pflacs: Faster load cases and parameter studies

Stephen McEntee[‡][*]

https://youtu.be/LgyBPAWDDU8

Abstract—The engineering design process has a significant computational component involving analysis of multiple load cases and parameter studies, with the aim of identifying a combination of design parameters that yields an optimal design solution. Traditionally engineering design methodolgies have been very manual and iterative, however recent developments in computer technologies are driving a growing trend towards automation. This article presents `pflacs`, an open-source Python package that takes advantage of Python's flexible dynamic nature and its introspection tools to provide an object-orientated framework for automating computational studies. `pflacs` binds data and Python functions together in an object-orientated fashion, and uses a tree data structure that is reflective of the hierarchical structure of many design projects. The author has a background in the subsea oil & gas industry, and has applied `pflacs` to automating the design of subsea pipelines. Although its origins are in engineering design, `pflacs` can be used to manage and automate parameter study type analysis in any domain.

Index Terms—parameter study, computational, engineering, tree

Introduction

Reproducibility, data management and workflow management are currently areas of considerable interest and activity in computational scientific research. Recent Scipy conferences have featured several interesting contributions addressing these issues [VRCSAPMD+18] [ACJLH17] [GKC+17] [CSOTATL16] [DLDSLSML+16]

In the domains of civil and mechanical engineering analysis and design, similar issues arise due to the significant component of computational work involved. There are however also some important differences when engineering computational work is compared with scientific research. Engineering design is a very iterative process, and scheduling issues can be critical. Often engineering workscopes are executed in parallel in order to maintain project schedules, even when a serial, or waterfall, workflow might be more appropriate. This can mean that work commences with incomplete information, requiring assumptions to made in the design basis. As the project progress, more information usually becomes available, and also issues inevitably arise, which can result in changes to the design basis. This leads to frequent re-work being required, and since the engineering design process is still very manual and hands-on, re-work can be a significant burden in terms of cost and schedule delay.

* *Corresponding author:* *stephenmce@gmail.com*
‡ *Qwilka Limited*

A historical note

In the 1990s cheap personal computers had become powerful enough to replace the previously more powerful and significantly more expensive class of computers known as engineering "workstations". Where previously engineers had only restricted computer access for computational work, and relied on secretaries for typing reports, it became normal for each individual engineer to have their own desktop computer. Before the advent of the personal computer, engineering design was a mostly manual process, based on paper calculation pads, and engineers tended to favour simpler calculations, ostensibly because simpler engineering theory was considered to be more robust and conservative. However, the fact that most computational work was manual may also have been an important factor, since there is a significant cost penalty associated with increasing computational complexity when calculations are carried out by hand on account of the additional "manhours" required, particularly when re-work is required.

The arrival of the personal computer on the engineer's desk was considered at the time to be a great advancement in terms of productivity improvement, augmentation of computational capability, and general improvement in engineering quality. In reality however, the PC was not so much of a revolution in the engineering office, it was more of an evolution from paper to the desktop computer. Apart from the fact that the actual computational work was now being carried out by computer, the actual work practices and design processes remain much the same to this day. Granted that there has been a significant increase in use of advanced computational technologies, like the finite element method, at the same time a lot of engineering design work continues to be based on the same well-understood, simpler engineering theory, except that now the caculations are carried out on electronic spreadsheets and "worksheets" that greatly resemble their paper predecessors. A time-and-motion study carried out on a junior engineer in most engineering design offices today would reveal that most of their time is spent manually entering data into computer programs, and manually copying-and-pasting the calculated results into word processors, and after that a great deal of time is spend manaully preparing over-formatted "reports" that require constant adjustments to pass quality checks (that are also still carried out manually.)

This at least has been the author's experience in working as a subsea pipeline engineer for more than 20 years in the oil & gas industry. In recent years however, a great change has come to this industry. Since 2014 an unexpected and sustained reduction in the oil price has led to calls to reduce costs, but without compromising safety. It is recognized that oil & gas has fallen behind other industries in terms of productivity and technology uptake, and that digitalization, adopting new technologies and changing the ways we work are widely regarded as means of achieving the goals of sustainably reducing costs without increasing risks.

