1 Introduction

The Python Inverse Problem Toolbox (PIPT) is an inversion framework built on the simple fact that the main parts of most inversion algorithms are basically the same: run some kind of simulator (forward part) and compare the simulation results to actual observed data in some manner (inversion part). Implementing the forward modelling part can be a laborious experience, involving a lot of code for overhead and bookkeeping; especially since we in most cases do not have direct access to simulator source codes and thus rely on making input files and process output files in order to get our simulation results. The inversion part is usually easier to implement, but often we tend to implement our inversion codes with a specific project in mind, and thus we typically cut some corners here and there and hard code some parameters, just to make the code to run. So when you are finished with the project, the inversion code is usually worthless for the next project, and you basically have to start over, which takes a lot of unnecessary time!

In PIPT we have not implemented a wrapper to all the simulators in the world, nor have we implemented all the inversion schemes you could think of (or not think of). What we aim for with PIPT is to have a framework where new simulators and/or inversion update schemes can easily be implemented in a plug-in/module fashion. This way the user can just focus on implementing the new simulator or the new update scheme, and not worry if the whole inversion 'loop' will work or not. By having a systematic framework to relate to at all times, we think that the user will use much less time on the implementation side and also get an inversion code that works for the next project.

Not only do we aim to it easy to do the implementations, we also want the toolbox to be easy to use. Inversion usually involves running a lot of experiments with different parameters, setups, data, etc., or even different update algorithms and simulators. Hence, in PIPT we have tried to make an interface where the user can do all the changes he/she wants without going into the code at all. The user interacts with PIPT using an keyword-based input file. Together with a simple Python main-file (and, of course, all the other files needed in the specific inversion), the input file makes PIPT run the inversion. With a simple-to-use interface to the inversion code, we also hope a user not intimately familiar with inversion also can use PIPT. This way the toolbox can serve a purpose for other people than just the us inversion fanatics. Details on how the input file works, and different mandatory and optional keywords can be found in section (REF).

A little disclaimer at the end: we are definitely not programmers, so do not expect PIPT to be the ultimate Python code which works regardless of what you throw at it. Do not expect every error message you get to be meaningful. We have tried to put in custom-made error checks and messages, but not nearly enough to satisfy every possible combination of error-triggers (even though we have encountered an abundance of errors at this point...). We appreciate feedback if you find some bugs, and we will try to squash them!

2 Implementation style and code organization

PIPT is, for the most part, written in an object-oriented manner. We will not provide any tutorial on object-oriented programming (OOP) (tutorials can be found everywhere and will probably be better than any explanations that we can cook up), but we can say a few words of why and how we use OOP for this framework. OOP allows for the framework to consist of different pieces, organized in a hierarchical manner, that, when needed, can be put together to form a complete, runnable code. By using classes and methods with as general purpose as possible, we can put together an inversion 'loop' based on information given by the user in the input file.

The glue between the different pieces is class inheritance (and in some places, implicit function calls, but do not tell anyone!). The inheritance structure or hierarchy in PIPT is as follows: at the top is the class for reading the input file; classes for forward runs in ensemble/stochastic or deterministic/single-parameter-updating schemes; classes for different type of inversion 'loops' in either ensemble or deterministic schemes; and lastly, the inversion update schemes. To be able to put together, initialize, run the correct classes, a main module (file) `ip.py` has been made. It is this module, together with the input file, a PIPT inversion run is initiated.

Let's have brief look at these classes and the main module `ip.py`.

Reading the PIPT input file – `ReadInitFile`

The Allfather of all classes in PIPT is the class to for reading the input file – `ReadInitFile`. All of the keywords and information therein is read and parsed to Python dictionaries and variables to be used further down the hierarchy (see section REF for more details). Some general notes on how keywords are converted to Python variables are given in section (REF). Other than the parsing, nothing really exciting is done in this class.

Run simulations in ensemble-based or deterministic algorithms – `Ensemble` or `???`

The classes in this part of the hierarchy are crucial, and it is also here the user will plug in the simulator he/she want to implement. How to plug in your simulator with the classes here are described in section (REF), so we will focus on the other parts of these classes. The first thing in these classes are that the `ReadInitFile` class is inherited, just to be able to pipe through and do the parsing of the input file, and thereby have parsed information from the input file to work with. Using the input file information, a lot of initialization is done in these classes: organizing the observed data and data variance that have been provided, and also initializing the `simulator` class. For `Ensemble` an initial ensemble is also generated by some algorithm in the `geostat` package, based on prior information given by the user in the input file.

The meat of `Ensemble` or `???` is the forward simulations. These are done in `Ensemble.calc_forecast` and `???`, and is used in the next step of the hierarchy to get predicted data to be compared with the observed data. It is also here the methods that are required in the `simulator` class is located (see section (REF)). The forward simulation methods in `Ensemble` or `???` is setup to only run simulations that you need at the current step in the inversion to compare to the observed data. How to implement a new simulator in these classes are described in (REF).

A useful method in `Ensemble` is `Ensemble.calc_prediction`. In this method we can run simulations using the model parameters at different stages of the inversion, to, e.g., evaluate the quality of the inversion updates. It functions principally the same way as `Ensemble.calc_forecast`.

Inversion 'loops' – modules `ensemble_loops.py` and `deterministic_loops.py`

In the two modules (files), `ensemble_loops.py` and `deterministic_loops.py`, there are several classes involving the, for a lack of a better word, 'loop' of an inversion method. A 'loop' contains the general steps in the inversion, which are often repeated. An example, is the `ensemble_loops.Sequential` class where the recursive two-step procedure of forecast and analysis steps, used in methods like EnKF, is located. The classes in these modules acts as a link between the forward simulations done in `Ensemble` or `???` and the update schemes implemented in the `update_schemes` package (see next section). Since these classes act as a link, restrictions are made on name for the method that do the inversion in the updating schemes. (In other words, we implicitly call the update scheme methods that are lower in the hierarchy, and thus they need to be named the same as in the 'loop' classes.) In section (REF), we give a detailed overview the classes that have been implemented so far. We also provide some tips, if a user wants to implement a 'loop'.

Lastly, we mention that the method where the inversion 'loop' takes place need to be called `run_loop`, due to the call of this method in `ip.run_inversion` (see the next section).

Update schemes – package `update_schemes`

At the bottom of the PIPT hierarchy are the update schemes. These are classes located in some (user-made) module in the package `update_schemes`. An update-scheme class must be attached to (i.e. must inherit) the correct inversion 'loop', described in the previous section; this way a complete inversion algorithm is made by following the inheritance structure to the top. Some methods in the update-scheme class are mandatory, due to the inversion 'loop', but for the most part the content of an update-scheme class is dependent on the inversion algorithm. Many algorithms have been implement already, but the nice thing with PIPT is that the user can add his/her own algorithm! That way the library of inversion methods in PIPT will continue to grow as users add new algorithms.

The details of how to add an update scheme to PIPT is given in section (REF).

The main module (file) – `ip.py`

The initialization and running of the correct classes for an inversion or prediction run that the user have specified in the input file, we have made a main module (file) called `ip.py`. This module contains two methods, `run_inversion` and `run_predicition`. The `run_inversion` method takes the PIPT input file name and, optionally, a save name for the final results of the inversion. Based which update scheme chosen in the input file, the correct classes are linked inside `run_inversion`, before `run_loop` from the inversion 'loop' is called, which gets the inversion going. The inversion results are saved in the end.

In `run_predicition`, the `Ensemble` class is initialized, and `Ensemble.calc_prediction` is called. The PIPT input file name and a save name for the prediction results are the only inputs.

3 The input file

To interact with PIPT, we use an input file containing various information via different keywords. There is no required suffix for the input file, but we recommend using '.txt' (we have not tested if other suffixes will give Python trouble when reading the file). The keywords are organized in two parts: DATAASSIM and FWDSIM, with no restrictions on which part comes first. In the DATAASSIM part information on the inversion is given, while in the FWDSIM information passed to the simulator is given. We will go through, in detail, both of these parts in the sections below.

