NRAP-Open-IAM Developer’s Guide

Release alpha 2.7.1-23.06.30

V. Vasylkivska, S. King, D. Harp, D. Bacon

Jun 30, 2023

CONTENTS:

CHAPTER ONE

INTRODUCTION

This document describes the software design and functional requirements of the Phase II Integrated Assessment Model (IAM), referred to as the NRAP-Open-IAM in the current document. This design process adapted a use-case-driven approach [6] to design a general purpose open source version of the IAM, built upon the development effort of Phase I NRAP-IAM-CS [4], [7]. Based on this general-purpose design, a core-functionality prototype version of the NRAP-Open-IAM was developed and tested.

In NRAP Phase II researchers are developing an integrated assessment model that will incorporate workflows for containment assurance, monitoring design, post-injection site care, assessment of model concordance to measured field data, evaluation of the performance of mitigation alternatives, and updating of probabilistic assessments as new data become available. The design goals are to create a flexible framework for the integrate assessment model that will be easily maintainable, flexible, and extendable.

The document is intended to outline the NRAP-Open-IAM design structure and functionality and give guidance to the developers of reduced order models (ROMs), monitoring modules, utility applications and algorithm developers for interfacing and integrating their models into the NRAP-Open-IAM. The scope of the functionalities of the NRAP-Open-IAM include risk quantification, risk management and support of iterative risk assessment processes. The system components include the geophysical and geochemical model components for risk quantification, analysis components for risk management and parameter passing workflows, procedures and mechanisms allowing the communications between the different system components to support iterative risk assessment processes.

CHAPTER TWO

OBTAINING NRAP-OPEN-IAM

2.1 Introduction

NRAP-Open-IAM is an open-source Integrated Assessment Model (IAM) for Phases II and III of the National Risk Assessment Partnership (NRAP). The goal of this software is to go beyond risk assessment into risk management and containment assurance. NRAP-Open-IAM is currently in active development and is available for testing and feedback only.

As this is a prototype of software being actively developed, we are seeking any feedback and/or issue reports. An online feedback form can be found here: https://docs.google.com/forms/d/e/1FAIpQLSed5mcX0OBx1dLNmYGbmS4Vfc0mdOLapIzFqw-6vHoho9B19A/viewform?usp=sf_link

Issue reports and feedback can be left at the forum on NETL's Energy Data eXchange webpage for NRAP-Open-IAM: https://edx.netl.doe.gov/workspace/forum/nrap-tools/topic?t=nrap-tools-nrap-open-iam or on GitLab issues page for NRAP-Open-IAM: https://gitlab.com/NRAP/OpenIAM/-/issues?sort=created_date&state=opened

If you have been given access to the code indirectly and would like to be notified when updates are available for testing, please contact the development team to be added to our email list.

2.2 Downloading NRAP-Open-IAM

NRAP-Open-IAM tool and examples can be downloaded from a public GitLab repository located at https://gitlab.com/NRAP/OpenIAM. If the NRAP-Open-IAM was downloaded from the GitLab repository, the folder name may have the repository's current hash appended to it. Feel free to rename the folder something simple like *NRAPOpenIAM* to simplify the navigation.

2.3 Installing NRAP-Open-IAM

The NRAP-Open-IAM requires Python version 3.9 or greater to operate. If you need to install Python, we describe all steps of the installation process below.

General Installation Guide:

- Extract the tool files from the provided/downloaded zip.
- Navigate to the *installers* folder within the recently unzipped directory.
- Navigate to the folder corresponding to the operating system that you are utilizing.
- Follow instructions file located in the folder for your operating system.

For Windows: The file *Installation_Instructions_Windows.txt* describes steps required to install needed Python packages for the proper work of NRAP-Open-IAM.

For macOS: The file *Installation_Instructions_macOS.txt* describes steps user needs to follow in order to install required Python packages.

For Linux OS: Linux users are assumed to know the installation commands for their specific version of Linux needed to install required tools. The file *Installation_Instructions_Linux.txt* specifies the needed software and package dependencies.

For alternative installation of Python the following packages are needed: `NumPy`, `SciPy`, `PyYAML`, `Matplotlib`, `Pandas`, `TensorFlow` (of version 2.6), `Keras`, `scikit-learn`, `Pmw`, `pip`, and `six`. In most cases (mainly dependent on the platform and Python distribition) the required libraries can be installed using `pip` or `conda` package managers. Additional libraries recommended to run Jupyter notebooks and scripts illustrating work of NRAP-Open-IAM are `IPython` and `Jupyter`.

On macOS and Linux machines the gfortran compiler needs to be present/installed to compile some of the NRAP-Open-IAM code (macOS users can find gfortran here: (https://gcc.gnu.org/wiki/GFortranBinariesMacOS)).

After the proper version of Python is installed, the NRAP-Open-IAM can be set up and tested. **Note: If Python was installed through Anaconda please use Anaconda prompt instead of command prompt for setup and tests.** In the NRAP-Open-IAM distribution folder find and open the sub-folder *setup*. Next, open a command prompt/Anaconda prompt in the *setup* folder (on Windows, this can be done by holding Shift and right clicking inside the folder when no file is selected, then selecting `Open command window here`; alternatively, one can navigate to the folder *setup*, type `cmd` in the address bar of the file browser and hit Enter to open the command prompt there). (On Windows, Anaconda prompt can be found in the programs menu under submenu `Anaconda3 (64-bit)`.) Run the setup script by entering the command:

```
python openiam_setup_tests.py
```

in the command prompt/Anaconda prompt. This will test the version of Python installed on the system. Next the setup script will test the versions of several Python libraries that the NRAP-Open-IAM depends on. The setup script will compile several Fortran libraries needed for some component models on Mac and Linux. Users of Windows OS will be provided with the compiled libraries. Finally, the setup script will run the test suite to see if the NRAP-Open-IAM has been installed correctly. If the results printed to the console indicate errors during the testing the errors have to be resolved before the NRAP-Open-IAM can be used. When contacting the developers to resolve problems please include all output from the setup script or test suite runs.

2.4 Testing installation

After setup the test suite can be run again by entering the NRAP-Open-IAM *test* directory in a terminal and typing:

```
python iam_test.py
```

Test results will be printed to the terminal. The setup script run during the installation process uses the same test suite after testing whether the necessary Python libraries are installed, and compiling the NRAP-Open-IAM libraries.

2.5 Contributors

During the Phase II and/or Phase III of the NRAP the following researchers contributed to the development of NRAP-Open-IAM (listed in alphabetical order with affiliation at the time of active contribution):

- Diana Bacon (Pacific Northwest National Laboratory)
- Seunghwan Baek (Pacific Northwest National Laboratory)
- Pramod Bhuvankar (Lawrence Berkeley National Laboratory)
- Suzanne (Michelle) Bourret (Los Alamos National Laboratory)
- Julia De Toledo Camargo (Pacific Northwest National Laboratory)
- Bailian Chen (Los Alamos National Laboratory)
- Abdullah Cihan (Lawrence Berkeley National Laboratory)
- Dylan Harp (Los Alamos National Laboratory)
- Paul Holcomb (National Energy Technology Laboratory)
- Jaisree Iyer (Lawrence Livermore National Laboratory)
- Elizabeth Keating (Los Alamos National Laboratory)
- Seth King (National Energy Technology Laboratory)
- Greg Lackey (National Energy Technology Laboratory)
- Ernest Lindner (National Energy Technology Laboratory)
- Kayyum Mansoor (Lawrence Livermore National Laboratory)
- Mohamed Mehana (Los Alamos National Laboratory)
- Saro Meguerdijian (Los Alamos National Laboratory)
- Nathaniel Mitchell (National Energy Technology Laboratory)
- Omotayo Omosebi (Lawrence Berkeley National Laboratory)
- Veronika Vasylkivska (National Energy Technology Laboratory)
- Ya-Mei Yang (National Energy Technology Laboratory)
- Yingqi Zhang (Lawrence Berkeley National Laboratory)

CHAPTER THREE

CODING LOGISTICS

3.1 Component model

The NRAP-Open-IAM is built on the open-source concept to promote transparency and to give advanced users the capability to contribute to the code. The main framework for the NRAP-Open-IAM is written in Python 3. Python provides the cross-platform capabilities, an extensive number of libraries for data handling, analysis, and visualization, the flexibility to interact with other languages that component models may use (Fortran, C++, etc.). No coding style is enforced but conforming to PEP 8 is recommended. The exceptions are that the line lengths should not exceed 120 characters, and spaces (no tabs) must be used for indentations.

Underpinning the NRAP-Open-IAM is the MATK Python package (Model Analysis ToolKit) [1] which provides a framework for the system model components. The NRAP-Open-IAM is designed to be flexible with regard to how component models are added so only the desired components of the system model need to be included for a specific use case.

We start with describing the two main terms developer has to know for the successful integration of a component model into the NRAP-Open-IAM framework. The first term is *simulation model* (sometimes referred to as a *component model*) that stands behind every building block (component) of the system. For example, a reservoir and a collection of wellbores can constitute one system within the NRAP-Open-IAM. The behavior of the reservoir and wellbores within the system is described by simulation models developed specifically for a given component (e.g., reservoir or wellbore). Thus, simulation model is a function which when provided with the component input parameters calculates the corresponding output (describes the component behavior). The component code must be licensable as open source. This guide is intended to instruct simulation code developers on the integration of their simulation models into the NRAP-Open-IAM framework regardless of the language used to develop the simulation model.

The second term is *component model* (`ComponentModel`) *class*. The NRAP-Open-IAM code relies on using several classes providing the framework for all use cases. Among these classes are `SystemModel` and `ComponentModel` classes. Every component (class) currently available within the NRAP-Open-IAM distribution is written as a class which inherits its main features, attributes and methods from `ComponentModel` class.

