README

September 21, 2021

1 ParamGen Manual (draft)

Alper Altuntas, NCAR
Boulder, CO - 2021

1.1 1. Introduction

ParamGen is a lightweight, generic Python module for generating runtime parameters for scientific modeling applications. The module supports arbitrary Python expressions for the specification of parameter values. This provides a high level of flexibility and genericity.

ParamGen infers the values of model parameters from inclusive sets of *default parameters databases* (DPD) to be put together and maintained by the model developers. These databases are typically stored in a file written in a markup language such as yaml or json. ParamGen is generic, i.e., it is agnostic of any details of a particular modeling framework, model component, or input/output format. By default, the base ParamGen class supports yaml and json as DPD (input) format. The only out-of-the-box output format, on the other hand, is the Fortran namelist format. New input and output formats can easily be introduced by the developers via class inheritance as will be discussed in this document.

The primary property of a ParamGen instance is its `.data` member, which is of type Python dictionary, i.e., a collection of key-value pairs. When a ParamGen instance gets created, a dictionary must be provided to the ParamGen constructor to be accepted as its initial `.data`. This initial dictionary corresponds to the DPD, which may be read from yaml, json, etc.

In the simplest case, the keys correspond to parameter names and the values correspond to parameter values. In a more involved case, the `.data` member may be formed as a nested dictionary for grouping model parameters into seperate namelist modules. Moreover, the keys of the `.data` member may consist of logical expressions, i.e., *guards*. The notion of guards is one of the most important concepts in ParamGen. A *guard* is a proposition of a parameter value (similar to how guards are propositions of commands in Dijkstra's Guarded Command Language). Take the following data, for instance:

```
NIGLOBAL:
    $OCN_GRID == "gx1v6":
       320
    $OCN_GRID == "tx0.66v1":
       540
...
```

In the above nested dictionary, `NIGLOBAL` is interpreted as one of the parameter names. Within the inner dictionary, however, we have two keys, both of which are logical expressions. These logical expressions, or guards, are regarded as propositions of the values following them. After the instantiation, the `.reduce()` method may be called to evaluate the guards and determine the values of model parameters. In the above example, assuming the expandable variable `OCN_GRID` is `"tx0.66v1"`, calling the reduce method results in:

```
NIGLOBAL:
    540
...
```

Finally, the `.write()` method may be called to write the set of parameters in a desired format.

Note: not sure about the "default parameters database" (DPD) term. Any alternative suggestions welcome. -aa

1.2 2. ParamGen in Action

Obtaining ParamGen Although ParamGen is model-agnostic, it is currently distributed within an experimental CESM fork. To obtain this CESM version, run the following commands:

```
git clone https://github.com/alperaltuntas/CESM.git -b paramGenBeta
(cd CESM; ./manage_externals/checkout_externals -o)
```

In the above CESM sandbox, ParamGen is located in `CESM/cime/scripts/lib/CIME/ParamGen`

Importing ParamGen class The first step of working with ParamGen is to import the module. To import this experimental version of ParamGen module:

[1]:
```
# import MOM_RPS
from paramgen import ParamGen
```

Note: In the case of a CESM model, ParamGen would be imported from a buildnml script. To do so, one would first append the ParamGen directory to the PATH. See `CESM/components/mom/cime_config/buildnml` as an example.

Instantiating a ParamGen object: ParamGen constructor expects a `data` argument, a Python dictionary which may be nested or not. This dictionary corresponds to the default parameters database (DPD) that is the collection of parameter name-value pairs for all possible configurations. Let's first define a simple Python dictionary containing three variables `X`, `Y`, and `Z`:

[2]:
```
DPD_dict = {"X" : 1.0,
            "Y" : True,
            "Z" : "foo" }
```

Now, create a ParamGen instance with this DPD dictionary:

[3]:
```
pg = ParamGen(DPD_dict)
```

[4]:
```
pg.data
```

[4]:
```
{'X': 1.0, 'Y': True, 'Z': 'foo'}
```

We can now call the reduce method to generate the final version of `.data`:

[5]:
```
pg.reduce()
```

[6]:
```
pg.data
```

[6]:
```
{'X': 1.0, 'Y': True, 'Z': 'foo'}
```

As expected, the reduced data is not any different from the initial data we passed to ParamGen constructor. The `.reduce()` method makes a difference only when the initial data contains conditionals, variable expansion, or Python expressions. Before describing these mechanisms, let's generate the same ParamGen instance via yaml and json formats:

Instantiating a ParamGen object via yaml or json: ParamGen can be instantiated via yaml or json files using the following methods:

- `.from_yaml()`
- `.from_json()`

Under the hood, these methods simply create a Python dictionary from files with these formats and then call the ParamGen constructor with the generated dictionary.

yaml

[7]:
```
%%writefile DPD.yaml
X: 1.0
Y: True
Z: foo
```

```
Overwriting DPD.yaml
```

[8]:
```
# (2) Create a ParamGen instance:
pg = ParamGen.from_yaml("DPD.yaml")
pg.data
```

[8]:
```
{'X': 1.0, 'Y': True, 'Z': 'foo'}
```

json

[9]:
```
%%writefile DPD.json
{
    "X": 1.0,
    "Y": true,
    "Z": "foo"
}
```

```
Overwriting DPD.json
```

[10]:
```
# (2) Create a ParamGen instance:
pg = ParamGen.from_json("DPD.json")
```

```
pg.data
```

```
[10]: {'X': 1.0, 'Y': True, 'Z': 'foo'}
```

Note that an xml method is also planned to be added soon. Out of the three, yaml has the most readible and concise syntax, especially when working with large number of parameters and nested entries. The disadvantage of yaml is that it is not distributed with the standard Python, unlike xml and json. So a third party yaml parser, e.g., PyYAML, is required.

Instead of using these file formats, we will continue to create ParamGen instances using Python dictionaries explicitly in the remainder of this documentation. Recall that ParamGen converts these formats to a dictionary before creating an instance so the instructions below apply to all ParamGen instances regardless of which DPD format is used.

1.3 ParamGen Mechanisms

- Variable Expansion
- Guards
- Formulas
- Appending

1.3.1 Variable expansion

Similar to shell parameter expansion mechanism in Linux, The `$` character may be used to introduce expandable variables in DPDs. These variables are expanded, i.e., replaced with their values, when the `.reduce()` method is called. Variable expansion may be employed in both keys and values of DPD dictionaries. To illustrate this mechanism, we define a new ParamGen instance:

```
[11]: pg = ParamGen({
          "${alpha}": 1.0,
          "Y": "$beta",
          "$gamma": "foo"
      })
```

```
[12]: pg.data
```

```
[12]: {'${alpha}': 1.0, 'Y': '$beta', '$gamma': 'foo'}
```

In the above ParamGen instantiation, we specify three expandable variables in keys and values: `alpha`, `beta`, `gamma`. Now, let's reduce the data:

```
[13]: pg.reduce()
```

```
[14]: pg.data
```

```
[14]: {'${alpha}': 1.0, 'Y': '$beta', '$gamma': 'foo'}
```

The reduced data is the same as the initial data. This is because we didn't tell ParamGen how to expand those variables. This is simply done by passing a function to the `.reduce()` method. This function is required to take a string as an argument and return a scalar, i.e., a string, integer, float, or a boolean variable. The passed string corresponds to the variable name, while the return value corresponds the value of the expandable variable.

[15]:
```
def expand_func(varname):
    if varname == "alpha":
        return "X"
    elif varname == "beta":
        return True
    elif varname == "gamma":
        return "Z"
    else:
        raise RuntimeError("Unknown variable")
```

Before we can call the reduce method again, we need to reset the ParamGen instance, i.e., undo the `reduce()` method.

