Expanding the PyVAFM

C Circuits

In this short tutorial the process of adding a new circuit to the pyvafm in C will be explained. Circuits in the PyVAFM contain two components one in Python that is used for the setup and one in C used for the actual processing. In this tutorial we will create a very simple gain circuit that you can use as a template to create more complicated circuits later on.

Circuit PyVAFM Setup

The python part of a circuit is used to set up a circuit and then we will pass our setup circuit to the C part for processing. First thing we must do is create a new python script and save it within /src folder of the PyVAFM. The name of the file doesn't matter although we have used the format of vafmcircuits_NAME.py where NAME is the name of the collections of circuit for example filters. So Create a new python script named "vafmcircuits_TutCircuit.py" in the /src folder of the PyVAFM.

Now that we have a new python file within our /src folder we must include this file within the PyVAFM. Within the src folder there is a file called "vafmcircuits.py". If you open this file there is a lot of various modules being imported here. We need to import our new python script we have included so add "import vafmcircuits_TutCircuit" to the header of vafmcircuits.py.

Python Circuit Setup

So now we have added our circuit file and added it to the pyvafm. Now we must set up our circuit, so reopen vafmcircuits_TutCircuit.py. We must import the circuit module from vafmbase, so lets add "from vafmbase import Circuit" to the top of our file. Since we are going to transfer data to c we must also import ctypes, so also add "from ctypes import *". So now our circuit file should only contain what is shown below.

```
1  from vafmbase import Circuit
2  from ctypes import *
```

Next lets add our circuit class. Add the following code to your file now:

```
4  class TutCircuit(Circuit):
5
6      def __init__(self, machine, name, **keys):
7
8          super(self.__class__, self).__init__( machine, name )
```

This is the initialisation for the class, this will be called when AddCircuit is used within the input script. The above code snippet remains the same for all circuits except the class name (in this case "TuTCircuit") must be different for each new circuit. This class name dictates the type of circuit you have created for example the above is a circuit of type TutCircuit.

Since we are creating a gain circuit lets add a gain variable and make it equal to one. Since we also want the user to be able to alter the gain as they want we must include some code to allow this to happen. When you add a circuit in the PyVAFM there are several options for most circuits, we call these initialisation parameters. This can be accessed within this circuit by using the keys dictionary. For example in this case we expect the user to input a initialisation parameter called gain, so if we access keys['gain'] we can get that user entered value. So lets add a small bit of code that checks if gain is entered as a initialisation parameter and if so make it equal to that user entered value.

```
9
10          gain=1
11
12          if 'gain' in keys.keys():
13              gain = float(keys['gain'])
```

So our code will look like this now:

```
1  from vafmbase import Circuit
2  from ctypes import *
3
4  class TutCircuit(Circuit):
5
6      def __init__(self, machine, name, **keys):
7
8          super(self.__class__, self).__init__( machine, name )
9
10         gain=1
11
12         if 'gain' in keys.keys():
13             gain = float(keys['gain'])
```

Ok next we must add our input and output channel names. We an add as many as we want (although the order does matter later on as we will see). So to add input channel we use self.AddInput("ChannelName") and for output self.AddOutput("ChannelName"), where ChannelName is the custom name you are giving to the channel. So since we are making a gain circuit we need one input named in and one output named out, so lets add self.AddInput("in") and self.AddOutput("out") as shown below:

```
15          self.AddInput("in")
16          self.AddOutput("out")
```

And our code now looks like this:

```
1  from vafmbase import Circuit
2  from ctypes import *
3
4  class TutCircuit(Circuit):
5
6      def __init__(self, machine, name, **keys):
7
8          super(self.__class__, self).__init__( machine, name )
9
10         gain=1
11
12         if 'gain' in keys.keys():
13             gain = float(keys['gain'])
14
15         self.AddInput("in")
16         self.AddOutput("out")
```

So now we have setup the channels, next we have to point our setup to the correct C method. Wehaven't written the method yet but we can start setting it up. In order to do this we use the following code format:

self.cCoreID = Circuit.cCore.CMETHOD(self.machine.cCoreID,INITIALISATION PARAMETERS)

Where CMETHOD is the name of the initialisation method in our c code and INITIALISATION PARAMETERS is the arguments we plan to pass to our C method. So we plan to make a Method called "Add_TutCirc" and we need to pass the "gain" argument so we have to add the following code snippit. It is worth noting that the data type in the arguments we enter here must equal the data type in the c code, for example in the C code gain will be a double so we must cast it is a c_double in this case.

```
18        self.cCoreID = Circuit.cCore.Add_TutCirc(self.machine.cCoreID,c_double(gain))
```

Finally we must add "self.SetInputs(**keys)" next:

```
20        self.SetInputs(**keys)
```

