tool developed by Johnson Analytics.[2,3] CAMRAD II performs aeromechanical analyses for complex rotorcraft systems using a combination of advanced technology models, multibody dynamics, nonlinear finite elements, structural dynamics, and rotorcraft aerodynamics. CAMRAD II calculates performance, loads, vibration, response, and stability for a wide variety of rotorcraft over for prescribed operating conditions.

The CAMRAD II application wrappers provided in RCOTOOLS are intended to be "light-weight," with minimal computational overhead. CAMRAD II is written in the Fortran programming language and compiles to three executables, CAMRADII, INPUT and OUTPUT, which are listed in Table 2. The paths to these executables are saved in the `rcotools.CAMRAD` dictionary using the keys `camradii`, `input_camrad` and `output_camrad`, respectively. The CAMRADII executable performs the rotorcraft analyses, while the INPUT executable is used to produce binary files representing inputs and tables for use by CAMRADII. The OUTPUT executable is used to examine CAMRAD II plot file outputs. The OUTPUT executable is not currently used or supported by RCOTOOLS.

Table 2: CAMRAD II Executables

Name	Description	Path Key
CAMRADII	Rotorcraft Analysis	*camradii*
INPUT	Input and Table File Preparation	*input_camrad*
OUTPUT	Plot File Examination	*output_camrad*

Use of the RCOTOOLS CAMRAD wrappers assumes a working knowledge of CAMRAD II, as viable CAMRAD II inputs are required as a starting point. CAMRAD II input typically consists of separate files for tables, basic shell input and job input.[‡] The INPUT executable is used to generate binary files for tables and basic shell input that are to be read in by the CAMRADII executable based on commands given in the job input file.

Similar to the RCOTOOLS NDARC wrappers, the RCOTOOLS CAMRAD II wrappers consist of "core" wrappers which are generic Python wrappers to read, modify and execute program input files and to read and extract data from solution files. These do not rely on or use OpenMDAO software. A specific, OpenMDAO wrapper is also provided, which inherits from OpenMDAO objects. This wrapper is built on top of the "core" wrappers, but is designed to interface with the OpenMDAO optimization

[‡] CAMRAD II utilizes "shell" and "core" inputs, where "shell" inputs simplify the definition of typical system models and "core" inputs allow for more detailed and flexible system model definitions.

architecture. The OpenMDAO wrapper was designed to be flexible and to work "out-of-the-box" for most needs.

RCOTOOLS provides two command-line functions that read and display CAMRAD II input and output files in a Graphical User Interface (GUI) to help users identify references to CAMRAD II input and output variables which are used in the CAMRAD wrappers. These are the CAMRAD Input Viewer (`ciview`), which is instantiated using the `ciview` command from a terminal or console, and the CAMRAD Solution Viewer (`csview`), which is instantiated using the `csview` command. Their use is similar to the NDARC Input Viewer, that was described in the Utilities section.

CAMRAD Core Wrappers

The CAMRAD core application wrapper is defined in the `camrad_core` module, which defines several functions and classes. The two principal classes providing the core wrapper interface are `CamradJob` and `CamradSolution`.

The `CamradJob` class is usually not instantiated directly, instead a CAMRAD II job file is read and processed by the `parseCamradInput()` function. This function takes a CAMRAD II job file as an argument and parses the contents to return either a `CamradDict` or `CamradJob` object. A `CamradJob` is only returned if the input file contains a NLJOB namelist group. Once a `CamradJob` is created, modifications can be made and then the object can be executed to produce CAMRAD II output files. Both `CamradDict` and `CamradJob` are nested dictionary type objects that represent the namelist input in the source files used to initiate them. The dictionary keys for `CamradDict` and `CamradJob` objects are based on the NLDEF namelist groups used to define them. There are four optional variables used in NLDEF groups, *CLASS*, *TYPE*, *NAME* and *ACTION*. The key will be of the form "*CLASS*:*TYPE*:*NAME*:*ACTION*" which is populated by the values assigned to each variable. If any of these variables are undefined, then they are represented by an empty string. Finally, any ":" characters at the end of the key are stripped. For example:

- The namelist input "`&NLDEF class='ROTOR',type='FLEXBEAM', name='ROTOR 1',&END`" would be represented by the key "`ROTOR:FLEXBEAM:ROTOR 1`".
- The namelist input "`&NLDEF class='FLUTTER ROTOR',name='ROTOR 1',&END`" would be represented by the key "`FLUTTER ROTOR::ROTOR 1`". (Note the two consecutive colons in the middle of the key.)
- The namelist input "`&NLDEF class='FLUTTER', &END`" would be represented by the key "`FLUTTER`".