[16]:
```
pg.reset()
```

Now call the reduce method again, this time with the `expand_func` defined above:

[17]:
```
pg.reduce(expand_func)
```

[18]:
```
pg.data
```

[18]:
```
{'X': 1.0, 'Y': 'True', '"Z"': 'foo'}
```

As seen above, all the expandable variables are expanded, i.e., replaced with their respective values. Notice that variable `beta` is converted from bool to string during variable expansion. The same behavior applies to numeric variables as well. However, this behavior is not restrictive because (1) all values are converted to strings before they are written out to text files anyways and (2) all logical expressions and formulas are to be strings to be evaluated.

Warning: Specifying an expandable variable with or without curly braces doesn't make a difference unless the variable is a string. However, if the variable is of type string, its value is enclosed by quotes when the variable is specified *without* curly braces. This behavior can be observed with the variable `gamma` which expands to `'"Z"'`. Had we specified gamma with curly braces, i.e., `${gamma}`, the value would rather be `'Z'`, and not `'"Z"'`. This can be confirmed with the variable `alpha` above, which expands to `'X'`.

This behavior is introduced in ParamGen as a means of keeping conditional expressions in YAML more concise. Compare the following two logical ParamGen expressions, which are equivalent. In the first version, not only do we have to explicitly enclose expandable params with quotes (`"${...}"`), but also the entire expression (`'...'`) so as to make sure that YAML parser treats the entire logical formula as a single expression. In the second version, neither of the quotes is necessary, except, of course, for the literal strings `"gx1v7"` and `"datm"`.

```
' "${OCN_GRID}" == "gx1v7" and "${COMP_ATM}" == "datm" ':
```

```
$OCN_GRID == "gx1v7" and $COMP_ATM == "datm":
```

CESM/CIME XML variables as expandable variables Within the CESM framework, CIME XML variables may easily be inserted to DBDs as expandable variables. Typically,`ParamGen` is utilized in `buildnml` scripts of components. The first argument of the `buildnml` method is the `case` variable which is an instance of `CIME.case.Case`. This CIME case object has a `.get_value()` method that returns the value of a given XML variable. This method may simply be passed to the `reduce()` method of ParamGen as an expand function:

```
pg.reduce(lambda varname: case.get_value(varname))
```

Examples of this usage can be found in MOM6 implementation of ParamGen. Check out the following derived ParamGen classes of MOM6:

```
- CESM/components/mom/cime_config/MOM_RPS/FType_input_nml.py
- CESM/components/mom/cime_config/MOM_RPS/FType_MOM_params.py
```

1.3.2 Guards

Recall that the keys of the `.data` dictionary specify the parameter names while the values correspond to the respective parameter values. Depending on the context, the keys may also be interpreted as guards, i.e., propositions of parameter values. The keys are interpreted as guards if all the keys at a certain level are logical expressions that evaluate to True or False. A data dictionary *without* any guards:

[19]:
```
dict1 = {
    "var1": 1,
    "var2": 2
}
```

A data dictionary with some guards:

[20]:
```
dict2 = {
    "var1": {
        True: 1,
        False: 0
    },
    "var2": 2
}
```

In the above dictionary `dict2`, the variable `var1` has two options, `1` and `0`. Which value gets picked for "var1" depends on the guards, i.e., the propositions `True` and `False`. Now let's create a ParamGen instance with the dictionary `dict2`:

[21]:
```
pg = ParamGen(dict2)
pg.data
```

[21]:
```
{'var1': {True: 1, False: 0}, 'var2': 2}
```

Observe the effect of calling the reduce method below:

[22]:
```
pg.reduce()
pg.data
```

[22]: {'var1': 1, 'var2': 2}

Logical Python expressions as guards The guards above are trivially specified to be `True` and `False`. In practice, however, we usually have guards that are strings of Python expressions which may involve expandable parameters, comparisons, and/or boolean operations. For a dictionary key of type string to be regarded as a guard, it must: - be one of the following keywords: `True`, `False`, `else`
or - be a comparison operation, e.g., `"${OCN_GRID}=='gx1v7'"`, `"$OCN_NCPL > 1"`, `"$x in [a,b,c]"`, etc.
or - be a logical operation, e.g., `"$X and $Y"`, or `"$X or $Y"`
or - a combination of comparison and/or logical operations.