The keywords and information therein are parsed in the `ReadInitFile` class (see section (REF)), and are stored in two Python dictionaries: `self.keys_da` for keywords in DATAASSIM and `self.keys_fwd` for keywords in FWDSIM. The keys in these Python dictionaries have the same name as the keywords (in lower case), making it easy to extract information from the input file inside the PIPT code.

A general note on structure: we have no character to end a keyword. Instead we rely on lineshifts between each keyword. Here the user must be aware to have a lineshift at the end of the file (maybe two for good measure) such that the last keyword is read properly. The beginning of each part starts with either DATAASSIM or FWDSIM followed be a lineshift. So do not begin with a keyword immediately below the part name; remember the lineshift.

Comments may be written in the input file. These start with an octothorpe (#), and can be put everywhere **exept** at the end of lines containing keywords, parts, or keyword information. That is, you cannot end a line where you have written any input to PIPT with a comment. So write comments on separate lines. In `ReadInitFile`, we read all lines in the input file except those starting with a #.

Before we dive into the sea of keywords, some recommendations when making the input file is in place. First of all, use the most basic text editor you have to make the input file. There is no need for any advanced formatting, so do not use a more complicated text editor than necessary! Next tip: if you get any error in PIPT that you do not understand, particularly if it occurs in `ReadInitFile`, look over the input file for any unnecessary characters, whitespaces, etc. Especially, if you (for some reason) make the input file in a different OS than the one you are running PIPT in, you have to be aware of that OS' may use different character encodings. These errors are a pain in the butt to debug, so avoid at all costs! There is actually no requirement on lower or upper case inputs, but we recommend at least that keyword names and different options/inputs in keywords have upper case. Also the part names should be written in upper case to be more visible. At the end, we recommend looking at section (REF) to get a sense on how keywords in the input file are converted to Python variables.

3.1 DATAASSIM

Most of the keywords in DATAASSIM part are mandatory. This is because information entered in each keyword is processed by `ReadInitFile` and used to setup the inversion (forecast and analysis step). Hence, no new keywords must be added here without conferring with the authors. Also, do not change how the mandatory keywords are handled in the code, since this will most likely break the whole program!

Below, we will go through the mandatory and optional keywords in the DATAASSIM part that are currently implemented. Examples of how each keyword works will be given.

Mandatory keywords in DATAASSIM

The mandatory keywwords are essential for the forecast and analysis part to function correctly.

`DAALG` keyword

Choose between two sequential and simulations ensemble-based methods. For sequential methods enter ENKF; for the simulations method enter ES. If ENKF is chosen, the order in which the assimilation is done matters – see ASSIMINDEX (REF!) keyword for more on grouping the data in ENKF. If ES is chosen, all data is assimilated at once, hence order in ASSIMINDEX does not matter (while the indices entered, of course, matters).

Example of choosing ENKF (sequential) as data assimilation algorithm:

```
DAALG
ENKF
```

NE keyword

Insert the number of ensemble members. Must be an integer value!

Example with the use of 100 ensemble members:

```
NE
100
```

OBSNAME keyword

Enter the name of assimilation index where the observed data is been taken from. Traditionally, the assimilation index is some time step, hence the entry in this keyword could be DAYS, DATES, etc. For non-temporal assimilation, like, e.g., a geophysical application, the assimilation index can be source positions or frequencies, in which case the entry in this keyword could be SOURCE or FREQ, respectively. The name entered in this keyword, is given to the user again, in the forecast step, when setting up the forward run and when getting results from a successful forward run. For example, if a user wants both source positions (e.g., SOURCE) and frequencies (e.g., FREQ) to be the assimilation index for an application, the user must implemented in the geophysical simulator's (say seismic simulator's) setup- and getting-results-methods (see REF! for setting up a `simulator` class) such that the predicted data is associated with source positions (SOURCE) or frequencies (FREQ).

Example of using dates as the assimilation index:

```
OBSNAME
DATES
```

TRUEDATAINDEX keyword

Here, the actual value of the assimilation index is entered (whereas the name is entered in OBSNAME (REF!)). For example, if OBSNAME is DATES, the values in TRUEDATAINDEX should be a list of dates (in some format). As with OBSNAME entries, the entries in TRUEDATAINDEX is given to the user when setting up and getting results from the simulator in the forecast step (see REF! for setting up a simulator). The entries here can be a list written either in a single column or row.

Example where OBSNAME is DATES (see example in REF!), and we list some dates in TRUEDATAINDEX (if pure values are entered here, it would not matter if each value is separated by a tab, indicated by the '→ |' symbol, or not – see REF!):

```
TRUEDATAINDEX
1 JAN 2015      →|1 FEB 2015      →|1 MAR 2015
```

TRUEDATA keyword

The observed data to be used in the inversion/data assimilation is entered in this keyword. A row corresponds to the data at the same entry in TRUEDATAINDEX (see REF!); and a column correspond to the same entry in DATATYPE (in FWDSIM, see REF). For example, an entry in row 2 and column 3 corresponds to the observed data for assimilation index given by entry number 2 in TRUEDATAINDEX and data type given by the entry number 3 in DATATYPE. Hence, the number of rows must be the same as number of entries in TRUEDATAINDEX, and the number of columns must be the same as the number of entries in DATATYPE.

An entry can be a single number input or, if the observed data for that entry is a vector, it can be given as Numpy save file (.npz file). The name of the .npz file must be correct according to where it is placed! If there is no data at a particular row and column, `N/A` must be given!

In some cases, there can be a lot of rows and columns in TRUEDATA. To make the input file comprehensible to read and make changes in, a .csv file (Comma Separated Value file) can be given. If a .csv file is given, it should be the only entry in TRUEDATA! A .csv file can be easily generate using your favorite spreadsheet program (like, Exc... I mean LibreOffice Calc :), or using the csv package for Python. The entries in the .csv file follow the same rules as if they shold have been entered directly in TRUEDATA.

Example with 2 data types and 3 assimilation index values, combining scalar, .npz file and `N/A` inputs (the values and strings are separated by a tab, indicated by the '→ |' symbol. If only scalars are inputed, the values do not need to be separated by a tab – see REF!):

```
TRUEDATA
101.8       grid_data0.npz
75.7        grid_data1.npz
111.1       N/A
```

Example with a .csv file (here, called 'true_data.csv') input:

```
TRUEDATA
true_data.csv
```

ASSIMINDEX keyword

In this keyword, the order of assimilation is determined. The entries are Python indices of the list entered in TRUEDATAINDEX, that is, the entries in ASSIMINDEX range from 0 to number of entries in TRUEDATAINDEX. For example, if 2 is given in ASSIMINDEX, this corresponds to the 3. – third – entry in TRUEDATAINDEX (remember – Python indexing!).Said in other words, the entries in ASSIMINDEX are used to pick out the correct assimilation index given in TRUEDATAINDEX, which again are linked with the corresponding observed data given in TRUEDATA (see REF!). So, by giving an entry in ASSIMINDEX, we choose what observed data we will assimilate at that assimilation step! Since this is the case, ASSIMINDEX entries are combined with TRUEDATAINDEX entries to setup forward simulation run such that we only make predicted data that corresponds with the relevant observed data at an assimilation step (see REF! for how to setup a simulator).

The actual order of assimilation is determined by entries at each row below ASSIMINDEX. Furthermore, the number of rows determined the number of assimilation steps! The order of assimilation only makes sense if ENKF (sequential ensemble method) is chosen in DAALG! If ES (simultaneous ensemble method) is chosen in DAALG, all data is assimilated at once, and there is only one assimilation step. Hence, if ENKF is chosen, the observed data can be grouped together at each assimilation step by entering multiple indices at each row under ASSIMINDEX. As an example of this, say there are five dates (here, DATES in OBSNAME) in TRUEDATAINDEX, and we want to assimilate observed data corresponding to these dates in two assimilation steps, where we group the data corresponding to the first three dates in the first assimilation step and the rest in the second assimilation step, the relevant keywords are then:

```
DAALG
ENKF

OBSNAME
DATES

TRUEDATAINDEX
1/1/2015        1/2/2015        1/3/2015        1/4/2015        1/5/2015

ASSIMINDEX
0 1 2
3 4
```

If we, in the same example, wanted to assimilate one observed data correspond to one date at each assimilation step, we only need to change ASSIMINDEX (the other keyword are the same as the example above!):

```
ASSIMINDEX
0
1
2
3
4
```

If we, again in the same example, want to assimilate the observed data in reversed chronological order (i.e., starting with the last date and ending with the first date in TRUEDATAINDEX), we, again, only change ASSIMINDEX:

```
ASSIMINDEX
4
3
2
1
0
```

Hence, we are able to change how we assimilate data by just changing ASSIMINDEX and keeping the other relevant keywords as they were.