The `ComponentModel` class is written to help integrate different kinds of simulation models into the NRAP-Open-IAM framework. It serves as a wrapper in the case when the simulation model is developed in a language different from Python. It is used as an interface between the NRAP-Open-IAM and simulation models already written in Python. Two instance methods of the `ComponentModel` class that we consider first are `__init__` and `simulation_model`. These two methods are examples of methods that are usually redefined within a given component.

```
class ComponentModel(object):
    """
    NRAP-Open-IAM ComponentModel class.

    Class represents basic framework to construct component models, handle different
    types of parameters and observations.
```

(continues on next page)

(continued from previous page)

```
    """
    def __init__(self, name, parent, model='',
                 model_args=[], model_kwargs={}, workdir=None):
        """
        Constructor method of ComponentModel class.

        :param name: name of component model
        :type name: str

        :param parent: the SystemModel object that the component model belongs to
        :type parent: SystemModel object

        :param model: Python function whose first argument is a dictionary
            of parameters. The function returns dictionary of model outputs.
        :type model: function or method that is called when the component is run

        :param model_args: additional optional parameters of the component
            model; by default, model_args is empty list []
        :type model_args: [float]

        :param model_kwargs: additional optional keyword arguments of
            the component model; by default, model_kwargs is empty dictionary {}.
        :type model_kwargs: dict

        :param workdir: name of directory to use for model runs (serial run case)
        :type workdir: str

        :returns: object -- ComponentModel object
        """
        self._parent = parent
        self.name = name
        self.model = model
        self.model_args = model_args
        self.model_kwargs = model_kwargs
        self.workdir = workdir

        # Parameters
        self.default_pars = OrderedDict()
        self.deterministic_pars = OrderedDict()
        self.pars = OrderedDict()
        self.composite_pars = OrderedDict()
        self.gridded_pars = OrderedDict()
        self.parlinked_pars = OrderedDict()
        self.obslinked_pars = OrderedDict()

        # Keyword arguments
        self.obs_linked_kwargs = OrderedDict()
        self.grid_obs_linked_kwargs = OrderedDict()
        self.dynamic_kwargs = OrderedDict()
        self.collection_linked_kwargs = OrderedDict()
```

(continues on next page)

(continued from previous page)

```
        # Observations
        self.linkobs = OrderedDict()
        self.accumulators = OrderedDict()
        self.obs = OrderedDict()
        self.grid_obs = OrderedDict()
        self.local_obs = OrderedDict()

        # Set the working directory index for parallel runs
        self.workdir_index = 0

        # Setup how often the model method should be run
        self.run_frequency = 2

    # Other methods of the class follow the __init__ method
```

The `__init__` method of `ComponentModel` class defines possible attributes of the `ComponentModel` class object. We will discuss them later in the guide. The `model` attribute of the class object is defined by the *model* argument provided to the `__init__` method. A user's component model class derived from the `ComponentModel` class usually utilizes `__init__` method to define the setup of the component. It can be used to set the attributes of the component needed for the proper functioning of its simulation model: e.g., input parameters of the component and their bounds, properties of component observations, directory containing any extra files. We will use snapshots of the available NRAP-Open-IAM code to show the possible implementation of the `__init__` method.

3.2 Method `__init__`

In this section we consider an example of a typical `__init__` method in a component.

```
class Component1(ComponentModel):

    def __init__(self, name, parent, attr1, attr2, **kwargs):
        # Set up keyword arguments of the 'model' method provided by the system model
        model_kwargs = {'time_point': 365.25, 'time_step': 365.25}
        super().__init__(
            name, parent, model=self.simulation_model, model_kwargs=model_kwargs)

        # Set default parameters of the component model
        self.add_default_par('par1', value=3.0)
        self.add_default_par('par2', value=2.0)
        self.add_default_par('par3', value=-7.0)
        self.add_default_par('par4', value=0.0)

        # Define dictionary of parameters boundaries
        self.pars_bounds = dict()
        self.pars_bounds['par1'] = [1.0, 5.5]
        self.pars_bounds['par2'] = [0.0, 10.0]
        self.pars_bounds['par3'] = [-100.0, 100.0]
        self.pars_bounds['par4'] = [-100.0, 100.0]

        # Define dictionary of temporal data limits
        # Boundaries of temporal inputs are defined by the simulation model
```

(continues on next page)

(continued from previous page)

```
        self.temp_data_bounds = dict()
        self.temp_data_bounds['temp_input1'] = ['Temporal input 1', 1., 5.]
        self.temp_data_bounds['temp_input2'] = ['Temporal input 2', 1.5, 4.5]

        # Define accumulators and their initial values
        self.add_accumulator('accumulator1', sim=0.0)
        self.add_accumulator('accumulator2', sim=1.0)

        # Define additional component model attributes
        self.additional_attr1 = attr1
        self.additional_attr2 = attr2

        # Indicate how often the component should be run
        self.run_frequency = 2     # 2 is default

        # Define the conditional attribute if it is provided in kwargs
        if 'cond_attr' in kwargs:
            self.conditional_attr = kwargs['cond_attr']
```

Next we consider the code line by line. The first line initiates the `Component1` class and specifies that it is derived from the `ComponentModel` class (i.e., inherits all methods and attributes from the `ComponentModel` class). The second (non-blank) line shows that the constructor method `__init__` has several arguments: *name*, *parent*, *attr1*, *attr2*, and possibly some extra keyword arguments whose names are not provided.

```
model_kwargs = {'time_point': 365.25, 'time_step': 365.25}
super().__init__(
    name, parent, model=self.simulation_model, model_kwargs=model_kwargs)
```

The next line indicates that the simulation model of the component is a time-dependent function since component requires time point and time step as its keyword arguments. Method `super` calls the constructor method `__init__` of the base class `ComponentModel`. It provides a shortcut to include a base class's methods without having to know the base class type or name. The `super` method uses the *model_kwargs* variable as arguments of the `ComponentModel` class's `__init__` method. Analysis of the arguments of the `__init__` method shows that for this component the `model` attribute of the class object (`self.model`) is defined as a method since `model` argument of the `__init__` method has to provide a *simulation model* of the component, i.e. model (method/function) that transforms components input parameters into its output.

```
self.add_default_par('par1', value=3.0)
self.add_default_par('par2', value=2.0)
self.add_default_par('par3', value=-7.0)
self.add_default_par('par4', value=0.0)
```

The next four lines show that the `Component1` class object has four parameters with names *par1*, *par2*, *par3* and *par4*. The parameters are assigned default values of 3.0, 2.0, -7.0, and 0.0, respectively. The number of possible model parameters is not limited by the NRAP-Open-IAM framework. The purpose of this section of code in the `__init__` method is to ensure that all parameters of the simulation model corresponding to the component are defined even when not all parameters values are defined by users at runtime. The values provided here are arbitrary.

```
self.pars_bounds = dict()
self.pars_bounds['par1'] = [1.0, 5.5]
self.pars_bounds['par2'] = [0.0, 10.0]
self.pars_bounds['par3'] = [-100.0, 100.0]
self.pars_bounds['par4'] = [-100.0, 100.0]
```

The four lines above define the dictionary attribute pars_bounds containing the upper and lower boundaries for each parameter of the simulation model. This attribute is used in the method check_input_parameters of the ComponentModel class which can be redefined within the derived component class to overwrite the default parameter bounds checks. The method check_input_parameters is called before the start of each simulation to check whether the provided input parameters satisfy the defined boundaries. An implementation of the default check_input_parameters method is discussed later.

In many cases, the simulation model accepts time-varying inputs which have to (possibly) satisfy some model limitations. The check for these limitations should be implemented in the simulation_model method since these inputs change in time: this way the limits will be rechecked at each time step as opposed to only at instantiation of the component model object. Similar to input parameters, the number of possible temporal inputs of the simulation model is not limited by the NRAP-Open-IAM framework.

```
self.temp_data_bounds = dict()
self.temp_data_bounds['temp_input1'] = ['Temporal input 1', 1.0, 5.0]
self.temp_data_bounds['temp_input2'] = ['Temporal input 2', 1.5, 4.5]
```

Each entry of the dictionary temp_data_bounds is a list of three elements (right sides of the last two expressions above): the first is an explanatory name of the temporal input, the last two are the lower and upper boundaries of the input. An example of the possible implementation of the temporal inputs check method is discussed later in the guide.

```
self.add_accumulator('accumulator1', sim=0.0)
self.add_accumulator('accumulator2', sim=1.0)
```

The two lines above allow the creation of observations wihin the model (sometimes strictly internal to the component model) whose current value depends on observation values the component model produced at the previous time step(s). The accumulators are usually used in the simulation_model method for the storage of internally calculated values needed for the simulation_model method at the next time step (e.g., total mass calculated from flow rates at each time step).

```
self.additional_attr1 = attr1
self.additional_attr2 = attr2
```

In some cases the setup of the component requires the definition of additional attributes specific for the given component (e.g., attributes that address the setup of the component) that do not change during simulation and can be utilized in the component model method or by other components. The names additional_attr1 and additional_attr2 are arbitrary and can be defined to represent the intended purpose.

The following line serves as a possible example of the use of additional attributes.

```
self.run_frequency = 2
```

Attribute run_frequency is a flag variable that informs the system model of the frequency at which the component simulation model should be called. The possible values are 2, 1, and 0. A value of 2 means that the model is called for each time point supplied by the system, thus, the value of 2 is assigned, by default, to every ComponentClass instance and to all derived class instances. Consequently, the line demonstrated above (self.run_frequency = 2) is not required if this is an intended behavior. If the behavior (frequency of calling the simulation model) should be different, the line should be changed to

```
self.run_frequency = 1
```

or

```
self.run_frequency = 0
```

A value of 1 means that the simulation model should be run only for the first time point. A value of 0 means that the simulation model of a particular component should not be run at all. This is useful in the situations when the component does not have a `simulation_model` method, and serves as a container for parameters rather than a source of outputs, or when the `simulation_model` method should not be run after a particular time point. In the latter case some other system component should control when the model method of a given component should be turned off.

```
if 'cond_attr' in kwargs:
    self.conditional_attr = kwargs['cond_attr']
```

Some of the `Component1` object attributes may not be defined for all instances. In this case the attribute can be assigned a value, for example, only if a particular argument of the `__init__` method was provided. The conditional arguments can control and define some features of the simulation model defined by the developer. The names `conditional_attr` and *cond_attr* are arbitrary and for illustration purposes only. They can be defined to represent the intended purpose of the attribute.

3.3 Method `simulation_model`

The `simulation_model` method is an instance method of the component class that either calls the simulation model (typical if the simulation model is an external code not written in Python), or is the simulation model itself (typical if the simulation model is written in Python and can be implemented directly in the component class). To avoid confusion the method is named differently (self.simulation_model) from the instance attribute (self.model) containing reference to the method. The `simulation_model` method must accept a dictionary of input parameter values keyed by parameter names as the first argument and return a dictionary of model results keyed by distinct observation names. It is assumed that the arguments of the `simulation_model` method can be split into time-constant and time-varying arguments. All constant parameters defined by the user of the component model must be passed to the `simulation_model` method in the dictionary described above. The time-varying and other types of arguments (not defined by user) can be passed as keyword arguments. The header of the `simulation_model` method can include the default values of these arguments for situations when they are not provided (see first line below; e.g., `temp_input1=2.0`). Sample code of a `simulation_model` method is provided below. Note that this is just one of the possible implementations of the `simulation_model` method. The comments along with code sections provide suggestions of `simulation_model` method implementation. For additional examples please refer to the source code of existing NRAP-Open-IAM components, e.g., *source/openiam/multisegmented_wellbore_component.py*. The source code of the example component can be found in the folder *examples/scripts/iam_example_component.py*.

```
def simulation_model(self, p, temp_input1=2.0, temp_input2=3.0,
                     time_point=365.25, time_step=365.25):
    """
    :param p: input parameters of Component1 model
    :type p: dict

    :param temp_in1: the first of the two varying in time inputs of simulation_model
        method with default value of 2.0
    :type temp_in1: float

    :param temp_in2: the second of the two varying in time inputs of simulation_model
        method with default value of 2.0
    :type temp_in2: float

    :param time_point: time point in days at which the model output is
        to be calculated; by default, its value is 365.25 (1 year in days)
    :type time_point: float