Now we are one with the python setup and our final code will look like this:

```
1  from vafmbase import Circuit
2  from ctypes import *
3
4  class TutCircuit(Circuit):
5
6      def __init__(self, machine, name, **keys):
7
8          super(self.__class__, self).__init__( machine, name )
9
10         gain=1
11
12         if 'gain' in keys.keys():
13             gain = float(keys['gain'])
14
15         self.AddInput("in")
16         self.AddOutput("out")
17
18         self.cCoreID = Circuit.cCore.Add_TutCirc(self.machine.cCoreID,c_double(gain))
19
20         self.SetInputs(**keys)
21
```

Ccore setup

Now we have the python part setup time to setup the C part. First thing we must do is create two files a normal .c and a .h file. So in this tutorial we will make core_tutcirc.c and core_tutcirc.h. These files must be placed in the src/cCore folder.

First open core_tutcirc.c and include “circuit.h” as well as the headerfile you just made in the preamble so for example in our case we use the following code snippet:

```
1  #include "circuit.h"
2  #include "core_tutcirc.h"
3
```

Next we must add the initialisation method for our circuit. The name of this circuit must be same as we used in the python part. So in the python part we used:

self.cCoreID = Circuit.cCore.Add_TutCirc(self.machine.cCoreID,c_double(gain))

hence we must name our method Add_TutCirc and include two input arguments one int and one double as shown below.

```
3
4  int Add_TutCirc(int owner, double gain) {
5
```

So far our code looks like this:

```
1  #include "circuit.h"
2  #include "core_tutcirc.h"
3
4  int Add_TutCirc(int owner, double gain) {
5
```

Now we need to initialise our circuit so first lets get a new instance of the circuit class by using “circuit c = NewCircuit();”

```
6      circuit c = NewCircuit();
```

after which we can set up the number of input and output channels our circuit is using by setting c.nI and c.nO variables. In this case we have one input and one output so we can set it up as shown below:

```
    c.nI = 1;
    c.nO = 1;
```

So far our code looks like this:

```
1  #include "circuit.h"
2  #include "core_tutcirc.h"
3
4  int Add_TutCirc(int owner, double gain) {
5
6      circuit c = NewCircuit();
7      c.nI = 1;
8      c.nO = 1;
```

In order to save variables between time steps we must implement an array of parameters. This comes in 3 flavours in the PyVAFM, iparmas for integers, params for doubles and vparams for any data type. So we know we only need to save one data type here (gain) between timesteps so lets set “c.plen = 1;” and use “c.params = (double*)calloc(c.plen,sizeof(double));” to setup our array.

```
c.plen = 1;
c.params = (double*)calloc(c.plen,sizeof(double));
```

Now that our array is setup let fill it up. Since we have only one element we can make “c.params[0] = gain”

```
c.params[0]=gain;
```

So far our code looks like this:

```
#include "circuit.h"
#include "core_tutcirc.h"

int Add_TutCirc(int owner, double gain) {

    circuit c = NewCircuit();
    c.nI = 1;
    c.nO = 1;

    c.plen = 1;
    c.params = (double*)calloc(c.plen,sizeof(double));
    c.params[0]=gain;

```

Now we need to tell the C code where the update function is located. The update function is the function in a circuit that is ran at every timestep. This is done by making c.updatef equal to the name of the update function. We plan to make a update function named “TutCirc” so lets make c.updatef equal to this as shown below:

```
    c.updatef = TutCirc;
```

Finally lets add in the following code snippet, all this does is get the circuit index (individual identifier for each circuit) and returns it back to the python, to the variable self.cCoreID that we used earlier in the python part of the setup. also included is a print statement that will print a message to the console this can be whatever you want it to be. This is all shown below:

```
    int index = AddToCircuits(c,owner);
    printf("cCore: added Tutorial Circuit\n");
    return index;

```

So far our code looks like this: (don't forget to add the } to designate the end of the method)

```
#include "circuit.h"
#include "core_tutcirc.h"

int Add_TutCirc(int owner, double gain) {

    circuit c = NewCircuit();
    c.nI = 1;
    c.nO = 1;

    c.plen = 1;
    c.params = (double*)calloc(c.plen,sizeof(double));
    c.params[0]=gain;

    c.updatef = TutCirc;

    int index = AddToCircuits(c,owner);
    printf("cCore: added Tutorial Circuit\n");
    return index;

}
```

Ok so now lets setup the update part of the function. First thing we have to do is name our method the same as we set our c.updatef earlier. So in this case we used TutCirc so lets use the below code snippet:

```
22  void TutCirc( circuit *c ) {
```