Note that if ES is chosen as DAALG, the indices in ASSIMINDEX must be in a single column (as in the two last examples above). If they are not, an error will occur! The order, however, does not matter, since with ES we assimilate all observed data in a single assimilation step. But the indices entered in ASSIMINDEX matters with ES as DAALG, since these are associated with entries in TRUEDATAINDEX and indicate which observed data to assimilate.

DATAVAR keyword

Entries in this keyword determine the variance of the observed data. The structure of this keyword follows the same rules as for TRUEDATA (see REF!); that is, the columns corresponds to a DATATYPE entries (see REF!), and the rows corresponds to TRUEDATAINDEX entries. The only exceptions is that one entry in DATAVAR consist of two inputs: a string choosing variance type and a value determining the variance, which must be separated by a tab. Each entry must also be separated by a tab (so, every entry in DATAVAR is separeted by a tab).

If all the entries in DATAVAR are the same, it is possible to only give a single entry and the Python code will copy it to the rest of the entries. It is also possible to give one row of entries and the Python code will copy this to the remaining rows.

In the same way that TRUEDATA can get a large number of entries when the number of DATATYPE and TRUEDATAINDEX entries are high, DATAVAR gets twice as much entries. Hence, it is possible to give a .csv file with all the entries, in the same way as for TRUEDATA (see REF!).

There are two options for the data variance: absolute and relative. For absolute variance, ABS is the string input and a value giving the absolute variance of that data type is given in the second input. For relative variance, REL is given as the string input and a value corresponding to the percentage of the observed data is given in the second input. Note that the percent value is the standard deviation which in turn is squared to give the variance!

As an example, we give some dummy observed data in TRUEDATA which we give some variance using both absolute and relative variance using DATAVAR:

```
TRUEDATA
110        →|125
100        →|38

DATAVAR
ABS        →|10        →|REL        →|5
REL        →|10        →|ABS        →|1
```

Here, the variance for the second entry in the first row of TRUEDATA will be $(125 * 0.05)^2 \approx 39$. In the same manner, the variance for the first entry in the second row of TRUEDATA will be $(100 * 0.1)^2 = 100$. The remaining entries have absolute variance which is given directly.

If we want all the observed data to have the same variance, say an absolute value of 10, we change DATAVAR to:

```
DATAVAR
ABS        →|10
```

If we want the two data types (the two columns in TRUEDATA) to have different variance, but the same for each TREUDATAINDEX (each row), say an absolute value of 10 for the first data type and relative value of 5 for the second, DATAVAR will change to:

```
DATAVAR
ABS        →|10        →|REL        →|5
```

Lastly, if we want to give the entries in DATAVAR as a .csv file, say 'data_var.csv', DATAVAR will be given as:

```
DATAVAR
data_var.csv
```

FREESTATE keyword

In this keyword, the static and/or dynamic variables to be inverted is entered. The names given here will be passed through to the simulator in the forecast step, when setting up the forward simulation run (see REF! on how to set up a simulator).

An example of two static parameter typically inverted in a reservoir history matching case, permeability and porosity, can be given as:

```
FREESTATE
PERMX
PORO
```

Note that the strings can also be on one row separated by a tab.

PRIOR keyword(s)

To make the initial ensemble, information regarding the prior distribution must be given in this keyword. The initial ensemble is generated using analytical covariance function, which is only dependent on the spatial distance. Various covariance functions have been implemented: spherical, exponential, and cubic. More functions can easily be added (just ask :). Currently, the initial ensemble is generated using Cholesky decomposition, which is not ideal for larger problems (>10 000, or so). Faster methods will be implemented in the future. The PRIOR keyword handles 2D and layered 2D structures (not general 3D at the moment).

First and foremost, one has to make one PRIOR keword for each entry in FREESTATE, where the name of the keyword must be PRIOR_<entry in FREESTATE> (see the example below). An error is given if there is not a PRIOR keyward for each entry in FREESTATE.

The setup of the entries in each of the PRIOR_< ... > keyword follow the same structure: First column is an option command followed by value or string inputs in the following columns. There may be multiple columns after the first option column if the state variable is a layered 2D structure (inputted in the GRID option, see below). Recall that, multiple columns of string inputs have to be separated by a tab, while value inputs can be separated by white space. If there is a layered 2D structure, but only a single column has been entered after the option, this information will be copied to the other layeres. Hence, it is not necessary to repeat the same information for all of the layeres.

The options for the PRIOR_< ... > keyword can be entered in any order, and the mandatory options are as follows:

GRID: Number of grid cells in x- and y-direction, and number of layers (may be ommitted for 2D grid)

VARIO: Variogram model.

MEAN: If array, a string with Numpy save file must be entered. If scalar, just enter the value. Note that, if layered 2D is chosen, it is assumed that the mean array is organized such that the mean for each layer is augmented below each other. Hence, the first $nx \times ny$ entries in the mean array belongs to the first layer, the next $nx \times ny$ entries belong to the second layer, etc.

VAR: Sill level of variogram

RANGE: Correlation distance (typically)

ANISO: Anisotropic factor. A value below 1, indicates a squeeze of the x-axis (with no rotation) by the factor. Set to 1 for isotropic covariance

ANGLE: Rotation out from the y-axis. Omitted if ANISO is 1.

A single optional command is:

LIMITS : Truncation values for prior realizations. Here, to values must be given *per* layer!

As an example, we want to make an initial ensemble of 100 member of the permeability and porosity we gave in FREESTATE (see example in REF!), where for the permeability want make an isotropic model using the spherical covariance function, and for the porosity want make an anisotropic model using the exponential covariance function. Moreover, we want to have 10 grid cells in both x- and y-direction, and 2 layers, where we want the same covariance model for both layes for PERMX, but different models for PORO. The relevant keywords are given as:

```
NE
100

FREESTATE
PERMX
PORO

PRIOR_PERMX
VARIO       sph
```

```
MEAN        mean_permx.npz
VAR         10
RANGE       5
ANISO       1
ANGLE       0
GRID        10 10 2

PRIOR_PORO
VARIO       exp         sph
MEAN        mean_poro.npz
VAR         0.5 0.25
RANGE       5 10
ANISO       0.5 0.25
ANGLE       90
GRID        10 10 2
LIMITS      0.9 0.1
```

First we note that the mean values for PERMX and PORO has been given as a .npz file, since we have a grid of 10×10. We also see that for PERMX we have isotropic prior because the anisotropy factor is 1, and for PORO we have anisotropic prior since the anisotropy factor is 0.5 (also anisotropy angle is 90°). Furthermore, the initial ensemble members for PORO have an upper limit of 0.9 and lower limit of 0.1. Note that if an option in PRIOR_PORO has not been written for each layer, then the first values is just copied (see, e.g., ANGLE option for PRIOR_PORO).

Optional keywords in DATAASSIM

Some optional keywords exists in DATAASSIM.

TEMPSAVE keyword

This keyword is used to save the ensemble of state variables initially (i.e. the ensemble made using the information in PRIOR) and after each assimilation step. The ensemble of state variables is stored in a Numpy save file (.npz) named 'Temp_state_assim.npz'. Here, the `state` dictionary is stored in a list of length equal to no. of assimilation steps + 1 (due to the storing of the initial ensemble).

The input in `TEMPSAVE` is simply `YES` or `NO`.

Example:

```
TEMPSAVE
YES
```

ANALYSISDEBUG keyword

Sometimes when, e.g., debugging, it can be advantageous to look at variables in the analysis step. This can be done with the ANALYSISDEBUG keyword. The entries are strings containing the analysis step variable names separated by tabs. The variables are stored for each analysis step in a Numpy save file named 'debug_analysis_step_<assimilation step>.npz'.