```

(continues on next page)

(continued from previous page)

```
    :param time_step: difference between the current and previous
        time points in days; by default, its value is 365.25 (1 year in days)
    :type time_point: float

    """
    # Obtain the default values of the parameters from dictionary
    # of default parameters
    actual_p = dict([(k,v.value) for k,v in self.default_pars.items()])

    # Update default values of parameters with the provided ones
    actual_p.update(p)

    # For the initial time point 0.0 the model should be able to return
    # the initial values of the component observations
    if time_point == 0.0:
        # Define initial values of the model observations
        # The values chosen are arbitrary and the number of possible
        # observations of model is not NRAP-Open-IAM framework limited
        out = {}
        out['obs1'] = 0.0
        out['obs2'] = 0.0
        out['obs3'] = 0.0

        # Exit method
        return out

    # Check whether the temporal inputs satisfy the model requirements
    # and/or assumptions if needed
    # The instance method check_temporal_input can use the attribute
    # temp_data_bounds defined in the __init__ method
    assumptions_satisfied = self.check_temporal_input(time, temp_input1, temp_input2)

    # The next steps depend on a particular implementation of the simulation_model method
    if assumptions_satisfied:
        # Calculate output of the component using parameters and
        # temporal keyword arguments. The signature of the model_function
        # is not defined by the NRAP-Open-IAM framework
        output = model_function(p, temp_input1, temp_input2, time_point, time_step)

        # Assign values to the component accumulators
        # acc_fun1 and acc_fun2 are replacement names for some actions performed
        # on the variable output in order to obtain accumulators values
        self.accumulators['accumulator1'].sim = acc_fun1(output)
        self.accumulators['accumulator2'].sim = acc_fun2(output)

        # Assign model observations
        # f1, f2, f3 are function names replacing some actions performed
        # on the variable output in order to obtain observations values
        out['obs1'] = f1(output)
        out['obs2'] = f2(output)
        out['obs3'] = f3(output)
```

(continues on next page)

(continued from previous page)

```
        # The next type of statement is required: the model method should
        # return a dictionary with keys corresponding to the names of
        # all possible Component1 observations
        return out
```

Additional coding is required for implementing a `simulation_model` method serving as a wrapper for the models developed in different programming languages. Simple examples of wrappers for the models implemented in Fortran are presented in the file *iam_simple_models.py* located in the *examples/scripts* folder of the NRAP-Open-IAM distribution and described in the next section.

3.4 Method `simulation_model` as a wrapper

In some situations the developer of the component might have a code written in language different from Python (e.g., Fortran), and rewriting the code in Python might not be worth the efforts. In this section, we present an approach which is used for some components in NRAP-Open-IAM and describes integration of the Fortran code. We start with a simple example of a Fortran routine involving different types of input parameters and results (model outputs).

```
subroutine quad_eq_fun(a, b, c, x, N, x1, x2, y, flag) bind(C, name='quad_eq_fun')
implicit none
real*8, intent(in) :: a
real*8, intent(in) :: b
real*8, intent(in) :: c
real*8 :: D
real*8, intent(out) :: x1
real*8, intent(out) :: x2
integer, intent(in) :: N
integer, intent(out) :: flag
integer :: i
real*8, parameter :: epsil = 1d-20
real*8, dimension(1:N), intent(in) :: x
real*8, dimension(1:N), intent(out) :: y

D = b*b - 4.0*a*c

if (abs(D) > epsil) then
    if (D .GT. 0.0) then
        x1 = (-b + sqrt(D))/(2.0*a)
        x2 = (-b - sqrt(D))/(2.0*a)
        flag = 1                  ! indicates two real distinct roots
    else
        x1 = -b/(2.0*a)           ! returns real part of the roots
        x2 = sqrt(-D)/(2.0*a)     ! returns complex part of the roots
        flag = 3
    endif
else
    x1 = -b/(2.0*a)
    x2 = x1
    flag = 2
endif
```

(continues on next page)

(continued from previous page)

```
do i = 1, N
    ! calculates parabola y-values for the entered parameters a, b, c and x-values
    y(i) = a*x(i)*x(i) + b*x(i) + c
end do

return
end subroutine quad_eq_fun
```

First, for the Fortran function to be used in the Python code it should be compiled into a library with the Fortran compilers. Compilation of the Fortran code should be tested for all supported platforms. We assume that the users working on Mac or Linux platforms should be able to easily and properly compile the code into corresponding libraries. For Windows users the compiled libraries will be provided. For illustration purposes we assume that the example Fortran routine is located in file *iam_quad_eq_fortran_model.f90*. In order to compile the above code in Windows one can use the gfortran compiler installed and run in a Cygwin environment as

```
$ gfortran -c iam_quad_eq_fortran_model.f90
$ gfortran -shared -o quad_eq_fun.dll iam_quad_eq_fortran_model.o
```

Option `-c` of the first step directs gfortran to compile the Fortran file to an object file, rather than producing a standalone executable. This flag should be used if the program source code consists of several files. The object files produced by this command can later be linked together into a complete program.

The compilation of the code on Mac or Linux differs only in the second line. In Mac the second step is

```
$ gfortran -dynamiclib -o quad_eq_fun.dylib iam_quad_eq_fortran_model.o
```

For Linux the steps look like

```
$ gfortran -fpic -c iam_quad_eq_fortran_model.f90
$ gfortran -shared -fpic -o quad_eq_fun.so iam_quad_eq_fortran_model.o
```

The resulting library files for Windows, Mac and Linux will be *quad_eq_fun.dll*, *quad_eq_fun.dylib*, and *quad_eq_fun.so*, respectively.

The following code demonstrates calling the compiled code within a `simulation_model` method. The method returns the y-coordinates of the points on a parabola $y = ax^2 + bx + c$ also calculated inside library for defined x-values and coefficients a, b, c.

```
def simulation_model(p, input_array_x=None):
    '''
    Return three outputs based on the provided input parameters.

    :param p: dictionary of parameters passed to the function
    :type p: dict

    :param input_array_x: input array
    :type input_array_x: numpy.array

    :returns dictionary containing output variable and array
    '''
    # Obtain the default values of the parameters from dictionary
    # of default parameters
    actual_p = dict([(k,v.value) for k,v in self.default_pars.items()])
```

(continues on next page)

(continued from previous page)

```
    # Update default values of parameters with the provided ones
    actual_p.update(p)

    a = actual_p['a']
    b = actual_p['b']
    c = actual_p['c']

    if input_array_x is None:
        input_array_x = np.linspace(-10.0, 10.0, 1000)

    # Determine the size of input array
    N = np.size(input_array_x)

    # Setup library and needed function names
    if platform == "linux" or platform == "linux2":
        # linux
         library = "quad_eq_fun.so"
    elif platform == "darwin":
        # OS X
        library = "quad_eq_fun.dylib"
    elif platform == "win32":
        # Windows...
        library = "quad_eq_fun.dll"
    functionName = "quad_eq_fun"

    # Load DLL
    external_lib = ctypes.cdll.LoadLibrary(os.path.join(os.getcwd(),library))

    # Get needed function as attribute of the library
    function = getattr(external_lib, functionName)

    # Define c classes to be used for inputs and outputs of the Fortran function
    INT = ctypes.c_int
    DOUBLE = ctypes.c_double
    NPointsArrayType = DOUBLE*N

    # Set argument types for values and pointers
    # The order should coincide with the order of arguments in the Fortran function
    function.argtypes = [ctypes.POINTER(DOUBLE),  # type for argument a
                         ctypes.POINTER(DOUBLE),  # type for argument b
                         ctypes.POINTER(DOUBLE),  # type for argument c
                         ctypes.POINTER(DOUBLE),  # type for array argument x
                         ctypes.POINTER(INT),     # type for integer argument N
                         ctypes.POINTER(DOUBLE),  # type for output x1
                         ctypes.POINTER(DOUBLE),  # type for output x2
                         ctypes.POINTER(DOUBLE),  # type for array output y
                         ctypes.POINTER(INT)]     # type for integer output flag
    function.restype = None

    # Define values of the input parameters that will be passed
    # to the Fortran function
    fun_arg_a = DOUBLE(a)
```

(continues on next page)

(continued from previous page)

```
    fun_arg_b = DOUBLE(b)
    fun_arg_c = DOUBLE(c)
    fun_arg_N = INT(N)
    fun_array_x = NPointsArrayType(*input_array_x)

    out_x1 = DOUBLE()                    # initialize output variable x1
    out_x2 = DOUBLE()                    # initialize output variable x2
    out_flag = INT()                     # initialize output variable flag
    out_array_y = NPointsArrayType()     # initialize output array

    function(fun_arg_a, fun_arg_b, fun_arg_c, fun_array_x, fun_arg_N,
             out_x1, out_x2, out_array_y, out_flag)

    # Create output dictionary
    out = dict()
    # The present simulation model is supposed to return sum and absolute
    # value of difference of the roots and y-values of the function.
    # The following calculations depend on the value of flag
    # Check flag values
    if out_flag.value==1:            # two real distinct roots
        # Extract value from the float type of output
        out['root_sum'] = abs(out_x1.value + out_x2.value)
        out['root_diff'] = abs(out_x1.value - out_x2.value)
    elif out_flag.value==2:          # single real root
        out['root_sum'] = abs(2*out_x1.value)
        out['root_diff'] = 0.0
    else:                            # two complex roots
        # Sum of roots is a doubled real part
        out['root_sum'] = abs(2*out_x1.value)
        out['root_diff'] = abs(2*out_x2.value)

    out['y'] = out_array_y[0:N]   # extract values from array type of output

    # Return output dictionary
    return out
```

3.5 Method `check_input_parameters`

The purpose of the `check_input_parameters` method is to ensure that the parameters supplied to the `simulation_model` method satisfy the model assumptions and/or limitations. Below we provide the default `check_input_parameters` method of the `ComponentModel` class.

```
def check_input_parameters(self, p):
    """
    Check whether input parameters satisfy the specified boundaries.

    :param p: input parameters of component model
    :type p: dict
    """
    # Import of logging package can be done at the module level
```

(continues on next page)

(continued from previous page)

```
    # above the definition of ComponentModel class
    import logging
    logging.debug(
        'Input parameters of component {name} are {p}.'.format(
            name=self.name, p=p))

    if hasattr(self, 'pars_bounds'):
        for key, val in p.items():
            if key in self.pars_bounds:
                if ((val < self.pars_bounds[key])or(val > self.pars_bounds[key])):
                    logging.warning(
                        'Parameter {key} is out of boundaries.'.format(key=key))
            else:
                logging.warning(('Parameter {key} is not recognized as '+
                                 'component {name} input parameter.').format(
                                     key=key, name=self.name))
    else:
        logging.debug(('Component {name} does not define boundaries of '+
                       'its model parameters.').format(name=self.name))
```

Note that there is no `print` function used in this example. NRAP-Open-IAM utilizes the python package `logging` for the functionality needed to provide the user with different kinds of messages related to the performance and setup of the system model. The single argument of the `check_input_parameters` method is a dictionary of simulation model parameters *p*, the same dictionary provided to the `simulation_model` method. Since parameters do not vary in time the method is called only once per simulation. We recommend to use a separate method for time-varying model inputs, with possible (among others) name `check_temporal_inputs`. The method `check_temporal_inputs` should be setup to be called for each time step within the `simulation_model` method of the component. The default `check_input_parameters` method assumes that the parameters boundaries are defined in the `pars_bounds` attribute of the `ComponentModel` class object. If the attribute `pars_bounds` is not defined by the developer, a message will be printed to the log to indicate this.