`CamradJob` objects have a ***`jobtype`*** attribute and an ***`exedict`*** attribute. The ***`jobtype`*** attribute should be either `camradii` or `camradii_input` to specify which CAMRAD II executable to use. This attribute is set automatically by the `parseCamradInput()` function if the input is a csh file for a CAMRAD job. The attribute can also be set directly or overridden when using the execute() method. The ***`exedict`*** attribute can also be set directly or overridden when using the `execute()` method. ***`exedict`*** defaults to the `CAMRAD` dictionary specified in the rcotools.ini file. The simplest use of `parseCamradInput()` and `CamradJob` would be as follows:

```
import rcotools.camrad.camrad_core as cc

# parse CAMRAD II job file and
# produce the CamradJob object, 'job'
job = cc.parseCamradInput(infpath)

# *** make namelist and job mods ***

# execute the CAMRAD II job
exitonerror = job.execute()
```

`CamradSolution` objects are nested, dictionary-type objects that represent parsed values from CAMRAD II analysis output files. The dictionary keys are based on section titles and variable names in the analysis output files. These outputs are formatted to be easily read and understood by human readers, not computer algorithms, so section titles and variable names are very descriptive. This results in dictionary keys that can be quite verbose. A simple example of how to use `CamradSolution` is shown in Listing 6.

Not all sections of CAMRAD solution output files have parsers defined and the existing parsers are not all thoroughly vetted. (Fully vetted means that each variable output in a section has been verified to be included in parsed data structure.) Users are encouraged to submit new parser requests and submit bug reports if parser errors are found. Table 3 provides a list of CAMRAD II solution output sections and the current status of parsers for those sections.

Listing 6: Use of CamradSolution to access output data.

```
import rcotools.camrad.camrad_core as cc

# Parse the CAMRAD II solution solution file, 'sample.out'
soln = cc.CamradSolution('sample.out')

# Access and print the radius of Rotor 1
radius = soln.getByAccesstext('CASE 1%SUMMARY%CONFIGURATION%ROTOR 1%RADIUS (FT)')
print('\nThe radius of Rotor 1 in the output summary is: %f (FT)' % radius)
```

Table 3: List of CAMRAD II output sections with current parser status.

CAMRAD II Solution Section	Parser Status
JOB DESCRIPTION	Generic Parser - Not Vetted
CASE *n*	
LABELS	Custom Parser - Vetted
ANALYSIS TASKS	Custom Parser – Vetted
CONFIGURATION	Custom Parser – Vetted
OPERATING CONDITION	Custom Parser – Vetted
CONTROL SETTINGS	Custom Parser – Vetted
TRIM SOLUTION PROCEDURE	No Parser
FLUTTER EQUATIONS	No Parser
FLUTTER SOLUTION PROCEDURE	No Parser
SHELL INPUT DATA	
NLDEF/NLVAL values	Generic Parser - Not Vetted
CORE INPUT DATA	No Parser
TABLE INPUT DATA	No Parser
SYSTEM AND FUNCTIONALITY	No Parser
TRIM CONVERGENCE	No Parser
TRIM SOLUTION	
ROTORCRAFT	Custom Parser - Partially Vetted
ROTOR *n*	Custom Parser - Partially Vetted
ROTOR BLADE LOAD SENSOR	Custom Parser - Partially Vetted
AIRFRAME SENSOR ROTOR	Custom Parser - Partially Vetted
ROTOR HUB LOAD SENSOR	Custom Parser - Partially Vetted
ROTOR BLADE POSITION SENSO	Custom Parser - Partially Vetted
ROTOR WING SENSOR	Custom Parser - Partially Vetted
ROTOR CONTROL LOAD SENSOR	Custom Parser - Partially Vetted
ROTOR INTVEL TRANSFORM	Custom Parser - Partially Vetted
ANALYSIS FRAMES	No Parser
FLUTTER SOLUTION	No Parser
ANALYSIS OF SYSTEM OF CONSTANT...	No Parser

CAMRAD OpenMDAO Wrapper

The CAMRAD II OpenMDAO 1.7 application wrapper is defined in the `camrad_mdao1` module. This module currently implements the class `CamradWrapper`, which is subclass of the OpenMDAO `Component` object, that serves as a wrapper for CAMRAD. The `CamradWrapper` initialization reads in a CAMRAD job file, makes user specified modifications and then sets up OpenMDAO inputs and outputs.