An example storing some of the variables listed above:

```
ANALYSISDEBUG
COV_DATA        COV_CROSS       PRED_DATA       KALMAN_GAIN
```

OBSVARSAVE keyword

Option to store the observed data and data variance as a Numpy save (.npz) file called 'obs_var.npz'. The observed data is just the values inputted in `TRUEDATA` organized in a dictionary called `obs` where the keys are the data types inputted in `DATATYPE`. The data variance is stored in a dictionary called `var` and is organized in the same manner as `obs`.

The input in `OBSVARSAVE` is `YES` or `NO`.

Example:

```
OBSVARSAVE
YES
```

3.2 FWDSIM

In the FWDSIM part, most of the keywords will be provided by the user as he/she implements the forward simulator to be used for a specific application. Hence, most of this section will give information on how keywords are converted to an input dictionary, which is a mandatory input when implementing a simulator. General instructions on how a simulator must be implemented to work with the Ensemble class, will also be given. But first we describe the few mandatory keywords in FWDSIM.

Mandatory keywords in FWDSIM

The mandatory keywords in FWDSIM are few, mostly giving the information on where the simulator is located and how it should run.

SIMULATOR keyword

Determines the simulator to be used. This keyword is important to write correctly! The first entry must correspond to *module* name (or file name), and the second entry must correspond to the *class* name of the simulator (the module must be located in the package/folder named *simulator* – see REF!). We need to be able to called the simulator as simulator.<entry 1>.<entry 2> in Python! If the name is not entered correctly, the simulator cannot be initialized, and no forward simulations can be done in the forecast step.

Example with the use of a simulator called MARE2DEM which is located in the CSEM module:

```
SIMULATOR
CSEM        →MARE2DEM
```

PARALLEL keyword

Enter the number of forward simulations to run in parallel. This is typically limited by the computational resources on the computer/cluster the simulations are run on. But in principle, the number of parallel runs can be equal to the number of ensemble members (NE keyword in DATAASSIM).

Example of 8 parallel runs:

```
PARALLEL
8
```

DATATYPE keyword

In this keyword, the data types of the observed data is entered. An example of a data type is well bottom hole pressure for fluid flow simulations, or real part of an electric field component in CSEM. The number of entries in this keyword must correspond to the number of columns in TRUEDATA and DATAVAR. Most importantly, entries in DATATYPE will be passed through to the simulator when getting results from a forward simulation run in the forecast step. Hence, the names of the data types given must be properly chosen by the user such that they can be used to get the correct predicted data in the data assimilation.

An entry in DATATYPE is a string with the name of the data type. The multiple names can be entered in separate rows, or on one row separated by a tab.

Example of DATATYPE in a reservoir history matching application. The data types are water saturation and well injection rate for a well arbitrarily named 'I01':

```
DATATYPE
SWAT
WWIR I01
```

4 Setup of a simulator for ensemble runs

The whole idea with PIPT is that it can be used with any simulator (of any application) as long as the simulator is connected to the class in the correct manner. All that is needed is to write a Python class for the simulator with a few mandatory methods (functions) to work with the forecast step of the `Ensemble` class. In general, simulators are separate programs, like commercial simulators, e.g., Eclipse or STARS (see step 2 REF! below if the simulator is implemented in Python directly), hence the methods in the PIPT `simulator` class are just wrappers around the external simulator program. Specifically, there are five methods that *needs* to be implemented in the `simulator` class, and they consist of the following steps:

1. Setup of the simulator
2. Run the forward simulation for each ensemble member as a background application
3. Check for errors in the runs
4. Check if simulations is finished
5. Get results from a simulation

These five methods *must* have a specific name, and we shall go through each method in the sections below. Note here that the `simulator` class can (and often must) include various other methods to work, e.g., to generate input files for the simulator, do some calculations that are required before entering it to the simulator, etc.. We also encourage the user to implement the simulator in such a way that it can be used outside the PIPT framework. This way other users may run one-off simulation experiments without initializing the whole PIPT framework.

Before we embark on the task of setting up the `simulator` class, we mention that the `simulator` class *must* be located in the `simulator` package (or folder), with a module and class name equal to the first and second entry, respectively, in SIMULATOR keyword (see REF!).

Disclaimer: It is assumed that the user has some general knowledge about Python, and what a class is and how it is structured in Python.

Authors' remark: When we go into the definitions of the Python methods below, the input variable *names* do not need to be the same as have been written here. However, the inputs are provided by the `calc_forecast` method in the `Ensemble` class and thus you need to make sure that the *positional* and *optional arguments* in the different methods follow the description that is written below. The same goes for the outputs expected by `calc_forecast`: you can change the variable names internally in the `simulator` class as long as `calc_forecast` gets the expected output! This is very important: make sure that the inputs and outputs from the simulator are the same as we explain below; if the variable names are not the same, that is OK.

4.1 Initializing the `simulator` class and extracting information from FWDSIM keywords

Before we describe the five methods needed in the `simulator` class, we have to describe how a `simulator` class have to be initialized in order to work with the `Ensemble` class. When the `Ensemble` class initialize the simulator to be used in the data assimilation, it passes all the keywords in FWDSIM as dictionary called `input_dict`. The dictionary `input_dict` have keys with the same name as the keywords in FWDSIM in *lower case*, and the content of each keyword is organized as explained in (REF!). Hence, `input_dict` *must* be the first input in the `simulator` class! The header of a generic `simulator` class typically looks as follows:

```
class <simulator_name>:
  """ Some info. on simualtor """

  def __init__(self, input_dict):
    # Input
    self.input_dict = input_dict

    # Extract sim. var. from input_dict
    self._ext_info_input_dict()
```

To extract the information we have written in the FWDSIM part and passed through via `input_dict`, we use a method called `_ext_info_input_dict` (not a mandatory name or method, but somewhere in `__init__` you should internalize the information in `input_dict`). This is a simple method for extracting the necessary variables to run the simulator from the FWDSIM keywords. What is important to keep in mind when making `_ext_info_input_dict` is that the strings and numbers given in FWDSIM are not converted to Numpy arrays in the `Ensemble` class. Hence, if the simulator have variables that is required to be Numpy arrays, you must convert the variables in `input_dict` to the correct form in this

method! Note also that if you want to make a Numpy array of some number that can either be a scalar *or* an array, a check for that keyword being a scalar or a list must be done. If not, the Numpy array of the single number is not accessed in the same manner as a standard numpy array (of a list of numbers):

```
# Scalar
a = np.array(1)
a[0]
# IndexError: too many indices for array

# Scalar in a list
a = np.array([1])
a[0]
1
```

A generic `_ext_info_input_dict` method typically looks like this:

```
def _ext_info_input_dict(self):
  # KEYWORD1 and KEYWORD2 are single inputs
  self.variable1 = int(self.input_dict['keyword1'])  # Integer
  self.variable2 = float(self.input_dict['keyword2'])  # Float

  # KEYWORD3 could be a single input or a list
  if isinstance(self.input_dict['keyword3'], list):  # KEYWORD3 is a list of  numbers
    self.array1 = np.array(self.input_dict['keyword3'])
  else:  # KEYWORD3 is a single number
    self.array1 = np.array([self.input_dict['keyword3']])
```

By conferring to (REF!) the user can make keywords in FWDSIM and know exactly how to extract it in `_ext_info_input_dict`. After the header of the `simulator` class is made, we can now move on to the five mandatory methods needed in the `simulator` class.

4.2 Step 1: Setup of the simulator – `setup_fwd_sim`

The first step when running the forecast step in the `Ensemble` class is to set up the simulator such that we can get the predicted data that is needed in the current assimilation step (and only that). Hence, the information that is put into to the simulator in this step is: information about the state variable and information of how long (for temporal applications) or what (for non-temporal problems) we want to simulate in the current forecast step.