The method can be redefined within the class derived from the `ComponentModel` class. The following code snapshot provides an example of the method modified according to the needs of `LookupTableReservoir` component class.

```
def check_input_parameters(self, p):
    """
    Check whether input parameters fall within specified boundaries.

    :param p: input parameters of component model
    :type p: dict
    """
    debug_msgs = ['Checking input parameters...',
                  'Input parameters {}'.format(p)]
    for ind in [0, 1]:
        logging.debug(debug_msgs[ind])

    if not self.linked_to_intr_family:
        err_msg = ''.join(['Application cannot proceed further. ',
                           'Lookup Table Reservoir component is ',
                           'not linked to any interpolator family.'])
        logging.error(err_msg)
        raise LinkError(err_msg)
```

(continues on next page)

(continued from previous page)

```
    # Save interpolators names
    if self.intr_names is None:
        self.intr_names = {}
    for intr_nm, intr_obj in self._parent.interpolators[self.intr_family].items():
        self.intr_names[intr_obj.index] = intr_nm

    # For lookup table reservoir component we need to make sure that the
    # signature created with input parameters coincide with the signature
    # of one of the interpolators used by component

    # Create signature based on default parameter values
    param_signature = {k: v.value for k, v in self.default_pars.items()}

    # Check whether there are any parameters not belonging to the signature
    for key in p:
        if key not in param_signature:
            msg = ''.join([
                'Parameter {key} not recognized as ',
                'a LookupTableReservoir input parameter.']).format(key=key)
            logging.warning(msg)

    # Update default signature with values of input parameters p
    param_signature.update(p)

    # Extract index from updated dictionary
    index = int(param_signature.pop('index'))
    if index != -2:
        if index not in self.intr_names:
            err_msg = ''.join([
                'Value {} of index parameter does not correspond ',
                'to any of the linked interpolators.']).format(index)
            logging.error(err_msg)
            raise ValueError(err_msg)
    else:
        # Check for the same signature among all connected interpolators
        signature_found = False
        for interpr in self._parent.interpolators[self.intr_family].values():
            # Compare signature of interpolator with the provided input parameters
            if interpr.signature == param_signature:
                signature_found = True
                break

        if not signature_found:
            err_msg = ''.join([
                'Signature of input parameters do not coincide with ',
                'signatures of connected interpolators {}.']).format(param_signature)
            logging.error(err_msg)
            raise ValueError(err_msg)
```

First, the method checks whether the component is linked to the interpolators, needed for the proper execution of the `simulation_model` method, and only after that checks whether the input parameters satisfy the model requirements. The next example of the `check_input_parameters` method taken from the `MultisegmentedWellbore` component class is close to the default one but represents a modified version of the original method and accounts for the varying number of possible (and similar) model parameters.

```python
def check_input_parameters(self, p):
    """
    Check whether input parameters fall within specified boundaries.

    :param p: input parameters of component model
    :type p: dict
    """
    debug_msg = 'Input parameters of component {} are {}'.format(self.name, p)
    logging.debug(debug_msg)

    for key, val in p.items():
        warn_msg = ''.join([
            'Parameter {} of MultisegmentedWellbore component {} ',
            'is out of boundaries.']).format(key, self.name)
        if (key[0:5] == 'shale' and key[-9:] == 'Thickness'):
            if (val < self.pars_bounds['shaleThickness'][0]) or (
                    val > self.pars_bounds['shaleThickness'][1]):
                logging.warning(warn_msg)
            continue
        if key[0:7] == 'aquifer' and key[-9:] == 'Thickness':
            if (val < self.pars_bounds['aquiferThickness'][0]) or (
                    val > self.pars_bounds['aquiferThickness'][1]):
                logging.warning(warn_msg)
            continue
        if key[0:7] == 'logWell' and key[-4:] == 'Perm':
            if (val < self.pars_bounds['logWellPerm'][0]) or (
                    val > self.pars_bounds['logWellPerm'][1]):
                logging.warning(warn_msg)
            continue
        if key[0:6] == 'logAqu' and key[-4:] == 'Perm':
            if ((val < self.pars_bounds['logAquPerm'][0]) or
                    (val > self.pars_bounds['logAquPerm'][1])):
                logging.warning(warn_msg)
            continue
        if key[0:3] == 'aqu' and key[-18:] == 'BrineResSaturation':
            if ((val < self.pars_bounds['aquBrineResSaturation'][0]) or
                    (val > self.pars_bounds['aquBrineResSaturation'][1])):
                logging.warning(warn_msg)
            continue
        if key in self.pars_bounds:
            if ((val < self.pars_bounds[key][0]) or
                    (val > self.pars_bounds[key][1])):
                logging.warning(warn_msg)
```

3.6 Method `reset`

The method `reset` is called once before each simulation. Its purpose is to reset all the needed attributes, parameters, accumulators and observations of the particular component to some initial state.

```
def reset(self):
    """
    Reset parameters, observations and accumulators.

    Parameters, observations and accumulators are reset to their
    initial/default values at the beginning of each new simulation.
    """
    pass
```

In the default method of the `ComponentModel` class there is only one statement: `pass`. The current default method serves as a placeholder for the method that can be redefined within derived component classes. Due to the inherent multitude of different features possibly needed by component models, it is not practicable to write a method common for all components; so the “proper” implementation of the method is left to the developer. In the current version of the NRAP-Open-IAM, the `SystemModel` instance method `single_step_model` sets the values of all accumulators of the system components to zero before each simulation: the accumulators are assumed to accumulate sum-like quantities (e.g., cumulative masses and/or volumes). For some accumulators the initial value of zero is assumed mainly out of convenience. The redefinition of the `reset` method is needed if, for example, the accumulator is supposed to keep track of product-like quantities and the initial value should be 1, or for some other reason an initial value of zero does not work. Additionally, the `reset` method can be used for setting other component variables that need to be reinitialized before each simulation. Setting the accumulator or other attribute values within the `reset` method is straightforward. Referring to the same accumulators we used for the example of the method `_init_` above, we replace the `pass` statement with appropriate statements for this case. Note that we assume that the initial value of `accumulator1` is 0, and the initial value of `accumulator2` is 1, thus, only the `accumulator2` has to be reinitialized within the method.

```
def reset(self):
    """
    Reset parameters, observations and accumulators.

    Parameters, observations and accumulators are reset to their
    initial/default values at the beginning of each new simulation.
    """
    self.accumulators['accumulator2'].sim = 1.0
    # self.accumulators['accumulator1'].sim = 0.0
```

3.7 Connecting components

There are many defaults methods of the `ComponentModel` class that are not meant to be reimplemented within the derived component class. The main purpose of these methods is to provide means to connect the components within the system model and setup the parameters and observations of each component model. By providing the examples illustrating the functionality of the available methods, we want to show the capabilities of NRAP-Open-IAM which might be limited or limit the development of new component models. Knowing what connections can be created between models helps to make sure that the new model is consistent with the available framework and can be merged seemlessly. On the other hand, many of these methods were added or modified based on the feedback of active developers, that is, new development is possible after a justified proposal and review.

Attribute `component_models` of a `SystemModel` class object is an ordered dictionary containing references to all component models involved in the simulation. The order in which the components are arranged in the system model is important and represents the order in which the corresponding `simulation_model` methods are called during the simulation. Note that the component `simulation_model` method should be developed in a way that would allow it to be called at each time step provided by system model. After all component models are run for a given time point, the control is returned back to the system model. The instance method `reorder_component_models` of the `SystemModel` class changes the order of the components in the system model after they have been added to the container, and can be used to fix the order components models that have been added out of execution order.

```
def reorder_component_models(self, order):
    """
    Reorder execution order of component models.

    :param order: list of component model names in desired execution order
    :type order: lst(str)
    """
    self.component_models = OrderedDict((k, self.component_models[k]) for k in order)
```

There are many methods of the component model class used to define the connections between the components that

- determine parameters of the model that the user can control (modify values) and specify observations that the user can analyze;
- determine which of the observations of a given component can provide input parameters of the next component;
- define which of the parameters of a given component model can be defined (calculated) in terms of the parameters of another component model;
- are used either at the script writing stage or are more likely to be used within the class derived from `ComponentModel`.

3.7.1 Parameters

We start with the description of the methods that allow the addition of parameters and observations to the component models.

The `add_par` method can be used to add deterministic or stochastic parameter by utilizing different values of the input argument *vary*. To add deterministic parameter one would use

```
cm.add_par(name=par_name, value=par_value, vary=False)
```

Here, *cm* is a name of variable containing a reference to the `ComponentModel` class instance.

By default the added parameter is assumed to be stochastic, thus, to add stochastic parameter one would use

```
cm.add_par(name=name, min=min_value, max=max_value, value=value)
```

Deterministic parameters of the component models can be accessed by referring to `deterministic_pars` attribute of the component model class instance, while the stochastic parameters of the component are kept in the `pars` attribute. The stochastic parameters are also added to and tracked by the system model to which the component belongs. Similar to the component model, the parameters can also be accessed through the `pars` attribute of the system model. The only difference is that the system model `pars` attribute contain all stochastic parameters of the system, i.e. stochastic parameters of all components.

The next method add_default_par is often used in the __init__ method of the component model class. The primary purpose of this method is to ensure that all parameters of the model are defined during the simulation even when not explicitly defined by the user. This is a convention that all existing component models of NRAP-Open-IAM rely on. If the parameter of a given component model is not defined by the user then the model itself should take care of the default value of the parameter, for example, assigning the value of the parameter not determined by the user within the simulation_model method of the derived class.

As mentioned above the common approach to the default parameters is to add them in __init__ method as follows (the values of the parameters are chosen arbitrarily for illustration)

```
self.add_default_par('par_a', value=2.71)
self.add_default_par('par_b', value=3.14)
```

Usually the two lines above would be followed by the definition of the component attribute pars_bounds which would describe the lower and upper boundaries of each model parameter (see notes above and example below).

```
# Define dictionary of boundaries
self.pars_bounds = dict()
self.pars_bounds['par_a'] = [1.0, 15.0]
self.pars_bounds['par_b'] = [-100.0, 5.0]
```

Note that the default value of the parameters can be either on the boundary or inside the interval specified by pars_bounds.