Note: In YAML, the quotes enclosing the expressions are not necessary, since the YAML parser automatically interprets those logical expressions as strings.

The following is an example with Python expressions as guards:

[23]:
```
def expand_func(varname):
    if varname == "one":
        return 1.0
    elif varname == "two":
        return 2.0
    else:
        raise RuntimeError("Unknown variable")


dict3 = {
    "var1": {
        '$one < $two' : 1,
        '$one > $two' : 0
    },
    "var2": 2
}

pg = ParamGen(dict3)
pg.reduce(expand_func)
pg.data
```

[23]: {'var1': 1, 'var2': 2}

Guard behavior

- If multiple guards evaluate to True, the last option gets picked. If it is desired to pick the first valid option, however, the default behavior may be changed by setting the optional `match` argument of ParamGen to `first`. For example: `pg = ParamGen(dict2, match='first')`
- If no guards evaluate to True, the parameter value gets set to `None`. In a model-specific write method, the parameters with the value `None` may be chosen to be omitted.
- the `else` keyword evaluates to True only if all other guards evaluate to False.

- When an expandable variable is attempted to be expanded, and if the value is undefined, ParamGen throws an error. In some cases, certain expandable variables may be defined only for certain configurations. For instance, in the below example, the variable `INIT_LAYERS_FROM_Z_FILE` is defined only if the `OCN_GRID` is one of `["gx1v6", "tx0.66v1", "tx0.25v1"]`. Therefore, to avoid undefined expandable variable error, we place the `INIT_LAYERS_FROM_Z_FILE` check below the `OCN_GRID` check, as follows:

```
tempsalt:
    $OCN_GRID in ["gx1v6", "tx0.66v1", "tx0.25v1"]:
        $INIT_LAYERS_FROM_Z_FILE == "True":
            "${INPUTDIR}/${TEMP_SALT_Z_INIT_FILE}"
```

This ensures that ParamGen attempts to expand `INIT_LAYERS_FROM_Z_FILE` variable only when `$OCN_GRID in ["gx1v6", "tx0.66v1", "tx0.25v1"]` evaluates to True.

1.4 3. Special Use Cases

1.4.1 Referencing across multiple ParamGen instances

The genericity that comes with the custom expand functions allows us to reference the data of a ParamGen instance in another ParamGen instance. To illustrate this use case, we define two ParamGen instances:

[24]:
```
pg1 = ParamGen({
    'var1' : 'foo',
    'var2' : 'bar'
})

pg2 = ParamGen({
    'var3': '${var1}${var2}'
})
```

Notice above that the second ParamGen instance, pg2, data includes references to variables defined in pg1 data. Now let's reduce the data of pg2 and pass a lambda function that returns the values of pg1 variables:

[25]:
```
pg2.reduce(lambda varname: pg1.data[varname])
```

[26]:
```
pg2.data
```

[26]:
```
{'var3': 'foobar'}
```

Note: Cross-referencing, i.e., references to variables within the same instance, is not supported. (May be added later on if need be. -aa)

1.4.2 Custom value inference

More involved expand functions may allow higher customizations. In the below example, for instance, we read in an xarray dataset `ds` and set the value of an expandable parameter `my_fields_list` to the list of all variables in `ds`, that is `"lat air lon time"`.

[27]:
```
pg1 = ParamGen({
    'var1' : 'foo',
    'var2' : 'bar'
})

pg2 = ParamGen({
    'param1': '${var1}${var2}',
    'param2': '$my_fields_list'
})

def expand_function(varname):
    if varname in pg1.data:
        return pg1.data[varname]
    elif varname == "my_fields_list":
        try:
            import xarray as xr
            ds = xr.tutorial.load_dataset("air_temperature")
            return ' '.join([var for var in ds.variables])
        except:
            print("Cannot load xarray module. Skipping...")
            return None
```