The definition of the Python method for this step has to be given in this manner:

```
def setup_fwd_sim(self, state, assim_ind, true_ind):
```

We first note that the whole `state` is inputted. This is a dictionary where the keys are the entries in FREESTATE (see REF!) and the content of each key is an ensemble array with no. or row equal the length of the state variable (for that key) and no. of columns equal the no. of ensemble members (determined in NE – see REF!). The second input, `assim_ind`, is a Python list providing the information from the ASSIMINDEX (REF) keyword on the indices from TRUEDATAINDEX (REF) to run at the current forecast stage, and also the observed data name in OBSNAME (REF). Specifically, the first entry in `assim_ind` is the name in OBSNAME, and the second entry is the ASSIMINDEX entries (either a single float or a sublist of floats). The third input, `true_ind`, is a similar Python list as `assim_ind` but the second entry is *all* information entered in TRUEDATAINDEX. Combining information from `assim_ind` and `true_ind` should be enough to run temporal and non-temporal simulators to provide the predicted data needed in the current assimilation step.

An example of `assim_ind` and `true_ind`, together with relevant keyword info, is given below. Note that we are on the second assimilation step, hence only information from the second line in ASSIMINDEX is relevant:

Input file:

```
OBSNAME
DATES

TRUEDATAINDEX
1 JAN 2000       ↪1 FEB 2000       ↪1 MAR 2000       ↪1 APR 2000       ↪1 MAY 2000

ASSIMINDEX
```

```
0 1
2 3 4
```

Python output:

```
assim_ind = ['dates', [2.0, 3.0, 4.0]]
true_ind = ['dates',['1 jan 2000', '1 feb 2000', '1 mar 2000', '1 apr 2000', '1 may 2000']]
```

An illustrative example on how simple `setup_fwd_sim` can look like follows below. We assume that `assim_ind` and `true_ind` contains the information in the example above, and furthermore that `state` is a dictionary containing values on a generic model parameter called PARAM1 in FREESTATE:

```
def setup_fwd_sim(self, state, assim_ind, true_ind):
  # Extract the model parameter that is inverted from the state dictionary
  self.inv_param1 = state['param1']

  # Combine assim_ind and true_ind to get information on how long the simualtion will run.
  if assim_ind[0] == 'dates':
    self.run_dates = true_ind[1][assim_ind]  # = ['1 mar 2000', '1 apr 2000', '1 may 2000']
```

4.3 Step 2: Run the simulator – `run_fwd_sim`

After the simulator is set up to run the forecast step, it is time to actually get going. In this step, everything that is required for the simulator to actually run, must be executed; before we at the end of this step press GO on the simulator. Specifics on what must be done in this step is difficult, since each simulator have different working procedures and requires different things to run. Since it possible to implement a simulator directly in the `simulator` class (do this only for small/simple simulators!), we will give recommendations/typical outline/tips on what needs to be executed in this step separately if the simulator is external or programmed directly.

The definition of the Python method is as follows:

```
def run_fwd_sim(self, en_member=None, folder=os.getcwd(), wait_for_proc=False):
```

The first thing we note here is that all of the inputs seem to be optional (they have default values associated with them). This is due to the idea that the same method could be used in a deterministic forward/forecast simulation, where some of the inputs are not required or must be changed to fit a single forward simulation. Also, it should be possible to use the same method *outside* PIPT, e.g. to just run the simulator for testing stuff or using the simulator to make synthetic data.

To run the forecast step in the `Ensemble` class, the two first input arguments *must* be in those exact positions. This is because the `calc_forecast` method in the `Ensemble` class treats `run_fwd_sim` as having two positional inputs, instead of optional [1]. The first input, `en_member`, provides which ensemble member in the state/inversion parameter the simulator is going to run. Recall that static and/or dynamic variables have been internalized in Step 1, and each one is an ensemble matrix. Hence, use `en_member` is used to pick out the correct member to be used in the simulation run.

The second input, `folder`, tells the simulator in which folder should the simulation be run. For an ensemble forecast run, each member will be run in separate subfolders of the main project folder the user is running his/her experiment, and name of that subfolder is given in `folder`. Hence, everything that is needed to run the simulator, e.g. input files, pre-stored variables, etc., must be copied to or, if files are generated in the simulation class, stored in `folder`. Also, when finally running the simulator, use `folder` to tell the simulator where to run the simulation. **Do not** 'go' into `folder` and start the simulation that way! This will almost surely end in an error! Do everything while 'standing' in the main project folder, and give the path to `folder` to the simulator when it is time to run the simulation (see example below for recommended way to organize `run_fwd_sim`).

A word about the third input, `wait_for_proc`, which is treated by the forecast step as having the default value – `False`. If you are going to run the simulator outside the PIPT framework, it is not necessary to run it as background process as we do in the `Ensemble` forecast step (also not necessary in deterministic inversion methods). Hence, `wait_for_proc` is an input that can be used to switch on and off running the simulation in the background or not. For the `Ensemble` forecast step the user is *required* to implement the simulator in a way such that it is possible to run it in the background! If not, the user will do a whole lot of waiting during the inversion! See the examples below for a way to implement the calling of the simulator with the `wait_for_proc` switch.

[1] If we in `calc_forecast` instead had treated the inputs as optional, the names of these variables would needed to be fixed. Hence, to give the user the freedom to use whatever variable name he/she wants internally in the `simulator` class, we treat the inputs as positional.

There are probably a lot of ways to call a simulator, whether it is an external program or have been implemented directly in the `simulator` class. A way that we suggest to do it is to make a separate method, which we typically call `call_sim`, and execute the simulator there. To call external simulator programs, we like to use some methods from the `subprocess` package, which is included in Python. To call a simulator which is directly programmed in the `simulator` class, we recommend that the user makes a method where the calculations (setting up and crunching of the linear system(s)) is done, such that we in `call_sim` can call that method with the `multiprocessing` package, also included in Python.

External simulator:

```
# Imports at the top of the file where the simulator class is located
from subprocess import call  # call simulator and wait for it to be finished
from subprocess import Popen  # call simulator and run it in the background

def call_sim(self, folder=None, wait_for_proc=False):
  # Generate path names to where the run files are located. Assume below that some kind of
  # simulation input file has been made and its name is stored in self.sim_file
  if folder is not None:  # e.g. in ensemble forecast runs
    filename = folder + self.sim_file

  else:  # files are located in project folder
    filename = self.sim_file

  # Run the simulator as a background process or not. Inside call or Popen, you write the same
  # as you would have written in a shell command to call the simualtor. See documentation for
  # call and Popen to correctly input the shell command
  if wait_for_proc is True:  # do not run in background
    call(['<simualtor name>', '<options>', filename])

  else:  # run in background
    Popen(['<simualtor name>', '<options>', filename])
```

Internal simulator (written directly in `simulator` class):
The simulator method here is called `calc_solution` and has input `folder` such that it is able to store the end result in `folder`:

```
# Import at the top of the file where the simulator class is located
from multiprocessing import Process  # run in background or not by commands in Process

def call_sim(self, folder=None, wait_for_proc=False):
  # The folder input is perhaps more seldom here as everything is typically stored
  # in some variable. If there are some external files it is possible to link the path
  # the same way as for an external simulator (see code in the above example!).

  # Initialize the Process class to run the simulator method calc_solution.
  # See documentation on Process for more information on inputs and usage.
  sim = Process(target=self.calc_solution, args=(folder,))

  # Start simualtion
  sim.start()

  # Run in background or not
  if wait_for_proc is True:  # do not run in background
    sim.join()
  else:  # run in background
    pass  # this 'else' is unnecessary, but is written to illustrate the point
```

To exemplify the whole `run_fwd_sim` method, it is assumed that the information provided to the simulator is the same as we provided in the example in Step 1 (REF); a state variable internalized as `self.inv_param1` and dates to run as `self.run_dates`. The simulator is an external program that needs an input file, so we assume that this is made in a method called `make_sim_input_file`, which is located somewhere in the same `simulator` class.

```
def run_fwd_sim(self, en_member=None, folder=os.getcwd(), wait_for_proc=False):
  # Check if the inv_param1 is an ensemble matrix or not (ensemble or deterministic inversion)
  if self.inv_param1.ndim > 1:  # ensemble matrix
    self.run_param1 = self.inv_param1[:, en_member]  # stored parameter for this run

  else:  # single array (deterministic inversion)
```

```
    self.run_param1 = self.inv_param1

  # Make the simulator input file into the folder where the simuation is to be run.
  # Ensemble forecast: subfolder named 'En_<en_member>'
  # Single simulation: current project folder.
  self.make_sim_input_file(folder)

  # Call the simulator
  self.call_sim(folder, wait_for_proc)
```

4.4 Step 3: Check for errors in the simulations – `check_sim_error`

4.5 Step 4: Check if simulations are finished – `check_sim_end`

Since we in the **`Ensemble`** forecast run the simulations in the background, we need to continuously check in the ensemble member's run folders to see if the simulation is finished. Typically, when a simulator is done (especially commercial simulators), they give an output file of some sort where the simulation results are located. Hence, in this step it can be easy to just look for this output file. If the user has made the simulator him-/herself either as an external program or programmed it directly in the **`simulator`** class, it is important that the results are stored inside the run folder as some kind of file (a Numpy '.npz' file perhaps?). Then it is easy to search for this file in this step.