Definition of parameters default values allows utilizing them by other components. Other component models can use the default parameter value by referring to the default_pars attribute of the component model class instance. For example,

```
cm.add_default_par(par_name)
print(cm.default_pars[par_name])
```

3.7.2 Observations

The following methods allows adding observations that will be tracked by the system model and will be available for different kinds of analysis. Component models responses not explicitly added using these methods are ignored and are not available for the processing once the simulation is complete.

Note that the method add_obs allows adding only scalar observations. The references to the observations of the component models are kept in the identically named attributes obs of the component model and system model. If the component model is supposed to return a structured (or "gridded") observation (array, matrix) then the observation should be added with the method add_grid_obs.

Due to the possibly large size, the structured observations are not tracked by the system model (i.e. not available by reference to obs[obs_name] attribute of the system model) but the simulated structured observations are saved in the compressed *.npz* format files with a filename defined by a pattern: component name, gridded observation name, simulation number, and time index all joined by undescore symbol _. For example, if the component with name *well* has a structured observation with name *rate* then the file which contains the mentioned observation evaluated at time point with index 10 will be named:

- *well_rate_sim_0_time_10.npz* for a single forward run of a system model and
- *well_rate_sim_1_time_10.npz*, *well_rate_sim_2_time_10.npz*, … for multiple stochastic simulations of a system model.

The reference to the properties of the structured observations are kept in the `grid_obs` attribute of the component model class instance. One can see the names of the gridded observations added to the output of the given component by accessing the mentioned attribute:

```
cm.add_grid_obs('grid_obs1', constr_type='array')
print(cm.grid_obs.keys())
print(cm.grid_obs['grid_obs1'])
```

If only a single element of the structured (gridded) observation (array) is needed, then method *add_local_obs* should be used. Since in this case the observation is a scalar (not an array) the system model keeps track of the observation.

Since the local observation is derived from the structured observation, the component model additionally keeps the index (or tuple of indices) of the element in an array (or matrix). In order to get the index of the local observation in the structured observation one has to know the name of the structured observation from which the local observation is derived (name of the structured observation is provided by the given component model) and the name of the local observation assigned by the user. For example,

```
# Add local observation
cm.add_local_obs(loc_obs_name, grid_obs_name)

# Print the index of the local observation
print(cm.local_obs[grid_obs_name][loc_obs_name])
```

If the observation of a given component model is to be used as input for other model it should be specifically added as such with the method `add_obs_to_be_linked`. For example,

```
cm.add_obs_to_be_linked(obs_name)
```

If the observation in addition should be tracked by the system model then it also should be added with the method `add_obs`, i.e.

```
cm.add_obs(obs_name)             # to be tracked by system model
cm.add_obs_to_be_linked(obs_name)  # to be used as input for other component
```

One important thing to mention before we continue with the next methods is that the data (parameters, keyword arguments, observations) passed between models need to have consistent units. There are some assumptions about the units of the data that comes into and out of the component models. So here is a list of units used by the NRAP-Open-IAM.

- Pressure is assumed to be in units of Pascals (Pa).
- Time is assumed to be in days (d).
- Length type parameters (distance, width, etc.) are assumed to be in units of meters (m).
- Fluxes of fluids are assumed to be in units of kilograms per second (kg/s).
- Masses are assumed to be in units of kilograms (kg).
- Viscosities are assumed to be in units of Pascal seconds (Pa s).

3.7.3 Linked parameters and observations

The following methods allow adding component model parameters that depend on other parameters and/or observations of the same or different component model. Method add_par_linked_to_par adds a parameter that has the same properties and value as an already defined parameter.

In general, any of the model parameters can be of any allowed type. For example, in one scenario a particular parameter can be setup as stochastic, in another it can be setup as deterministic. Consider an example of the code utilizing the add_par_linked_to_par method. As in the previous examples, the values of the parameters are defined arbitrarily.

```
# Assume we have component model cm1 defined somewhere above in the code
# as instance of Component1 class with name 'cmpnt1'
cm1.add_par('par_1', value=4.5, min=2.0, max=5.0)  # added parameter is stochastic
cm1.add_par('par_2', value=1.8, vary=False)  # added parameter is deterministic
cm1.add_default_par('par_3', value=1.5)      # added parameter is default

# Now assume there is another component model cm2 also defined as
# an instance of Component1 class with name 'cmpnt2'. We also assume that cm2
# have parameters defined as dependent on the parameters of component model cm1
# The corresponding dictionary should be used for each type of parameters of
# component model cm1
cm2.add_par_linked_to_par('par_1', cm1.pars['par_1'])
cm2.add_par_linked_to_par('par_2', cm1.deterministic_pars['par_2'])

# The names of the parameters of component cm2 and linked parameters of component cm1
# should not necessary be the same if this is what is intended
cm2.add_par_linked_to_par('par_3', cm1.default_pars['par_4'])
cm2.add_par('par_4', value=2.5)                # added parameter is default
```

Method add_par_linked_to_obs adds a parameter that obtains its value from the output of some other component. The observation that the added parameter depends on should be added with the add_obs_to_be_linked method.

Consider the following example on the use of the method.

```
# Add observation of component cm1 to be used as parameter of component cm2
cm1.add_obs_to_be_linked('obs_1')
# Add parameter of component cm2: parameter name of component cm2 does not
# necessarily coincide with the name of observation returned by component model cm1
cm2.add_par_linked_to_obs('par_1', cm1.linkobs['obs_1'])
```

Since the work of NRAP-Open-IAM is based on the assumption that parameters of the component model are constant in time, the use of method add_par_linked_to_obs is appropriate only in situations when the observation that is linked to the parameter does not vary in time.

Method add_composite_par adds a parameter whose value is determined by an expression which may contain references to parameters of the same and/or other components.

The following piece of code contain several examples utilizing add_composite_par method.

```
# Assume we have component model cm1 defined somewhere above in the code
# as instance of Component1 class with name 'cmpnt1'
cm1.add_par('par_1', value=4.5, min=2.0, max=5.0) # added parameter is stochastic
cm1.add_par('par_2', value=1.8, vary=False)  # added parameter is deterministic
cm1.add_default_par('par_3', value=1.5)      # added parameter is default

# Now assume there is another component model cm2 also defined as
# an instance of Component1 class with name 'cmpnt2'.
cm2.add_par_linked_to_par('par_1', value=2.5, min=2.1, max=4.9, vary=True)
cm2.add_par_linked_to_par('par_2', min=2.0, max=5.0, vary=True)

# We define par_3 of component cm2 as a sum
# of the first three parameters of component cm1
cm3.add_composite_par('par_3', expr='cmpnt1.par_1+cmpnt1.par_2+cmpnt1.par_3')

# We define par_4 of component cm2 as a difference
# of the first parameters of component cm1 and component cm2
cm3.add_composite_par('par_4', expr='cmpnt1.par_1-cmpnt2.par_1')
```

Note that the expression for each added parameter directly utilizes the name of the parameter which consists of the parental component name and parameter name as defined during the component setup separated by a dot **.**. It does not require knowing the parameter type (stochastic, deterministic, default). There is a different way to write an expression for composite parameters which utilizes the variables containing references to the corresponding components and names of the parameters used in the expression. For example, line

```
cm3.add_composite_par('par_3', expr='cmpnt1.par_1+cmpnt1.par_2+cmpnt1.par_3')
```

can be replaced with

```
cm3.add_composite_par('par_3', expr='+'.join([cm1.pars['par_1'].name,
                                              cm1.deterministic_pars['par_2'].name,
                                              cm1.default_pars['par_3'].name]))
```

Then line

```
cm3.add_composite_par('par_4', expr='cmpnt1.par_1-cmpnt2.par_1')
```

can be replaced with

```
cm3.add_composite_par('par_4', expr='-'.join([cm1.pars['par_1'].name,
                                              cm2.pars['par_1'].name]))
```

This method does not require knowing the name of the parental component but rather the name of the variable that keeps the reference to the corresponding component and the type of parameters involved in the expression for the composite parameter.

3.7.4 Keyword arguments

We discussed previously that inputs to the component model can be of two main types: constant in time, scalar numerical parameters and (possibly) varying in time model arguments. If the component's `simulation_model` method requires one or more of the later types, the model arguments have to be added to the component model using one of the methods discussed below. If the argument of the `simulation_model` method does not change with time: for example, cannot be defined as a parameter of the model but might change from one setup of the component to another, the simplest way to define it is to make it an argument of the constructor method `__init__`. For example, see the definition above of *time_step* as both an argument of `__init__` and `simulation_model` methods. Adding of `'time_step'` key (and/or `'time_point'`, `'time_index'` keys) to the dictionary attribute `model_kwargs` of the component tells the system model that these arguments of the component `simulation_model` method are the same arguments provided by the system model. If the attribute `model_kwargs` of the component model does not contain keys `'time_step'`, `'time_point'` and `'time_index'` the component model will not be aware of the values provided by the system model and will have to utilize the default values provided with the definition of the `__init__` method.

```
# Inside code of the __init__ method
self.model_kwargs = {'time_step': 365.25, 'time_point': 365.25}
```

…

```
# Inside script setting component model
cm = Component1(name='cm1', parent=sm)  # sm is a system model cm belongs to
cm.model_kwargs['time_step'] = 365.25
```

Argument *time_point* is an argument of the component model that changes in time but is defined by the system model. To define the arguments that change in time in a predetermined way, one can use method `add_dynamic_kwarg`.

The main purpose of this method is to allow writing the scripts and/or tests for a single component in the system model whose arguments otherwise would have to depend on the output of other components. In the code example below, we use the definition of the `Component1` class defined above.

```
# Create a component
cm = Component1(name='cm1', parent=sm)

# Create an array of ten random real numbers between 1 and 5 that would
# serve as an input for the model method
t_array = 1.0 + 4.0*np.random.rand(10)
# Add dynamic argument of the model method
cm.add_dynamic_kwarg('temp_input1', t_array)
```

To satisfy possible needs of keyword arguments types of model methods, `add_kwarg_linked_to_obs` and `add_kwarg_linked_to_collection` were added to the inventory of `ComponentModel` instance methods. Method `add_kwarg_linked_to_obs` can be used to connect keyword argument of one component to the observation of another component.