The `CamradWrapper.solve_nonlinear()` method modifies the CAMRAD job file based on OpenMDAO inputs, executes the CAMRAD job and then passes on selected outputs from the CAMRAD solution file to OpenMDAO. Most of the initialization parameters are stored as attributes in `self.cw_` to avoid potential namespace conflicts with attributes or methods in future versions of the OpenMDAO Component object.

The essential parameters needed to initialize a `CamradWrapper` object are a CAMRAD II job file path, a definition of the OpenMDAO variable inputs and their mappings to CAMRAD II input variables and a definition of the OpenMDAO variable outputs and their mappings to CAMRAD II solution variables. Both the inputs and the outputs are specified as a list of (*openmdao_variable_name*, *camrad_access_string*) tuples or (*openmdao_variable_name, camrad_access_string*, *meta_data_dict*) tuples.

The *camrad_access_string* items are specified as Fortran namelist variable names for values in the CAMRAD input or output data structures. These can be determined by using the `ciview` and `csview` GUI utilities. Attributes of the `CamradWrapper` object can also be specified by *camrad_access_string*. For these, the variable name is the attribute name (from `CamradWrapper.cw_`) prepended by "`%`". So to get the `CamradWrapper.cw_.success flag`, the *camrad_access_string* would be "`%success`".

The optional *meta_data_dict* items are parameters passed to a call to `CamradWrapper.add.param(`*openmdao_variable_name, val=<object object>, **kwargs*) for inputs or to `CamradWrapper.add_output`(*openmdao_variable_name, val=<object object>, **kwargs*) for outputs, where the key is the *camrad_access_string*. If the *meta_data_dict* is not provided or if the *meta_data_dict* does not include the key '`val`', then `meta_data_dict['val'] = 0.0`. OpenMDAO variables can be mapped to more than one CAMRAD input value, but there has to be a one-to-one mapping of OpenMDAO variables to CAMRAD output (or solution) variables. An example showing the specification of inputs and outputs for creating a wrapper around an CAMRAD II analysis job is given Listing 7.

A simple example for using the CAMRAD OpenMDAO 1.7 wrapper is provided by the script in Listing 8. The `CamradWrapper` class has numerous customization options that are set during initialization, which are detailed in the source code documentation.

Listing 7: Use of CamradWrapper.

```
from rcotools.camrad.camrad_mdao1 import CamradWrapper

inputs = [
          ('radius', 'CASE 1%SHELL%ROTOR:STRUCTURE:ROTOR 1%RADIUS', {'val': 20.0}),
          ('twistl', 'CASE 1%SHELL%ROTOR:STRUCTURE:ROTOR 1%TWISTL', {'val': -10.0}),
         ]
outputs = [
           ('lift2drag', 'CASE 1%TRIM SOLUTION:ROTOR 1 PERFORMANCE%PERFORMANCE%L/D'),
           ('Mx', 'CASE 1%TRIM SOLUTION:ROTOR 1 PERFORMANCE%ROTOR FORCES AND MOMENTS%SHAFT AXES%ROLL MOMENT%MX'),
           ('My', 'CASE 1%TRIM SOLUTION:ROTOR 1 PERFORMANCE%ROTOR FORCES AND MOMENTS%SHAFT AXES%PITCH MOMENT%MY'),
           ('lift', 'CASE 1%TRIM SOLUTION:ROTOR 1 PERFORMANCE%ROTOR FORCES AND MOMENTS%WIND AXES%LIFT%L'),
           ('eqdrag', 'CASE 1%TRIM SOLUTION:ROTOR 1 PERFORMANCE%PERFORMANCE%ROTOR EQUIV DRAG%D=P/V+X'),
          ]
```