The definition of the Python method for this step is as follows:

```
def check_sim_end(self, current_run):
```

The method only has one input, **`current_run`**, which provides a list of the ensemble members that are currently running in the forecast step. Hence, in this step it is easy to just loop over all ensemble members in **`current_run`** and check the 'En_<ensemble member>' folder for any output file from the simulation.The output required in this method is **`member`** (or whatever name you want to give it), which contains the index of the of the finished ensemble member. **NOTE:** **`member`** must be given property **`None`** if no results are ready! The forecast code check if **`member`** is different from **`None`** when it is extracting the results (next step).

An example of how **`check_sim_end`** typically looks like follows below. We assume that the simulator provided an '.out' (made up for the example) file when finished, and we search for this in each folder. Checking for specific files in a folder can probably be implemented much more efficiently than what is done below, but it serves well for illustrating the point of the method:

```
def check_sim_end(self, current_run):
  # Initialize member as None
  member = None

  # Loop over the ensemble members in current_run
  find = False
  for i in current_run:
    # Loop over the files in subfolder of an ensemble member
    for files in os.listdir('En_' + str(i)):
      # Check if a .out file is in folder
      if files.endswith('.out'):
        member = i
        find = True
        break

    # If member is found, break loop
    if find is True:
      break

  # Return member (either an index or None)
  return member
```

4.6 Step 5: Get results from finished simulation – `get_sim_results`

The crucial last step in the **`Ensemble`** forecast run is to extract results from a simulation that is finished. This step is very specific to what simulator that has been used, hence it is difficult to give exact description of what should be done

in here. Nevertheless, we will provide the details on what the inputs and the outputs is expected by the Ensemeble forecast, and give a simple example at the end of the section. Again, we encourage the user to implement this method to work outside the PIPT framework. A method that extracts results from a simulator is always useful!

The Python method for this step is:

```
def get_sim_results(self, which_resp, data_info, member=None):
```

The first input, `which_resp`, is the data type of the simulation result to be extracted, which is one of the entries in the keyword DATATYPE (REF). Next input, `data_info`, contains information on where or at place the simulation result should be extracted from. Specifically, `data_info` is a list with OBSNAME as first entry and the TRUEDATAINDEX associated with the index of the current assimilation step (given in ASSIMINDEX) in the second entry. Note here that if ASSIMINDEX contains more than one index for the assimilation step, the `Ensemble` forecast method loops over each index one at the time, and provides the information for one index in the second entry of `data_info` (see example below). The last input, `member`, is an index of the ensemble member that is finished with the simulation. This can be used to know in which folder the result file is located. Note that it is an optional keyword, and this is because if the method is used outside the `Ensemble` forecast step for a single simulation run, `member` is not needed.

If the output from the simulation is some kind of output file(s), `get_sim_results` typically contains method(s) for reading or loading the file(s). In addition, if the simulator does not directly give you the results in the manner you want or have defined in `which_resp`, some kind of calculations are also done in `get_sim_results`. Whatever needs to be done to get the correct the results from the simulator, it has to be explicitly returned to the `Ensemble` forecast method. Note that the responses have to be organized as a 1D array. A tip for the user here: Make sure that the results from the simulator is organized in the same manner as the observed data! If not, unpleasant inversion results may ensue, which can be hard to debug (trust me...).

To give a simple example of `get_sim_results`, we assume that OBSNAME, ASSIMINDEX, and TRUEDATAINDEX is given as in the example in Step 1 (REF), and we are again at the second assimilation step to extract simulation results from ASSIMINDEX 3 for member indexed 40. The DATATYPE we want at the moment is RATIO. The results from the simulation is given in a file named 'sim_results.out' (recall from Step 4 that we searched for a '.out' file), and we use a method (located somewhere in the `simulator` class) to extract what we need. However, the results from the simulator needs to be converted to some other form (calculate some kind of ratio) before we return it to the `Ensemble` forecast method:

```
def get_sim_results(self, which_resp, data_info, member=None):
  # Extract results from simulation by reading the .out file. If it is an ensemble run,
  # the file is located in an 'En_<member> folder, else we assume it is in the current
  # folder.
  # We use the information in data_info to know exactly where to extract the results from.
  # For this example data_info = ['dates', '1 apr 2000'].
  if member is not None:  # ensemble run
    out_sim = self.read_out_file('En_' + str(member) + os.sep + 'sim_results.out', data_info)
  else:  # single simulation
    out_sim = self.read_out_file('sim_results.out', data_info)

  # If the data type is 'ratio', we calculate a ratio from the results; else we give
  # out the simulation results wholesale.
  # For this example which_resp = 'ratio'.
  if which_resp == 'ratio':
     # The ratio is the first half of out_sim divided by the second half
     out_sim_split = np.split(out_sim, 2)
     output_resp = out_sim_split[0]/out_sim_split[1]

  else:
    # Whole array with simulation results will be returned
    output_resp = out_sim

  # Return the simulation result the way we request through which_resp
  return output_resp
```

5 Keywords to Python variables

When adding a new application, new keywords in FWDSIM part are typically made to be able to pass variables and parameters to added simulator. How strings and values defined within a keyword are converted to strings and values in

Python is crucial knowledge. Hence, below we will show numerous examples of how keyword information is organized in Python. In the examples, a tabular space in the input text file (where keywords are defined) is indicated by a '→ |' symbol. Note that all strings will be stored in lower case in Python! Also, every keyword in the FWDSIM part passed to the simulator in a dictionary called `input_dict` with `'keys'` equal to the keyword names. Lastly, values are converted to floats when read in Python. Hence, if integer values are needed, a conversion must be done when extracting information from `input_dict`

Single string and value input

Single string or number inputs will be stored in the same way they are inputted.

Input file:

```
KEYWORD1
STRING1

KEYWORD2
1
```

Python:

```
input_dict['keyword1'] = 'string1'
input_dict['keyword2'] = 1
```

Multiple strings and values in a single row

Adding multiple strings in a single row will be stored in Python as a list of strings. The important thing here is that each string must be separated by a tab. Spaces between strings will be gathered in a single string in Python. For value inputs, it does not matter if the spaces between them are single spaces or tabs (OBS: this is not true when we combine strings and values, see below!).

Input file:

```
KEYWORD1
STRING1       →|STRING2       →|STRING3  STRING4

KEYWORD2
1 2       →|3 4 5       →|6       →|7
```

Python:

```
input_dict['keyword1'] = ['string1', 'string2', 'string3  string4']
input_dict['keyword2'] = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
```

Single string and value in multiple rows

Single string or value in multiple rows will be stored in a list with entry equal to what is written at each row.

Input file:

```
KEYWORD1
STRING1
STRING2

KEYWORD2
1
2
```

Python:

```
input_dict['keyword1'] = ['string1', 'string2']
input_dict['keyword2'] = [1.0, 2.0]
```

Multiple strings and values in multiple rows

Having multiple strings or values in multiple rows just combines all the variations in Python we have discussed above, BUT now each row is stored in SEPARATE lists within the 'main' list for the whole keyword. In other words, the inputs are stored in Python as 2D lists. Note that the list structure in Python is flexible, so number of columns in each rows may vary.

Input file:

```
KEYWORD1
STRING1       →STRING2
STRING3

KEYWORD2
1 2 3
4 5
```

Python:

```
input_dict['keyword1'] = [['string1', 'string2'], ['string3']]
input_dict['keyword2'] = [[1.0, 2.0, 3.0], [4.0, 5.0]]
```

Combining strings and values

Combining strings and values is a bit more complex. The main rule is that a string and a value needs to be separated by a tab, else they will be interpreted as a single string. Also, the string and value will be put in a list (when separated by correctly by a tab).