Observation to which keyword arguments are to be linked should be added to the corresponding components (observation provider) as such with `add_obs_to_be_linked` method. Keyword arguments of the `simulation_model` method can be linked to both types of observations in NRAP-Open-IAM: scalar and gridded/structured. Depending on the type of the linked observation a corresponding set of arguments should be used. We consider several examples illustrating the possible use of this method.

```
# Suppose that component cm1 simulation_model method returns observation with name obs1
# that can serve as a keyword argument for the component cm2.
# Add observation of component cm1 to be linked to the argument
# of the second component. Note that use of the obs_type argument is not
# necessary when observation is scalar. Here, it is used to emphasize this fact
cm1.add_obs_to_be_linked('obs1', obs_type='scalar')

# Add keyword argument of cm2 linked to the observation of component cm1
cm2.add_kwarg_linked_to_obs('input1', cm1.linkobs('obs1'))
```

As illustrated in the example the names of the keyword argument and of the observation provided as input should not necessary be the same. The next example illustrates the situation when the keyword argument is linked to the gridded observation. Keyword argument can be linked to any part of the gridded observation. Recall that the gridded observation should be returned either as an array or matrix:

```
# Suppose that component cm1 simulation_model method returns gridded observation
# with name grid_obs1 that can serve as a keyword argument for the components
# cm2, cm3 and cm4 linked to the different parts of the gridded observation
# Add observation of component cm1 to be linked to the argument
# of the second component. Note that use of the obs_type argument
# is not necessary when observation is scalar.
# Here, it is used to emphasize the type of observation
cm1.add_obs_to_be_linked('grid_obs1', obs_type='grid')

# It is a responsibility of the user to make sure that the returned
# observation types are compatible with the format of keyword arguments
# accepted by the subsequent components.
# Add keyword argument of cm2 linked to the observation of component cm1
cm2.add_kwarg_linked_to_obs('input1', cm1.linkobs('grid_obs1'))

# Add keyword argument of cm3 linked to the several elements
# of observation of component cm1
cm3.add_kwarg_linked_to_obs('input1', cm1.linkobs('grid_obs1'),
                            constr_type='array', loc_ind=list(range(10)))

# Add keyword argument of cm4 linked to a single element
# of observation of component cm1
cm4.add_kwarg_linked_to_obs('input2', cm1.linkobs('grid_obs1'),
                            constr_type='array', loc_ind=[0])
```

There are differences in three uses of the method `add_kwarg_linked_to_obs` that we want to discuss next. The keyword argument *input1* of the component *cm2* `model` method is linked to the gridded observation with name *grid_obs1* provided by component *cm1*. We do not use any extra arguments of the method which shows that all observation data provided by *cm1* is copied to the keyword argument. The keyword argument *input1* of the component *cm3* is linked only to the part of the gridded observation with name *grid_obs1* provided by component *cm1*. We indicate this by specifying the list of indices of observation array at which it should be provided to the keyword argument *input1*. Note that both components *cm2* and *cm3* should be able to accept the array-like keyword argument *input1*. With component *cm4* the situation is slightly different. Its keyword argument *input2* is also linked to the gridded observation *grid_obs1* but it requests a single value of the gridded observation indicated by a single index (in the example it is 0) of the element in observation array. Note that although there is only a sinle index it still has to be provided in the list format. The value of the observation array will be passed as a scalar variable rather than array-like type as it is done for components *cm2* and *cm3*.

The following method is used to add a keyword argument created from a collection of scalar observations provided by the same or different components. Essentially, the observations are combined into one structure (collection) and passed in this form to the corresponding component.

The example below illustrates the use of the method for linking to the collection of similar observations.

```
# Suppose that 5 components references to which are stored in a list variable sup_cm
# can return an observation with name obs1
# Add observation of components to be used to create a collection of observations
for ind in range(5):
    # Option obs_type='scalar' can be omitted
    sup_cm[ind].add_obs_to_be_linked('obs1', obs_type='scalar')

# Create a list of references to just created linked observations
obs_collection = [sup_cm[ind].linkobs('grid_obs1') for ind in range(5)]

# Add keyword argument of cm2 linked to the collection
cm2.add_kwarg_linked_to_collection('input1', obs_collection)
```

The method is used to link to the collection of scalar observations. We emphasize this by specifying option obs_type, non-mandatory for scalar observations. The following simple example illustrates the situation when the collection is created from not necessarily similar observations.

```
# Suppose that component cm1 simulation_model method returns observation with name obs1
# Add observation obs1 of component cm1 to be used to create
# a collection of observations. Note that we omit option obs_type='scalar'
# since it is not needed
cm1.add_obs_to_be_linked('obs1')

# Suppose that component cm2 simulation_model method returns observation with name obs2
# Add observation obs2 of component cm2 to be used to create
# a collection of observations. Note that we omit option obs_type='scalar'
# since it is not needed
cm2.add_obs_to_be_linked('obs2')

# Create a list of references to just created linked observations
obs_collection = [cm1.linkobs('obs1'), cm2.linkobs('obs2')]
# Add keyword argument input1 of component cm3
cm3.add_kwarg_linked_to_collection('input1', obs_collection)
```

In this example we assume that the `simulation_model` method of component *cm3* would accept an argument *input1* which in this case is a 2-element array.

CHAPTER

FOUR

CODE DOCUMENTATION

4.1 Docstrings

Since the integration of the component based on simulation model into the NRAP-Open-IAM framework will not necessarily be performed by the developers who initially developed the model, it is necessary for the model developers to provide some level of details covering the possible use of the model. The description of the code and logistics behind the model is provided in comments and dosctrings written within the code. Docstrings are a “special form” of comments. They usually occur as the first statement in a class, method or function description. We strongly recommend to include dosctrings in all modules written for the NRAP-Open-IAM tool. Information provided in docstrings is used to create the components description section in the NRAP-Open-IAM user’s manual. This means that docstrings describing the components should follow the specific format. For examples of the desired format, the developer should consult the available components code. Here, we consider the docstring for the CementedWellbore component class which is the first statement one would see after opening a Python file with the component code. This type of docstrings is referred to as a module string. The pieces of code considered above also contain docstrings.

```
"""
The Cemented Wellbore component model is based on a multiphase well leakage
model implemented in the NRAP-IAM-CS, :cite:`HARP2016150`. The model is
built off detailed full-physics Finite Element Heat and Mass (FEHM)
simulations, :cite:`Zyvoloski2007`. The FEHM transfer simulations are
three-dimensional (3-D), multiphase solutions of heat and mass transfer of
water and supercritical, liquid, and gas |CO2|. After the simulations are
completed, the surrogate model is built based on the key input parameters
and corresponding output parameters. The approximate (surrogate) model is
represented by polynomials in terms of input parameters that then can be
sampled to estimate leakage rate for wells.  Early development work can be found
in :cite:`RN1606`.

When using the control file interface with more than 3 shale layers,
the *ThiefZone* keyword can be used to specify the thief zone aquifer and the
*LeakTo* keyword can be specified to name the upper aquifer.  These values will
default to 'aquifer1' and 'aquifer2' respectively.

Component model input definitions:

* **logWellPerm** [|log10| |m^2|] (-13.95 to -10.1) - logarithm of wellbore
  permeability (default -13).

* **logThiefPerm** [|log10| |m^2|] (-13.995 to -12) - logarithm of thief zone
  permeabilty (default -12).
```

(continues on next page)

(continued from previous page)

```
* **wellRadius** [m] (0.025 to 0.25) - radius of the wellbore (default 0.05 m).

* **initPressure** [Pa] (1.0e+5 to 5.0e+7) - initial pressure at the base of
  the wellbore (default 2.0e+7 Pa, or 20 MPa). *From linked component.*

* **wellDepth** [m] (960 to 3200) - depth in meters from ground surface to
  top of reservoir (default 1500 m). *Linked to Stratigraphy.*

* **depthRatio** [-] (0.3 to 0.7) - fraction of well depth to the center of
  the thief zone from the top of the reservoir (default 0.5).
  *Linked to Stratigraphy.*

The possible outputs from the Cemented Wellbore component are leakage rates
of |CO2| and brine to aquifer, thief zone and atmosphere. The names of the
observations are of the form:

* **CO2_aquifer1**, **CO2_aquifer2**, **CO2_atm** [kg/s] - |CO2| leakage rates.

* **brine_aquifer1**, **brine_aquifer2**, **brine_atm** [kg/s] -
  for brine leakage rates.

* **mass_CO2_aquifer1**, **mass_CO2_aquifer2** [kg] - mass of |CO2|
  leaked into aquifers.

"""
```

In this example of a module docstring, the first part of docstring is devoted to the general description of the component, including references for users interested in additional details about the model, and details covering the use of model. The second and the most important part provides the description of all parameters of the model, including the names, short descriptions and ranges. The final part of dosctring contains the names of all possible observations of the `simulation_model` method. When this docstring is compiled as part of the user's guide it looks different from above and is formatted according to the specifications used inside the dosctring. Below is the example of how the CementedWellbore module docstring will be compiled for the document.

Cemented Wellbore Component Docstring Output

The Cemented Wellbore component model is based on a multiphase well leakage model implemented in the NRAP-IAM-CS, [2]. The model is built off detailed full-physics Finite Element Heat and Mass (FEHM) simulations, [8]. The FEHM simulations are three-dimensional (3-D), multiphase solutions of heat and mass transfer of water and super-critical, liquid, and gas CO_2. After the simulations are completed, the surrogate model is built based on the key input parameters and corresponding output parameters. The approximate (surrogate) model is represented by polynomials in terms of input parameters that then can be sampled to estimate leakage rate for wells. Early development work can be found in [3].

When using the control file interface with more than 3 shale layers, the `ThiefZone` keyword can be used to specify the thief zone aquifer and the `LeakTo` keyword can be specified to name the upper aquifer. These values will default to *aquifer1* and *aquifer2*, respectively, if are not provided by user. In the FEHM simulations used to create the surrogate model some of the stratigraphy layers were setup with a fixed thickness. In particular, shale above aquifer had thickness 11.2 m; aquifer and thief zone to which leakage was simulated were set to have thicknesses 19.2 m and 22.4 m, respectively; and reservoir had thickness of 51.2 m.

Component model input definitions:

- **logWellPerm** [$\log_{10}$ m^2] (-13.95 to -10.1) - logarithm of wellbore permeability (default: -13)
- **logThiefPerm** [$\log_{10}$ m^2] (-13.9991 to -12.00035) - logarithm of thief zone permeability (default: -12.2)
- **wellRadius** [m] (0.025 to 0.25) - radius of the wellbore (default: 0.05)
- **initPressure** [Pa] (1.0e+5 to 5.0e+7) - initial pressure at the base of the wellbore (default: 2.0e+7 Pa, or 20 MPa); *from linked component*
- **wellDepth** [m] (960 to 3196.8) - depth in meters from ground surface to top of reservoir (default: 1500); *linked to Stratigraphy*
- **depthRatio** [-] (0.30044 to 0.69985) - fraction of well depth to the center of the thief zone from the top of the reservoir (default: 0.5); *linked to Stratigraphy*.

Temporal inputs of the Cemented Wellbore component are not provided directly to the component model method but rather are calculated from the current and several past values of pressure and CO_2 saturation. The calculated temporal inputs are then checked against the boundary assumptions of the underlying reduced order model. The Cemented Wellbore component model temporal inputs are:

- **deltaP** [Pa] (105891.5 to 9326181.69) - difference between the current and initial pressure at the wellbore
- **pressurePrime** [Pa/s] (-6675.03 to 2986.7) - first pressure derivative
- **pressureDPrime** [Pa/s^2] (-111.265 to 10.806) - second pressure derivative
- **saturation** [-] (0.001 to 1.0) - CO_2 saturation at the wellbore
- **saturationPrime** [$1/s$] (-4.290e-7 to 1.117e-3) - first CO_2 saturation derivative
- **saturationDPrime** [$1/s^2$] (-6.923e-6 to 1.176e-6) - second CO_2 saturation derivative.