Introducing pflacs

`pflacs` [SM19a] is a pure Python module that has been developed to manage and expedite computational studies that are typically carried out as part of the engineering design process. The inspiration for `pflacs` came from the author's work as a pipeline engineer in the subsea oil & gas industry [SM16], where design and analysis work has tended to be an intensively manual and iterative process. The geometrical simplicity of pipelines, effectively 1-dimensional structures, means that it is comparatively straightforward to parameterize, compartmentalize and automate their design. That means that pipeline design is the ideal domain in which to demonstrate the capabilities and useage of `pflacs`.

Computational studies, whether in engineering design or scientific research or generally, tend to be hierarchical in structure, with an over-arching fundemental *base-case* study at the root of the project, and multiple, various *load cases* or *parameter studies* that explore variations on the *base-case*. This project hierarchical structure is often exploited by computational analysts by organizing study components in directories or folders in the computer file system.

The limitions of using the computer file system to manage large computational projects quickly become evident, as it gets harder to maintain a consistent naming scheme for parameters and load cases, and other scaling issues arise as the project grows. Typically, even the best organized analyist can quickly fall into an ad-hoc approach to managing data and work flows, and this makes it more difficult to resume work at a later point or for another analyst to take over the project workscope.

The objective of `pflacs` is to address these issues in a familiar `Python` computational environment. `pflacs` inherits from a companion `Python` module called `vntree` [SM19b], and that makes `pflacs` a tree data structure. Study input parameters become attributes of the nodes in a `pflacs` tree, and when a node requires a parameter it can ascend the tree to find its value, if the parameter is not an attribute of that node. So effectively parameters can be inherited from higher levels in the tree structure.

Computational functionality is added by *plugging-in* (or *patching*) external `Python` functions, turning the functions into class attributes, or methods, that are available to all the nodes in the tree. The plugged-in functions are bound to the parameter attributes, and this means that it is not necessary to explicity specify the function arguments when a function is invoked on a `pflacs` node. If an argument is not specified in a function call, `pflacs` will substitute the value of the parameter attribute it finds with the same name as the required argument. Binding node parameter attributes to functions in this manner facilitates automation of computations.

`pflacs` achieves this by using the introspection tools provided by Python's `inspect` module. Then an external function is plugged in, `pflacs` uses the `inspect.Signature` class to obtain the call signature of the function. When the function is invoked on a `pflacs` node, the function call signature is used to match any unspecified arguments with the appropriate parameters.

`pflacs` is a lightweight and unopinionated environment, the only requirement is that the user adopts their own naming scheme for parameters, and maintains consistency with that scheme within the project. The idea behind this approach is to allow the user to re-use, or re-purpose, existing code without the need to alter or adapt the original code. There is no requirement to decorate or modify plug-in functions, which means that external functions can continue to be used in their original form as standalone code, or in another computational environment outside of `pflacs`. The only restriction on this is that `pflacs` plugin functions must be pure Python code, due to the dependency on `inspect.Signature` which has this limitation. In order to use `pflacs` with compiled libraries, like the functions in Python's built-in `math` module, the work-around would be to wrap the compiled function inside a Python wrapper function which can be accessed by `inspect.Signature`.

We will now further explore the use and capabilities of `pflacs` through two examples, first a very simple study that showcases basic usage, and after that a real-life example is presented demonstrating the engineering design of a subsea pipeline using `pflacs`.

Basic usage

Taking a very simple example to illustrate basic usage, we start by importing the `pflacs.Premise` class. `Premise` is the fundemental class in `pflacs`, it is a sub-class of `vntree.Node` [SM19b], and hence `Premise` instances are nodes in a tree data structure. The purpose of `Premise` is to contain the study parameters (these are the *premise* of the study), and to group together other tree nodes.

```
from pflacs import Premise
base = Premise("Base case",
               parameters={"a":10, "b":5} )
print(f"base.a={base.a} base.b={base.b}")
```

```
base.a=10 base.b=5
```

The `parameters` dictionary items are passed to a method `Premise.add_param` that uses a `pflacs` descriptor class called `Parameter` to convert the parameters into attributes of the `Premise` node instance.