Input file:

```
KEYWORD1
STRING1  1

KEYWORD2
STRING2       →2
```

Python

```
input_dict['keyword1'] = 'string1  1'
input_dict['keyword2'] = ['string2', 2.0]
```

If multiple values are inputted after the string, the values will be put in a separate list. OBS: If the values are separated by a tab, this will generate a new list with the values following the tab (see example KEYWORD2)

Input file:

```
KEYWORD1
STRING1       →1 2 3 4

KEYWORD2
STRING2       →1 2 3       →4 5 6
```

Python:

```
input_dict['keyword1'] = ['string1', [1.0, 2.0, 3.0, 4.0]]
input_dict['keyword2'] = ['string2', [1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]
```

Combining strings and values in multiple rows will by converted in Python using the same rules as mentioned above (see 'Multiple strings and values in multiple rows' section)

Input file

```
KEYWORD1
STRING1       →1 2
STRING2       →1 2 3       →4 5 6

KEYWORD2
```

```
STRING3       ↦1 2       ↦3 4       ↦STRING4       ↦1
STRING5       ↦10 11 12
```

Python:

```
input_dict['keyword1'] = [['string1', [1.0, 2.0]],
                          ['string2', [1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]]
input_dict['keyword2'] = [['string3', [1.0, 2.0], [3.0, 4.0], 'string4', 1.0],
                          ['string5', [10.0, 11.0, 12.0]]],
```

6 Implementing a new update scheme

An advantage with PIPT is the possibility to easily implement new inversion update schemes. Many of the general features of most inversion schemes have been already been implemented in PIPT (REF). Hence, the user can just attach his/her update scheme to the appropriate parts of PIPT and everything is set to go (we do not promise this for all algorithms; some of them may need some more implementation up the code chain). New update schemes must be put in the package (folder) `update_schemes`, and added to an appropriate module (file). The last part is not a strict rule, you can make a separate module with your update scheme, but to keep things tidy and somewhat cataloged, we appreciate if update schemes with common structure are located in the same module (file).

Similar to implementing a new simulator in PIPT, a new update scheme has to be a class with name equal to what is written in DAALG (REF) in *lower* case. The class must inherit one of the classes in the `ensemble_loops.py` or `deterministic_loops.py`; see section (REF) for descriptions of available 'loops'. If none of the 'loops' are appropriate, the user can implement a new, see section (REF).

The goal of all update schemes is to update the (unknown) state variable (dynamic or static, or both) with some kind of algorithm. Hence, in every method where the actual updating is done, the `self.state` dictionary must be updated. Somewhere in the update method (usually the very last thing that is done) `self.state` must be updated *directly*, by replacing the old values with updated ones. It is `self.state` that is used in the next forward simulation, if there inversion 'loop' is actually a loop (some algorithms do one forward simualtion and one update, but regardless you must update `self.state`). If you want to save previous instances of `self.state` use the TEMPSAVE keyword (REF)!

Regardless of update scheme to be implemented, the `__init__` method is mandatory with input `input_file`. `input_file` is the name of the PIPT input file coming from `ip.py` (REF), which must be passed to the parent class. Of course, other initializations can be done in `__init__`, just as in a normal class. A generic update-scheme class may look like:

```
class <update_scheme_name>(<inversion_loop>):
  """ Some info. on the algorithm """

  def __init__(self, input_file):
    # Pass the PIPT input file upwards in the hierarchy
    super().__init__(input_file)

    # Some internal initialization
    self.var1 = None
```

The other methods besides `__init__` that are mandatory, is dependent on which inversion 'loop' class the user inherits. The user is referred to section (REF) if he/she does not know which 'loop' to attach the update scheme to. If the user do not find an appropriate 'loop' in section (REF), he/she can implement his/her own, and we refer to the tips given in section (REF). Below we will go through the mandatory methods for each inversion 'loop' described in section (REF).

To debug an implementation of an update algorithm, or to extract some of the variables in the code for future evaluations, we give a short code in the last section that can be copy-pasted at the end of the update method and that is attached to the ANALYSISDEBUG keyword (REF). Even though it is optional to implement this code, we highly recommend it, both for the user's and future users' sake. It does, however, require that variables that are in the ANALYSISDEBUG list to be local variables, i.e., without the `self` prefix.

Before we start on the description of the various methods, we remind the user on the **Authors' remark** in section (REF): the name of the input and output is not so important; as long as the input/output is correct, name them what you like. However, for future reading and understanding of the code, please be descriptive! And use English!

6.1 Mandatory methods for the `ensemble_loops.py` module

Below are sections with the mandatory methods for classes in the `ensemble_loops.py` module. We also mention that some useful tools often used in ensemble-based algorithms have been implemented in the `analysis_tools` under the `misc_tools` package. Take a look there if you are saying to yourself 'this is so general it must have been used in some of the other update schemes'. If it is not implemented there, you are free to implement it yourself. The more reusable code, the better!

Mandatory method for `Sequential` and `Simultaneous`

There is only one mandatory method for `Sequential` and `Simultaneous`, and we lump both of these 'loops' together since `Simultaneous` is just one step of the 'loop' in `Sequential`. The mandatory method must be called `calc_analysis` and it has one input: `assim_step`. As the name implies, `assim_step` gives the current assimilation step, which is an integer from 0 to N_s, where N_s is the total number of assimilation steps. An assimilation step is associated with the correspond row in ASSIMINDEX, and thus, in combination with ASSIMINDEX, can give us information on which observed data, data variance, etc. to use in the update.

A generic, and simple, example of `calc_analysis` follows below. We use `assim_step` and information from ASSIMINDEX stored in `self.keys_da['assimindex']` to get some arbitrary data. Also, we assume that the algorithm requires some covariance matrices for updating the state. Lastly, we update the state variable with a generic/arbitrary method, which we will not define.

```
def calc_analysis(self, assim_step):
  # Get current assimilation indices
  assim_ind = [self.keys_da['obsname'], self.keys_da['assimindex'][assim_step]]

  # Extract the inputted observed data vector and data variance
  obs_data_vector = self.obs_data[assim_ind[1]]['data1']
  data_var = self.datavar[assim_ind[1]]['data1']

  # Generate ensemble of realizations of the observed data
  obs_data = self.gen_obs_data(obs_data_vector, data_var)

  # Extarct predicted data (fwd. sim. results)
  pred_data = self.pred_data[assim_ind[1]]['data1']

  # Calculate covariance matrices using state and predicted data
  cov = self.calc_cov(self.state, pred_data)

  # Update the state variable (remark here that self.state is input and output,
  # signalling that we update self.state directly)
  self.state = self.update_state(self.state, obs_data, data_var, pred_data, cov)
```

For a more elaborate update scheme see, e.g., the `enkf` class in the `seq_ensemble.py` module.

Mandatory methods for `Iteration`

The mandatory methods for `Iteration` are: `calc_analysis` and `check_convergence`. The `calc_analysis` is almost exactly the same as for the `Sequential` and `Simultaneous` 'loops', but the input is now `iteration`. As the name implies, `iteration` is an integer with the current iteration number. This may be used, for example, if something special is suppose to be done on the first iteration (= 0), like skipping evaluations on changing damping parameter(s), etc.. Extracting which data to assimilate in an `Iteration` 'loop' is easy. We do not have an `assim_step` input now, but we do not need it either since the `Iteration` 'loop' is initialized as for a `Simultaneous` algorithm, i.e., all indices in ASSIMINDEX (or, in other words, all observed data) is used at once. Hence, we can extract all indices in ASSIMINDEX via `self.keys_da['assimindex'][0]` (this is ensured in `__init__` of `Iteration`). If you do this, you can use the tools provided in `analysis_tools`, or other codes you or others may have developed for a `Sequential` or `Simultaneous` method. (As long as you extract all observed data and compare it correctly with the forward simulations, other procedures than what we describe above can be used.)

The output from `calc_analysis` is `success_iter`. This ensures that only the iteration counter in `Iteration` are only updated if there is a successful iteration. In addition, temporary results (TEMPSAVE keyword) will only be saved during successful iterations.