The possible outputs from the Cemented Wellbore component are leakage rates of CO_2 and brine to aquifer, thief zone and atmosphere. The names of the observations are of the form:

- **CO2_aquifer1**, **CO2_aquifer2**, **CO2_atm** [kg/s] - CO_2 leakage rates
- **brine_aquifer1**, **brine_aquifer2**, **brine_atm** [kg/s] - brine leakage rates
- **mass_CO2_aquifer1**, **mass_CO2_aquifer2** [kg] - mass of CO_2 leaked into aquifers.

Since we utilize Sphinx for the compilation of the user's guide, all module docstrings provided with the component code have to satisfy the reStructuredText (reST) format that Sphinx relies on. The general docstrings conventions are described in PEP 257, the Python docstrings guide. The main difference between comments and docstrings is that the former explains what a given section of code is doing, while the latter describes how a particular method can be used.

CHAPTER

FIVE

INTRODUCTION TO GIT

5.1 What is Git?

Git is one of the most often used version control systems in the world. The main benefits and features provided for developers employing the version control system are the following:

- system provides means for developers to keep track of code changes,
- it allows developers to see a history of changes, to work on the same code pieces at the same time, to isolate their code through branching, merge code from different branches,
- helps developers see and resolve conflicts on code merges, to revert their changes to a previous state, to merge only selected changes.

Git is an example of what is called a Distributed Version Control System: each developer has a copy of the whole repository on their computer and can see the entire history of changes. Git is also cross-platform which makes it useful when developers are working with different OSs. Developers can divide their work between different branches depending on different priorities. Developers can create experimental features and revert the changes when something goes wrong.

The NRAP-Open-IAM project is hosted at a public repository located at https://gitlab.com/NRAP/OpenIAM. GitLab is a web-based Git-repository manager with wiki and issue-tracking features, using an open-source license. There is also the NRAP-Open-IAM development repository which is private and open only to the members of the development group. The master branch of the development repository NRAP-Open-IAM code is protected. Any new code development is done on separate branches and goes through a review process before merging with the master branch. It ensures that new code does not break existing features and is properly documented and tested. A merge request is approved when the code is complete and ready to be part of the master branch.

In order to start working with the NRAP-Open-IAM the developer should register on https://gitlab.com and provide the user name to be added. Once developer has registered, the code can be downloaded in a compressed format from GitLab. However, the compressed file will only contain the source code; the git repository files including revision history will not be included. To utilize Git during development, the repository should be cloned with the command

```
git clone https://gitlab.com/NRAP/OpenIAM
```

Note that the command uses address for NRAP-Open-IAM repository open for public. The command creates a directory named *OpenIAM* (at the current local file system location), initializes a .git directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version.

If the clone of the repository is needed in a different location, e.g., folder with name *new_NRAPOpenIAM_location*, then the following command should be used:

```
git clone https://gitlab.com/NRAP/OpenIAM new_NRAPOpenIAM_location
```

This command does the same thing as the previous one, but the target directory is now called *new_NRAPOpenIAM_location*.

The installation of Git on the local computer depends on the operating system. The current NRAP-Open-IAM developers use the following Git management systems:

- Windows: Git installed with Cygwin, Git Bash
- Linux, Mac: Git might already be installed locally.

If Git is not available by default, it is worth checking the following sources: for Mac https://git-scm.com/download/gui/mac, and for Linux https://git-scm.com/download/gui/linux. Additional Git clients compatible with Windows operating systems can be found at https://git-scm.com/download/gui/windows .

Comprehensive book on Git which includes more details that can be covered in this guide is available as an open-source project at https://git-scm.com/book/en/v2. The website https://gitlab.com provides an introduction material on use of GitLab and Git in general.

5.2 Most frequently used Git commands

As the name of the current section suggests we provide a list of the most frequently used commands in Table 1.

Table 1: Most common Git commands

Command	Description
Get Help	
$ `git help`	Get help on git.
$ `git help <command>` or $ `git <command> -h`	Get help on any git command.
Configure	
$ `git config --global user.name <name>`	Set the name the developer wants to be attached to the commits.
$ `git config --global user.email <email>`	Set the email the developer wants to be attached to the commits.
$ `git config --list`	List git configuration settings.
Create and Clone	
$ `git init <new-repository>`	Create a new local repository.
$ `git clone <url>`	Clone an existing repository.
Keep Track of Changes	
$ `git status`	List all new and modified files in the working directory.
$ `git diff`	Show not yet staged differences in the tracked files.
$ `git diff --staged`	Show differences between staged and committed version of files.
$ `git add <file or dir>`	Stage <file> or <dir> (take a snapshot of the content) in preparation for commit or stash.
$ `git add -p <file>`	Stage selected changes in <file>.
$ `git add -A`	Stage all changes (new untracked, modified tracked and deleted files) in the working directory.

continues on next page

Table 1 – continued from previous page

Command	Description
`$ git add .`	Stage all new untracked and modified tracked (but not deleted files) in the working directory.
`$ git add -u`	Stage all modified tracked and not deleted files in the working directory.
`$ git reset HEAD <file>`	Unstage <file> but preserve its content.
`$ git checkout -- <file>`	Revert the convert of <file> to the last commit version.
`$ git commit -m <commit-message>`	Record (commit) previously staged changes in version history.
`$ git commit -a`	Commit all changes in tracked files (without staging step).
`$ git commit --amend`	Change the last commit.
Review Commit History	
`$ git log`	Show all commits starting with the most recent.
`$ git log -p <file>`	Show changes over time for a given <file>.
`$ git blame <file>`	Show who changed what and when in a given <file>.
`$ git diff <branch1>...<branch2>`	Show differences in the content of two branches.
Create a Branch	
`$ git branch`	Show all local branches in the current repository.
`$ git branch -av`	Show all existing branches with some extra information.
`$ git branch <new-branch-name>`	Create a new branch based on the current branch.
`$ git branch -d <branch>`	Delete fully merged <branch>.
`$ git checkout <branch>`	Switch to different <branch>.
Update	
`$ git remote -v`	Show all currently configured remotes.
`$ git remote show <name-of-remote>`	Show information about a remote.
`$ git fetch <name-of-remote>`	Download all changes from the <name-of-remote> branch but do not integrate them into the local branch.
`$ git pull <name-of-remote> <branch>`	Download all changes from the <name-of-remote> branch and integrate/merge them into the local <branch>.
`$ git push <name-of-remote> <branch>`	Publish local changes from <branch> to the <name-of-remote>.
`$ git merge <branch>`	Merge <branch> to the current local branch.
`$ git rebase <branch>`	Rebase the current local branch onto <branch>.
`$ git rebase --continue`	Continue a rebase after resolving conflicts.
Save Fragments	
`$ git stash` or `$ git stash -m <stash-entry-message>`	Record the current state of the working directory and go back to the clean working directory.
`$ git stash apply`	Restore the most recently stashed files.
`$ git stash list`	List the currently available stash entries.
`$ git stash clear`	Remove all the stash entries.

CHAPTER

SIX

TESTS AND EXAMPLES

6.1 Tests

Tests and examples can originate from the same process in development but they fulfill very different roles. Tests are the integral part of the code development and QA process. In general, tests are not for the users application. There are many types of tests designed for specific purposes at the different stages of the code development. At this point, the test suite of NRAP-Open-IAM is made up of several types of tests: unit tests (tests of single components), feature tests (particular functionality testing), integration tests (whole system model tests) and installation (setup) tests. Installation test checks user's Python environment and tool functionalility work by starting all unit, feature and integration tests. Each component model distributed as part of NRAP-Open-IAM has at least one test in the test suite meant to check the observation values against expected results. We rely on `unittest`, a Python standard library, to facilitate test writing and execution. It is considered a good practice to run a test suite after each update of the code to make sure that changes do not cause unexpected changes in the output. An approach consistent with the code testing practices of NRAP-Open-IAM development team is described in [5]. Here, we state the most important rules that should be followed with all the new development. Each test should be written for testing a particular feature or new component. The test should be fast since with every major update the new functionality should be tested which increases the total number of tests to be run. The NRAP-Open-IAM tests are located in the file *iam_test.py* in the test *folder*. Below we provide an example of the test written for a component utilizing a Fortran library calculating the roots of the quadratic equation. This component model method was described in detail in Chapter Coding Logistics. Recall that the model method calculates the absolute values of the sum and difference of two roots and values of the quadratic function for entered coefficients and *x*-values. We show the test which would check the solutions of the two quadratic equations - one with both real roots and another one with complex roots - against the known solutions. We create a test for the component as a method of a `ExampleTests` class derived from `unittest.TestCase` class. Note that we do not write a separate component class but rather utilize the available base class `ComponentModel` and create an instance of the class with model method specified by `quad_eq_model` (defined above in Chapter Coding Logistics). The example provided below can be found in the subfolder *scripts* of folder *examples* in the file *iam_simple_models.py*.

```python
class ExampleTests(unittest.TestCase):

    def test_quad_model(self):
        """
        Test work of quadratic model function and corresponding fortran library.
        """
        # Create system model
        sm = SystemModel()

        # Add component model with model function utilizing the fortran library
        # calculating the roots of quadratic equation
        qmc = sm.add_component_model('quad', model=quad_eq_model)
```

(continues on next page)

(continued from previous page)

```
        # Add parameters of qmc component
        qmc.add_par('a', value=2.0)
        qmc.add_par('b', value=2.0)
        qmc.add_par('c', value=-12.0)
        # Add observations of qmc component
        qmc.add_obs('root_sum')
        qmc.add_obs('root_diff')

        # Run forward simulation
        sm.forward()
        # True roots for the defined a, b, c coefficients are -3 and 2.
        # Thus, absolute values of roots sum and difference are 1 and 5.
        true_vals = [1.0, 5.0]
        # Get simulated values of the observations
        sim_vals = [sm.obs['quad.root_sum'].sim, sm.obs['quad.root_diff'].sim]

        # Compare true and simulated values
        for tv, sv in zip(true_vals, sim_vals):
            self.assertTrue(abs((tv-sv)) < 0.001,
                            'The result is {} but should be {}'.format(sv,tv))

        # Check model output for complex roots
        # Change values of the component parameters
        qmc.pars['a'].value = 1.0
        qmc.pars['b'].value = -4.0
        qmc.pars['c'].value = 13.0

        # Run forward simulation
        sm.forward()
        # True roots for the defined a, b, c coefficients are 2+3i and 2-3i.
        # Thus, absolute values of roots sum and difference are 4 and 6.
        true_vals = [4.0, 6.0]
        sim_vals = [sm.obs['quad.root_sum'].sim, sm.obs['quad.root_diff'].sim]

        # Check results
        for tv, sv in zip(true_vals, sim_vals):
            self.assertTrue(abs((tv-sv)) < 0.001,
                            'The result is {} but should be {}'.format(sv,tv))

        return
```

In order to run the test one can use the following script:

```
# Setup test runner
runner = unittest.TextTestRunner(verbosity=2, stream=sys.stderr)
# Create a test suite to which tests can be added
test_suite = unittest.TestSuite()
# Add corresponding test(s)
test_suite.addTest(ExampleTests('test_quad_model'))
# Execute added tests
runner.run(test_suite)
```

A test runner is responsible for the execution of tests and returns the outcome to the user. Test method name must start with word `test`: in our example the name of the test is `test_quad_model`. It signals the test runner which of the methods should be run. Additionally, each of the tests should contain the assertion statement involving one of the following:

- `assertEqual()` to compare the obtained results versus the expected ones; or
- `assertTrue()` or assertFalse() to verify a particular condition; or
- `assertRaises()` to check whether a specific exception gets raised.