Listing 8: Example use of CamradWrapper with OpenMDAO 1.7.

```
from openmdao.api import IndepVarComp, Problem, Group
from rcotools.camrad.camrad_mdao1 import CamradWrapper

# Setup the CamradJob inputs and outputs
inputs = [
          ('radius', 'CASE 1%SHELL%ROTOR:STRUCTURE:ROTOR 1%RADIUS', {'val': 20.0}),
          ('twistl', 'CASE 1%SHELL%ROTOR:STRUCTURE:ROTOR 1%TWISTL', {'val': -10.0}),
         ]
outputs = [
           ('lift2drag', 'CASE 1%TRIM SOLUTION:ROTOR 1 PERFORMANCE%PERFORMANCE%L/D'),
           ('Mx', 'CASE 1%TRIM SOLUTION:ROTOR 1 PERFORMANCE%ROTOR FORCES AND MOMENTS%SHAFT AXES%ROLL MOMENT%MX'),
           ('My', 'CASE 1%TRIM SOLUTION:ROTOR 1 PERFORMANCE%ROTOR FORCES AND MOMENTS%SHAFT AXES%PITCH MOMENT%MY'),
           ('lift', 'CASE 1%TRIM SOLUTION:ROTOR 1 PERFORMANCE%ROTOR FORCES AND MOMENTS%WIND AXES%LIFT%L'),
           ('eqdrag', 'CASE 1%TRIM SOLUTION:ROTOR 1 PERFORMANCE%PERFORMANCE%ROTOR EQUIV DRAG%D=P/V+X'),
          ]

# Create NdarcWrapper component
cjob = CamradWrapper("/path/to/job/file", inputs, outputs)

# Set up MDAO Problem
top = Problem()
root = top.root = Group()

# add components
root.add('camrad', cjob)
root.add('p1', IndepVarComp('radius', 20.0))
root.add('p2', IndepVarComp('twistl', -10.0))

# add connections
root.connect('p1.radius', 'camrad.radius')
root.connect('p2.twistl', 'camrad.twistl')
top.setup()

# Run the problem and print inputs & outputs
top['p1.radius'] = 21.0  # Set a new radius
top.run()
print('Inputs:')
print('    Rotor Radius = %0.3f (ft)' % top['p1.radius'])
print('    Rotor Linear Twist = %0.3f (deg)' % top['p2.twistl'])
print('Outputs:')
print('    L/D = %.4f' % top['camrad.lift2drag'])
print('    Roll Moment = %.4f (ft-lb)' % top['camrad.Mx'])
```

Concluding Remarks

RCOTOOLS has already proven to be a useful tool for including NDARC and CAMRAD II in multi-disciplinary design and analysis of rotorcraft. Further improvements and additions are expected as development continues. Currently, one or two RCOTOOLS updates are distributed to users each month. Both the NDARC and the CAMRAD II wrappers are actively being used, with users often submitting bug reports and/or feature requests. A wrapper for the Numerical Propulsion System Simulation (NPSS)[9] is under development and wrappers for other design tools may also be considered for future additions to RCOTOOLS. The source code is extensively documented and a user's guide is provided with the distribution package. This documentation is available in both HTML and PDF formats. (The PDF version is currently over 100 pages in length.) RCOTOOLS runs under both Python 2.7 and 3 and is tested on the Windows 10, MacOS and Linux operating systems. RCOTOOLS distribution is currently limited to users in the US Government or working on US Government contracts. In the future, RCOTOOLS is expected to be made available to more users, possibly as part of the NDARC software distribution.

References

[1] Johnson, W., *NDARC - NASA Design and Analysis of Rotorcraft*, NASA TP-2015-218751, April 2015.

[2] Johnson, W., "Technology Drivers in the Development of CAMRAD II," *American Helicopter Society Aeromechanics Specialists Conference*, San Francisco, CA: 1994.

[3] Johnson, W., "Rotorcraft Aeromechanics Applications of a Comprehensive Analysis," *AHS International Meeting on Advanced Rotorcraft Technology and Disaster Relief*, Gifu, Japan: 1998.

[4] Johnson, W., Moodie, A. M., and Yeo, H., "Design and Performance of Lift-Offset Rotorcraft for Short-Haul Missions," *American Helicopter Society Future Vertical Lift Aircraft Design Conference*, San Francisco, CA: 2012.

[5] Heath, C. M., and Gray, J. S., "OpenMDAO: Framework for Flexible Multidisciplinary Design, Analysis and Optimization Methods," *AIAA Journal*, vol. 51, 2013, pp. 2380–2394.

[6] Avera, M., and Singh, R., "OpenMDAO/NDARC Framework for Assessing Performance Impact of Rotor

Technology Integration," *American Helicopter Society 70th Annual Forum*, Montréal, Québec, Canada: 2014.

[7] Sinsay, J. D., and Alonso, J. J., "Optimization of a Lift-Offset Compound Helicopter in a Multidisciplinary Analysis Environment," *American Helicopter Society 71st Annual Forum*, Virginia Beach, VA: 2015.

[8] Sinsay, J. D., Hadka, D. M., and Lego, S. E., "An Integrated Design Environment for NDARC," *AHS Technical Meeting on Aeromechanics Design for Vertical Lift*, San Francisco, CA: 2016.

[9] Lytle, J. K., *The Numerical Propulsion System Simulation: An Overview*, NASA TM-2000-209915, June 2000.