The `check_convergence` method has not input. Its purpose is to do an evaluation of convergence, and provide information whether to stop or keep going with the iterations, based on the outcome of the convergence check. The evaluations can

be done on variables stored in `calc_analysis` (via `self`) or by calculating some measures in `check_convergence` itself, or a combination. The outputs of `check_convergence` are: `conv` and `why_stop`. `conv` is just a logical operator (`True` or `False`) telling is the iterations have converged or not (if `True`, `Iteration` 'loop' stops). `why_stop` is a possibility for the user to define why the iterations have stopped. This can be, e.g., a Python dictionary with information on which convergence evaluation that was fulfilled and what value it had, and/or what the values of the other evaluations were; it can also just be a string telling the user which evaluation that was fulfilled; etc. Ultimately, `why_stop` is a tool for the user to get useful information on the convergence check, which is saved at the end of `Iteration`.

Before we give an example, we note that it is possible to provide information on parameters and tolerance vales used in the update-scheme class using the ITERATION keyword (REF). Everything given in ITERATION is stored in `self.keys_da['iteration']`, which can be used in `__init__` method to extract all the information we need. This makes it very simple to change various static parameters that typically is a part of an iterative algorithm, like, e.g., damping parameters, convergence tolerances, etc.

A simple example of how `calc_analysis` and `check_convergence` can look like is given below. We assume that the algorithm here requires some kind of gradient and Hessian to calculate the step direction. As in the example of `calc_analysis` in the `Sequential` and `Simultaneous` 'loops' given above, we will not provide detail on any algorithm or method in `calc_analysis` and `check_convergence` (everything is made up anyway...). We also provide a simple loop in `__init__` to extract some of the static parameters in the iteration algorithm from `self.keys_da['iteration']` (ITERATION keyword) (damping parameter and convergence tolerances). We also check if the iteration was successful by evaluating the distance between predicted and observed data (very simple here, just for illustrative purpose).

```
import numpy as np  # for check_convergence

                                                    .........

def __init__(self, input_file):
    # Pass the PIPT input file upwards in the hierarchy
    super().__init__(input_file)

    # Loop over entries in ITERATION and extract parameters needed in
    # the algorithm. If the parameters are not given in ITERATION, we
    # use a default value.
    for i, opt in enumerate(list(zip(*self.keys_da['iteration']))[0]):
      if opt == 'damp_param':
        self.damp_param = self.keys_da['iteration'][i][1]
      else:  # default value
        self.damp_param = 100
      if opt == 'tol_step':
        self.tol_step = self.keys_da['iteration'][i][1]
      else:  # default value
        self.tol_step = 1e-3
      if opt == 'tol_grad':
        self.tol_grad = self.keys_da['iteration'][i][1]
      else:  # default value
        self.tol_grad = 1e-2

def calc_analysis(self, iteration):
  # Get current assimilation indices
  assim_ind = [self.keys_da['obsname'], self.keys_da['assimindex'][0]]

  # Extract the inputted observed data vector and data variance
  obs_data_vector = self.obs_data[assim_ind[1]]['data1']
  data_var = self.datavar[assim_ind[1]]['data1']

  # Generate ensemble of realizations of the observed data
  obs_data = self.gen_obs_data(obs_data_vector, data_var)

  # Extarct predicted data (fwd. sim. results)
  pred_data = self.pred_data[assim_ind[1]]['data1']

  # Calculate gradient and Hessian
  self.grad = self.calc_gradient()
  self.hessian = self.hessian()

  # Evaluate distance between pred_data and obs_data
  misfit = np.norm(pred_data - obs_data)
```

```
    # Calculate a damping parameter if iteration is not 0. If misfit has
    # decreased from previous iteration, we decrease damping;
    # else increase.
    # In addition, if misfit decrease we save the misfit for use in the
    # next iteration and set success_iter = True.
    if iteration > 0:
      if misfit < self.prev_misfit
        # Damping decreased
        self.damp_param = self.damp_param/2

        # Save misfit for next iteration
        self.prev_misfit = misfit

        # Successful iteration
        success_iter = True

      else:
        # Damping increased
        self.damp_param = self.damp_param/2

        # Successful iteration
        success_iter = False

    # Calculate the step direction
    self.step = self.calc_step(self.grad, self.hessian, obs_data, pred_data,
                               self.damp_param)

    # Update the state variable with the calculated step direction
    self.state = self.state + self.step

    # Return success_iter
    return success_iter

  def check_convergence(self):
    # We evaluate the gradient and step direction calculated in calc_analysis.
    if np.linalg.norm(self.step) < self.tol_step or \
        np.linalg.norm(self.grad) < self.tol_grad:  # CONVERGENCE!
      # Convergence reached and we add information on which criteria that was met
      why_stop = {'step_stop': np.linalg.norm(self.step) < self.tol_step,
                  'step_val': self.step,
                  'grad_stop': np.linalg.norm(self.grad) < self.tol_grad,
                  'grad_val': self.grad}
      conv = True

    else:  # no convergence...
      # Convergence has not been reached, and we leave why_stop empty
      why_stop = {}
      conv = False

    # Return the outputs
    return conv, why_stop
```

For a more elaborate iterative update scheme see, e.g., the `lm_enrml` class in the `iter_ensemble.py` module.

Mandatory methods for `Mda`

The `Mda` 'loop' is similar to a `Sequential` 'loop'. With predefined number of assimilations/iterations, the only mandatory method is `calc_analysis`. The only input here is `mda_step`, which is **not** the same input as in the `calc_analysis` method in a `Sequential` update scheme! A `Mda` algorithm is not sequential; the assimilation indices in ASSIMINDEX are used all at once as in an `Iteration` or `Simultaneous` update scheme. The `mda_step` is mainly used to regulate the inflation parameter that is a part of a `Mda` update scheme. Other than this, the content of `calc_analysis` is typically similar to a `Simultaneous` or `Iteration` method.

Information about the inflation parameter to be used the update scheme can be extracted in the `__init__` method

from the MDA keyword via `self.keys_da['mda']`. This way the user can change the inflation parameter easily without mucking around in the code.

A simple example of an update scheme using the `Mda` 'loop' is given below. We assume that the inflation parameter have been given in the MDA keyword, extract it in the `__init__` method.

```
def __init__(self, input_file):
    # Pass the PIPT input file upwards in the hierarchy
    super().__init__(input_file)

    # Extract inflation parameter from the MDA keyword, where we have called
    # the option INFLATION_PARAM (there may be other options in MDA so we
    # use a loop in a list comprehension to extract the parameter). If the
    # inflation parameter is not given in MDA, we use a default value
    # equal to the total number of mda steps (stored in self in the Mda
    # class)
    if 'inflation_param' in list(zip(*self.keys_da['mda']))[0]:
      self.inf_param = [item[1] for item in self.keys_da['mda']
                        if item[0] == 'inflation_param'][0]
    else:
      self.inf_param = [self.tot_assim]*self.tot_assim

def calc_analysis(self, mda_step):
  # Get current assimilation indices
  assim_ind = [self.keys_da['obsname'], self.keys_da['assimindex'][0]]

  # Extract the inputted observed data vector and data variance
  obs_data_vector = self.obs_data[assim_ind[1]]['data1']
  data_var = self.datavar[assim_ind[1]]['data1']

  # Generate ensemble of realizations of the observed data,
  # where the data variance is inflated
  obs_data = self.gen_obs_data(obs_data_vector, data_var, self.inf_param[mda_step])

  # Extarct predicted data (fwd. sim. results)
  pred_data = self.pred_data[assim_ind[1]]['data1']

  # Calculate covariance matrices using state and predicted data
  cov = self.calc_cov(self.state, pred_data)

  # Update the state variable (remark here that self.state is input and output,
  # signalling that we update self.state directly). The inflation parameter is
  # also part of the update.
  self.state = self.update_state(self.state, obs_data, data_var, pred_data, cov,
                                 self.inf_param[mda_step])
```

For a more elaborate MDA update scheme see, e.g., the `es_mda` class in the `iter_ensemble.py` module.

6.2 Mandatory methods for the `deterministic_loops.py` module

6.3 Implementing ANALYSISDEBUG in an update scheme