We would like to add some functionality to our study, so taking a very simple function:

```
def adda(a, b, c=0):
    print(f"«adda» w/args a={a} b={b}", end="")
    print(f" c={c}") if c else print()
    return a + b + c
```

and using the method `Premise.plugin_func` to plug-in (or "patch") the function `adda` into our study tree nodes, and invoking `adda` on instance `base`:

```
base.plugin_func(adda)
result = base.adda()
print(f"base.adda() result={result}")
```

```
«adda» w/args a=10 b=5
base.adda() result=15
```

Method `plugin_func` invokes a *pflacs* class called *Function* that wraps the plug-in function and binds it to the *Premise* node instance. The *Function* class uses Python's `inspect.Signature` class to determine the plug-in function's call signature, which includes names of the arguments that *adda* requires. When *adda* is invoked on a *Premise* node, any argument that is not explicitly specified is supplied from the node attribute with the same name. If an attribute with the argument name is not found in the current node instance, *pflacs* ascends the tree until it finds an ancestor node that has the required attribute, and applies its value as the required argument.

So, argument values are applied in accordance with the following precedence order:

1) argument explicitly specified in function call,
2) node instance attribute,
3) ancestor node attribute,
4) original function default value.

The follow examples use explicit arguments, node instance attribute values, and function default values:

```
result = base.adda(b=-3)
print(f"base.adda(b=-3) result={result}")
result = base.adda(5, 4.2, -3)
print(f"base.adda(5,4.2,-3) res={result}")
```

```
«adda» w/args a=10 b=-3
base.adda(b=-3) result=7
«adda» w/args a=5 b=4.2 c=-3
base.adda(5,4.2,-3) res=6.199999999999999
```

To make things a bit more interesting, we will add more functionality:

```
def subx(x, y, z=0):
    print(f"«subx» w/args x={x} y={y}", end="")
    print(f" z={z}") if z else print()
    return x - y - z
```

Inconveniently, the arguments of function `subx` do not correspond with our adopted parameter naming scheme, so we need to supply a mapping to indicate how the node parameters/ attributes should be applied to `subx`. We will also introduce a new parameter as instance attribute `base.c`:

```
base.plugin_func(subx, argmap={"x":"a",
        "y":"b", "z":"c"} )
base.add_param("c", 6.5)
print("base.subx() =", base.subx() )
print("base.subx(b=99) =", base.subx(b=99) )
```

```
«subx» w/args x=10 y=5 z=6.5
base.subx() = -1.5
«subx» w/args x=10 y=99 z=6.5
base.subx(b=99) = -95.5
```

We would now like to introduce a new load case, or parameter study, so we instantiate a new *Premise* node with root node *base* as its parent:

```
lc1 = Premise("Load case 1", parent=base,
                              parameters={"a":100})
result = lc1.adda()
print(f"lc1.adda() result={result}")
```

```
«adda» w/args a=100 b=5 c=6.5
lc1.adda() result=111.5
```

Node «Load case 1» has its own attribute *a* and it applies the value `lc1.a` as the first argument to *adda*. Node «Load case 1» inherits values for attributes `lc1.b` and `lc1.c` from its parent node *base*, and applies those values as *adda* arguments *b* and *c* in the function call.

Premise nodes do not automatically store the results of function calls, but we now introduce a new node class that does. *pflacs.Calc* is a sub-class of *Premise* that has a defined `__call__` method that invokes a specific plug-in function.

```
from pflacs import Calc
lc1_sub = Calc("LC1 «subx()»", lc1, funcname="subx")
lc1_sub(); print(lc1_sub._subx)
```

```
«subx» w/args x=100 y=5 z=6.5
88.5
```

The return value that results from executing the `Calc` node is assigned to a node attribute called `_subx`. By default, this result attribute takes its name from the function, prefixed with an underscore to avoid a name-clash. The name of the return result attribute can be specified by adding an item with key 'return' to the argument mapping:

```
lc1_add = Calc("LC1 «adda()»", lc1, funcname="adda",
                        argmap={"return":"adda_res"})
lc1_add(); print(lc1_add.adda_res)
df = lc1_add.to_dataframe(); print(df)
```

```
111.5
.    a   b    c  adda_res
0  100   5  6.5     111.5
```

The `Calc.to_dataframe` method creates a `Pandas` dataframe from the argument values and the function return value.