[28]:
```
pg2.reduce(expand_function)
```

[29]:
```
pg2.data
```

[29]:
```
{'param1': 'foobar', 'param2': '"lat air lon time"'}
```

1.5 4. Notes on ParamGen for MOM6 in CESM

Here, we briefly describe how ParamGen is used within CESM to generate MOM6 runtime parameters. Called from the `buildnml` script of MOM6+CESM, the ParamGen module is used to generate the four main MOM6 runtime input files:

1. **MOM_input:** Default MOM6 runtime parameters. The file syntax is based on the simple `key = value` pair. Example parameter entries from a typical MOM6 experiment:

```
DIABATIC_FIRST = True   ! If true, apply diabatic and thermodynamic processes...
DT_THERM = 3600.0       ! The thermodynamic and tracer advection time step.
MIN_SALINITY = 0.0.     ! The minimum value of salinity when BOUND_SALINITY=True.
```

2. **MOM_override:** An auxiliary file to override parameter values set in MOM_input
3. **input.nml:** A file to set some general MOM6 and FMS variables. The file is in classical Fortran namelist format.
4. **diag_table:** An input file to configure the model diagnostics. It has a relatively complex syntax. See https://mom6.readthedocs.io/en/latest/api/generated/pages/Diagnostics.html

In addition to these files, ParamGen is used to generate the MOM6 version of the `input_data_list` file needed by CIME.

Default Parameters Databases

For each of the input files mentioned above, we have a DBD that includes all of the default parameter values for any possible model configuration. In the case of `MOM_input` for instance, we have a DBD called `MOM_input.yaml` located in `components/mom/param_templates`. An example entry from this file:

```
DT_THERM:
   description: |
       "[s] default = 3600.0
       The thermodynamic and tracer advection time step.
   value:
       $OCN_GRID == "MISOMIP": 1800.0
       else: >
           = ( ( $NCPL_BASE_PERIOD =="decade") * 86400.0 * 3650 +
               ( $NCPL_BASE_PERIOD =="year") * 86400.0 * 365 +
               ( $NCPL_BASE_PERIOD =="day") * 86400.0 +
               ( $NCPL_BASE_PERIOD =="hour") * 3600.0 ) / $OCN_NCPL
```

In the above entry, the default value of the runtime parameter `DT_THERM` is specified, which depends on a few CIME variables such as `OCN_GRID`, and `OCN_NCPL`. Notice the usage of expandable parameters, guards, and a formula.

Assuming `$OCN_GRID != "MISOMIP"`, `$NCPL_BASE_PERIOD == "day"`, and `$OCN_NCPL==24`, the runtime parameter `DT_THERM` gets reduced to 3600.0 when `.reduce()` method is called.

1.5.1 Utilizing ParamGen class as a base

For each of the file category, we have developed individual classes derived from the `ParamGen` class. These classes are located in `CESM/components/mom/cime_config/MOM_RPS/` and are utilized in the `buildnml` file to generate the corresponding input files.

```
- FType_MOM_params.py
- FType_diag_table.py
- FType_input_data_list.py
- FType_input_nml.py
```

Since the Fortran namelist syntax is already available as an out-of the box format, the most straightforward one is `FType_input_nml.py` which produces the `input.nml` file. The whole module consists of 15 lines of code:

[ ]:
```python
import os, sys

CIMEROOT = os.environ.get("CIMEROOT")
if CIMEROOT is None:
    raise SystemExit("ERROR: must set CIMEROOT environment variable")
sys.path.append(os.path.join(CIMEROOT, "scripts", "lib", "CIME", "ParamGen"))
from paramgen import ParamGen

class FType_input_nml(ParamGen):
    """Encapsulates data and read/write methods for MOM6 (FMS) input.nml file"""
    def write(self, output_path, case):
        self.reduce(lambda varname: case.get_value(varname))
        self.write_nml(output_path)
```

Within the `buildnml` script, the above class is instantiated and utilized as follows:

```
...
input_nml = FType_input_nml.from_json(input_nml_src)
input_nml.write(input_nml_dest, case)
```

(todo) more descriptions and notes to come.

[ ]:

```
```

[ ]:

```
```