There are other available options (e.g., see `unittest` documentation https://docs.python.org/3.6/library/unittest.html) that can be used for specific checks. For example, the code

```
for tv, sv in zip(true_vals, sim_vals):
    self.assertTrue(abs((tv-sv)) < 0.001,
                    'The result is {} but should be {}'.format(sv,tv))
```

can be replaced with

```
for tv, sv in zip(true_vals, sim_vals):
    self.assertAlmostEqual(tv, sv, 2,
                           'The result is {} but should be {}'.format(sv,tv))
```

In the last code snapshot, the "2" argument in the assertAlmostEqual indicates the number of places to which the difference of two values is rounded before comparison with zero.

At the final stages of the development of a new component the component's test should be written and made available as part of the NRAP-Open-IAM test suite.

6.2 Examples

Integration of any component to the NRAP-Open-IAM framework involves not only the development of the tests but also the creation of examples illustrating the utility and capabilities of the new component. Examples are written to show users how the component code works and how to interact with it. Ideally, examples should show off all of the major features of the new development. Writing an example starts with a basic description of the scenario. It helps to start with understanding which additional components (if any) should be utilized. The NRAP-Open-IAM framework allows running the system model with a single component included by utilizing `add_dynamic_kwarg` method of the `ComponentModel`. If the work of the component can be shown without utilizing other components, it makes sense to think whether the additional scenarios, where the component can be linked to other component either as a source of input or output, can be created. The information provided in the section Connecting components in Chapter Coding Logistics can be very useful here. The examples illustrating work of the components currently available as part of the NRAP-Open-IAM can be found in the *examples/scripts* folder of the tool distribution. Examples often utilize the methods for adding and creating component parameters and observations and connecting the components within a given system model. In general, most examples consist of the following steps:

- setup system model: simulation time, time step size or number of time points;
- add component models needed in the scenario;
- add system and components parameters and observations;
- create connections between components through models input/output using appropriate linking methods;
- run a chosen type of analysis: forward simulation, multiple stochastic simulations (Latin Hypercube sampling or Monte Carlo), parameter studies;
- collect observations from the system model;
- post-process the analysis results (if/as needed);

- plot or print the results.

The example files developed for the all components available within the NRAP-Open-IAM tool can be found in the *examples/scripts* folder of the tool distribution. It is strongly encouraged that all example files developed for the tool follow the same naming convention: the file names should start with *iam_* followed either by the list of components used for a specific scenario and/or functionality of the tool featured in the example. The following code illustrates an example of the system model containing two linked component models available in NRAP-Open-IAM: reservoir model and cemented wellbore model. Some of the components parameters are deterministic, some are stochastic, some are composite or linked to other parameters. Keyword arguments of the wellbore component (pressure and CO_2 saturation at the bottom of the well) are obtained from the observations of the reservoir component. Leakage rates of two fluids (CO_2 and brine) are calculated and shown on the produced figures. The name of the file containing the example script is *iam_sys_reservoir_cmwell.py* in the folder *examples/scripts*.

```
import sys,os
import matplotlib.pyplot as plt
import numpy as np

sys.path.insert(0,os.sep.join(['..','..','source']))
from openiam import SystemModel, SimpleReservoir, CementedWellbore

if __name__=='__main__':
    # Create system model
    num_years = 50.
    time_array = 365.25*np.arange(0.0, num_years+1)
    sm_model_kwargs = {'time_array': time_array}   # time is given in days
    sm = SystemModel(model_kwargs=sm_model_kwargs)

    # Add reservoir component
    sres = sm.add_component_model_object(SimpleReservoir(name='sres', parent=sm))

    # Add parameters of reservoir component model
    sres.add_par('numberOfShaleLayers', value=3, vary=False)
    sres.add_par('shale1Thickness', min=500.0, max=550., value=525.0)
    sres.add_par('shale2Thickness', min=450.0, max=500., value=475.0)
    sres.add_par('shale3Thickness', value=11.2, vary=False)
    sres.add_par('aquifer1Thickness', value=22.4, vary=False)
    sres.add_par('aquifer2Thickness', value=19.2, vary=False)
    sres.add_par('reservoirThickness', value=51.2, vary=False)

    # Add observations of reservoir component model
    sres.add_obs_to_be_linked('pressure')
    sres.add_obs_to_be_linked('CO2saturation')

    # Add cemented wellbore component
    cw = sm.add_component_model_object(CementedWellbore(name='cw', parent=sm))

    # Add parameters of cemented wellbore component
    cw.add_par('logWellPerm', min=-13.9, max=-12.0, value=-12.0)

    # Add keyword arguments of the cemented wellbore component model
    cw.add_kwarg_linked_to_obs('pressure', sres.linkobs['pressure'])
    cw.add_kwarg_linked_to_obs('CO2saturation', sres.linkobs['CO2saturation'])

    # Add composite parameters of cemented wellbore component
```

(continues on next page)

(continued from previous page)

```
    # Here, we illustrate two ways to define expressions for composite parameters
    # One way
    cw.add_composite_par('wellDepth', expr=sres.pars['shale1Thickness'].name+
        '+'+sres.pars['shale2Thickness'].name+
        '+'+sres.deterministic_pars['shale3Thickness'].name+
        '+'+sres.deterministic_pars['aquifer1Thickness'].name+
        '+'+sres.deterministic_pars['aquifer2Thickness'].name)
    # Second shorter (and more explicit) way
    cw.add_composite_par('depthRatio',
        expr='(sres.shale2Thickness+sres.shale3Thickness' +
        '+ sres.aquifer2Thickness + sres.aquifer1Thickness/2)/cw.wellDepth')
    cw.add_composite_par('initPressure',
        expr='sres.datumPressure + cw.wellDepth*cw.g*sres.brineDensity')

    # Add observations of the cemented wellbore component
    cw.add_obs('CO2_aquifer1')
    cw.add_obs('CO2_aquifer2')
    cw.add_obs('brine_aquifer1')
    cw.add_obs('brine_aquifer2')

    # Run forward simulation
    sm.forward()

    # Collect observations
    CO2_leakrates_aq1 = sm.collect_observations_as_time_series(cw, 'CO2_aquifer1')
    CO2_leakrates_aq2 = sm.collect_observations_as_time_series(cw, 'CO2_aquifer2')
    brine_leakrates_aq1 = sm.collect_observations_as_time_series(cw, 'brine_aquifer1')
    brine_leakrates_aq2 = sm.collect_observations_as_time_series(cw, 'brine_aquifer2')

    # Print results: CO2/brine leakage rates and pressure/saturation at the well
    print('CO2 leakage rates to aquifer 1:', CO2_leakrates_aq1, sep='\n')
    print('CO2 leakage rates to aquifer 2:', CO2_leakrates_aq2, sep='\n')
    print('Brine leakage rates to aquifer 1:', brine_leakrates_aq1, sep='\n')
    print('Brine leakage rates to aquifer 2:', brine_leakrates_aq2, sep='\n')

    # Plot CO2 and brine leakage rates along the wellbore
    plt.figure(1)
    plt.plot(sm.time_array/365.25, CO2_leakrates_aq1, color='blue',
             linewidth=2, label='aquifer 1')
    plt.plot(sm.time_array/365.25, CO2_leakrates_aq2, color='red',
             linewidth=2, label='aquifer 2')
    plt.ticklabel_format(style='sci', axis='y', scilimits=(0, 0))
    plt.legend()
    plt.xlabel('Time, t [years]')
    plt.ylabel('Leakage rates, q [kg/s]')
    plt.title(r'Leakage of CO$_2$ to aquifer 1 and aquifer 2')

    plt.figure(2)
    plt.plot(sm.time_array/365.25, brine_leakrates_aq1, color='blue',
             linewidth=2, label='aquifer 1')
    plt.plot(sm.time_array/365.25, brine_leakrates_aq2, color='red',
             linewidth=2, label='aquifer 2')
```

(continues on next page)

(continued from previous page)

```
    plt.ticklabel_format(style='sci', axis='y', scilimits=(0, 0))
    plt.legend()
    plt.xlabel('Time, t [years]')
    plt.ylabel('Leakage rates, q [kg/s]')
    plt.title('Leakage of brine to aquifer 1 and aquifer 2')
    plt.show()
```

BIBLIOGRAPHY

[1] D. R. Harp. Model analysis toolkit (MATK). URL: http://matk.lanl.gov.

[2] D. R. Harp, R. Pawar, J. W. Carey, and C. W. Gable. Reduced order models of transient CO2 and brine leakage along abandoned wellbores from geologic carbon sequestration reservoirs. *International Journal of Greenhouse Gas Control*, 45:150–162, 2016. URL: http://www.sciencedirect.com/science/article/pii/S1750583615301493, doi:10.1016/j.ijggc.2015.12.001.

[3] A. B. Jordan, P. H. Stauffer, D. Harp, J. W. Carey, and R. J. Pawar. A response surface model to predict CO2 and brine leakage along cemented wellbores. *International Journal of Greenhouse Gas Control*, 33:27–39, 2015. doi:10.1016/j.ijggc.2014.12.002.

[4] R. J. Pawar, G. S. Bromhal, S. Chu, R. M. Dilmore, C. M. Oldenburg, P. H. Stauffer, Y. Zheng, and G. D. Guthrie. The National Risk Assessment Partnership's integrated assessment model for carbon storage: a tool to support decision making amidst uncertainty. *International Journal of Greenhouse Gas Control*, 52:175–189, 2016.

[5] K. Reitz and T. Schlusser. *The Hitchhiker's Guide to Python*. O'Reilly Media, 1st edition edition, 2016.

[6] D. Rosenberg and K. Scott. *Use Case Driven Object Modeling with UML: A Practical Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-43289-7.

[7] P. H. Stauffer, S. Chu, E. H. Keating, G. N. Keating, J. W. Carey, H. S. Viswanathan, C. Tauxe, and R. J. Pawar. *CO2-PENS User's Guide*. 2015.

[8] G. Zyvoloski. FEHM: a control volume finite element code for simulating subsurface multi-phase multi-fluid heat and mass transfer. Technical Report, Los Alamos National Laboratory, Los Alamos, NM, 2007. URL: https://fehm.lanl.gov/pdfs/FEHM_LAUR-07-3359.pdf.

PYTHON MODULE INDEX

INDEX