We would now like to create another parameter study, similar to "Load case 1". The easiest way to do this is to copy the branch we have already prepared, and make the necessary changes to the new branch. In this code block, we are using tree methods inherited from *vntree.Node*:

```
lc2 = base.add_child( lc1.copy() )
lc2.name = "Load case 2"
lc2.a = 200
lc2_sub = lc2.get_child_by_name("LC1 «subx()»")
lc2_sub.name = "LC2 «subx()»"
lc2_add = lc2.get_child_by_name("LC1 «adda()»")
lc2_add.name = "LC2 «adda()»"
```

Let's add more functionality to our study. Again, we are plugging-in a function that has argument names that are inconsistent with our parameter naming scheme:

```
def multk(k:"a", l:"b", m:"c" = 1) -> "mult_res":
        return k * l * m
base.plugin_func(multk)
result = base.multk()
print(f"{base.a} * {base.b} * {base.c} = {result}")
```

```
10 * 5 * 6.5 = 325.0
```

Here, we are taking advantage of Python's function annotations to avoid having to explicitly specify an argument map for plug-in function `multk`. If we did not have access to the original function code, or if we wanted to use function annotations for other purposes, we would define argument `argmap={"k":"a", "l":"b", "m":"c", "return":"mult_res"}` when invoking method `plugin_func` in this case.

Let's add another `Calc` node using `multk`:

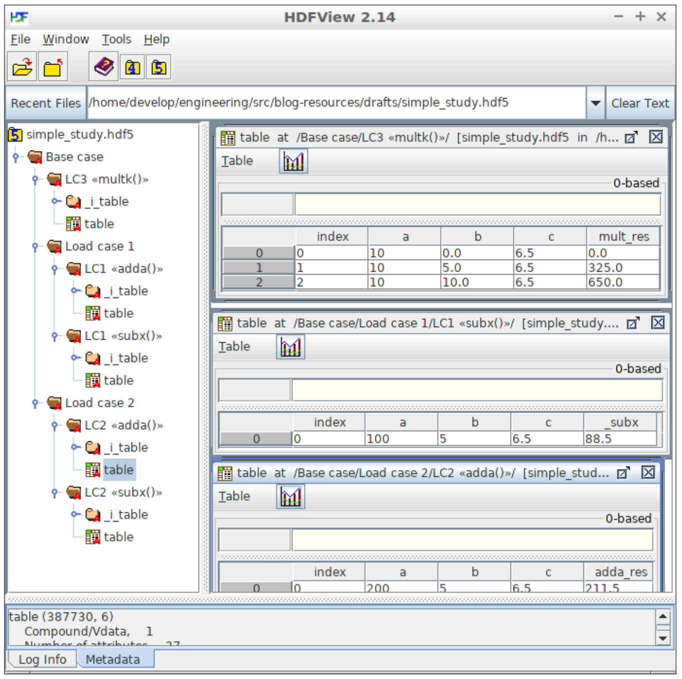
```
lc3_mul = Calc("LC3 «multk()»", base, funcname="multk")
import numpy as np
lc3_mul.b = np.linspace(0,10,3)
lc3_mul()
lc3_mul.to_dataframe()
```

```
.   a     b    c  mult_res
0  10   0.0  6.5       0.0
1  10   5.0  6.5     325.0
2  10  10.0  6.5     650.0
```

Let's take a look at the tree structure of the study we have built:

```
print(base.to_texttree())
```

```
|Base case
+--|Load case 1
.   +--|LC1 «subx()»
.   .   |LC1 «adda()»
```



/home/develop/engineering/src/blog-resources/drafts/simple_study.hdf5

```
simple_study.hdf5
  Base case
    LC3 «multk()»
      _i_table
      table
    Load case 1
      LC1 «adda()»
        _i_table
        table
      LC1 «subx()»
        _i_table
        table
    Load case 2
      LC2 «adda()»
        _i_table
        table
      LC2 «subx()»
        _i_table
        table
```

table at /Base case/LC3 «multk()»/ [simple_study.hdf5 in /h...

	index	a	b	c	mult_res
0	0	10	0.0	6.5	0.0
1	1	10	5.0	6.5	325.0
2	2	10	10.0	6.5	650.0

table at /Base case/Load case 1/LC1 «subx()»/ [simple_study....

	index	a	b	c	_subx
0	0	100	5	6.5	88.5

table at /Base case/Load case 2/LC2 «adda()»/ [simple_stud...

	index	a	b	c	adda_res
0	0	200	5	6.5	211.5

```
table (387730, 6)
   Compound/Vdata,    1
```

Fig. 1: *Tables of study results in HDFview.*

```
.   |Load case 2
.   +--|LC2 «subx()»
.   .   |LC2 «adda()»
.   |LC3 «multk()»
```

A `vntree.Node` instance is a generator, the whole tree, or a sub-tree, can be traversed simply by interating over the root node. In this example, the study tree is traversed top-down, and all the `Calc` found are executed:

```
for node in base:
  if type(node) == Calc:
    node()
```

Now that our study has been re-calculated, we will save it:

```
base.savefile("simple_study.pflacs")
```

This saves the study tree in the Python `pickle` format. To re-open the study, we would use the class method :code:‘ Premise.openfile‘:

```
new_study = Premise.openfile("simple_study.pflacs")
```

Using the `pickle` format to persist *pflacs* trees is convenient because it can easily serialize most common *Python* object types. However, we could be using *pflacs* to carry out large projects with many input parameters and calculation loadcases, in which case there would a lot of output data to save. In that case, *HDF5* is a more suitable format for saving the results dataframes in tables:

```
for node in base:
  if type(node) == Calc:
    node.to_hdf5()
```

As you can see in Figures 1

Engineering design example

TODO: describe use of pflacs for design of subsea pipeline.

References

[ACJLH17] Alicia Clark and Joseph L. Hellerstein. SciSheets: Providing the Power of Programming With The Simplicity of Spreadsheets. In Katy Huff, David Lippa, Dillon Niederhut, and M Pacer, editors, *Proceedings of the 16th Python in Science Conference*, pages 41 – 48, 2017. doi:10.25080/shinma-7f4c6e7-007.

[CSOTATL16] Christian Schou Oxvig, Thomas Arildsen, and Torben Larsen. Storing Reproducible Results from Computational Experiments using Scientific Python Packages. In Sebastian Benthall and Scott Rostrup, editors, *Proceedings of the 15th Python in Science Conference*, pages 45 – 50, 2016. doi:10.25080/Majora-629e541a-006.

[DLDSLSML+16] David L. Dotson, Sean L. Seyler, Max Linke, Richard J. Gowers, and Oliver Beckstein. datreant: persistent, Pythonic trees for heterogeneous data. In Sebastian Benthall and Scott Rostrup, editors, *Proceedings of the 15th Python in Science Conference*, pages 51 – 56, 2016. doi:10.25080/Majora-629e541a-007.

[GKC+17] Klaus Greff, Aaron Klein, Martin Chovanec, Frank Hutter, and Jürgen Schmidhuber. The sacred infrastructure for computational research. In Katy Huff, David Lippa, Dillon Niederhut, and M Pacer, editors, *Proceedings of the 16th Python in Science Conference*, pages 49–56, Austin, TX, 2017.

[SM16] Stephen McEntee. Cost, Design and Data in Subsea, 2016. URL: https://www.linkedin.com/pulse/cost-design-data-subsea-stephen-mcentee.

[SM19a] Stephen McEntee. «pflacs» faster load cases and parameter studies, 2019. URL: https://github.com/qwilka/pflacs.

[SM19b] Stephen McEntee. «vntree» a simple python tree data structure, 2019. URL: https://github.com/qwilka/vn-tree.

[VRCSAPMD+18] Vyas Ramasubramani, Carl S. Adorf, Paul M. Dodd, Bradley D. Dice, and Sharon C. Glotzer. signac: A Python framework for data and workflow management. In Fatih Akici, David Lippa, Dillon Niederhut, and M Pacer, editors, *Proceedings of the 17th Python in Science Conference*, pages 152 – 159, 2018. doi:10.25080/Majora-4af1f417-016.