In order to access the data channels we have to use GlobalSignals[c->inputs[INDEX]]; for the input channels and GlobalBuffers[c->outputs[INDEX]]; for the output channels. Where index is the order (starting from 0) on which we added the input and output circuits. So in this case we only added one input and one output so INDEX is 0 in both our cases. If we had several input channels then the value of index will correspond to the order of which we added channels in the python. So for example if we added two input channels then index = 0 would be the first channel and index = 1 would be the second input channel. The same applies for output channels. So the following code snippet takes the input channel value and makes it equal to the variable input, add this next to your code:

```
    double input = GlobalSignals[c->inputs[0]];
```

Now in every time step we want to take this input signal and multiply by the gain we specified earlier. We can access the param variable we set earlier by using 'c->parmas[INDEX]:' where index is the index you set that variable to in the initialisation method we did previously. (for iparmas use c->iparms[INDEX] and for vparams use c->vparams[INDEX]). So finally lets multiply the input value we got from the last step by c->params[0] and update the output channel (GlobalBuffers[c->outputs[0]]).

```
    GlobalBuffers[c->outputs[0]] = input * c->params[0];
```

```c
#include "circuit.h"
#include "core_tutcirc.h"

int Add_TutCirc(int owner, double gain) {

    circuit c = NewCircuit();
    c.nI = 1;
    c.nO = 1;

    c.plen = 1;
    c.params = (double*)calloc(c.plen,sizeof(double));
    c.params[0]=gain;

    c.updatef = TutCirc;

    int index = AddToCircuits(c,owner);
    printf("cCore: added Tutorial Circuit\n");
    return index;

}

void TutCirc( circuit *c ) {
        double input = GlobalSignals[c->inputs[0]];
        GlobalBuffers[c->outputs[0]] = input * c->params[0];

    }
```

So now our update function looks like this:

```c
void TutCirc( circuit *c ) {
        double input = GlobalSignals[c->inputs[0]];
        GlobalBuffers[c->outputs[0]] = input * c->params[0];

    }
```

So now we have finally completed our .c file for our circuit and should look like this :

Now all we have to do is build our header file that we defined early (core_tutcirc.h).

In this file we have to include the methods we have in our circuit, so in this case int Add_TutCirc and void TutCirc. This is shown below.

```
2  int Add_TutCirc(int owner,double gain);
3  void TutCirc( circuit *c );
4
```

Make File

Ok almost done one last thing before testing is to add the c files we made to our makefile!

So open the makefile within the src/cCore folder. So we need to add our .c file that we made to the end of this (before the clean function). So the format is:

```
COMPILERVARNAME:
        $(CC) $(CFLAGS) CFILENAME
```

Where COMPILERVARNAME is a user defined name that is used by the makefile and CFILENAME is the name of the c file we just made. So in our case lets set COMPILERVARNAME to tutcirc and CFILENAME is core_tutcirc.c.

```
54  tutcirc:
55      $(CC) $(CFLAGS) core_tutcirc.c
```

Now we have to add our COMPILERVARNAME to the list at the top. On line 9 of the makefile you can see a large list of variable names. Simple append your COMPILERVARNAME to the end of this. For example in our case our COMPILERVARNAME is tutcirc so we will append it to the end as shown below:

```
all: container cantilever siggen scanner interpo outputs rsa flops maths logics filters comparison control sigproc avg vdw stm pyc dipole main.o tutcirc
```

So now our make file should look like the one below where the change are highlighted in red.

```
makefile
CC  = gcc
CFLAGS  = -c -w -std=c99 -fpic -O3
LFLAGS  = -w -lm -shared -fpic -O3 -o vafmcore.so

PYINC   = /usr/include/python2.7/

#CIRCUITS = core_container.o core_signals.o core_output.o core_maths.o core_logic.o  core_filters.o

all: container cantilever siggen scanner interpo outputs rsa flops maths logics filters comparison control sigproc avg vdw stm pyc dipole main.o tutcirc
    $(CC) $(LFLAGS) *.o
    rm *.o
    cp vafmcore.so ../.
    cp vafmcore.so ../../examples/.
main.o:
    $(CC) $(CFLAGS) main.c
cantilever:
    $(CC) $(CFLAGS) core_cantilever.c
scanner:
    $(CC) $(CFLAGS) core_scanner.c
interpo:
    $(CC) $(CFLAGS) core_interpolation.c
outputs:
    $(CC) $(CFLAGS) core_output.c
rsa:
    $(CC) $(CFLAGS) core_rsa.c
maths:
    $(CC) $(CFLAGS) core_maths.c
logics:
    $(CC) $(CFLAGS) core_logic.c
comparison:
    $(CC) $(CFLAGS) core_comparison.c
siggen:
    $(CC) $(CFLAGS) core_signals.c
filters:
    $(CC) $(CFLAGS) core_filters.c
control:
    $(CC) $(CFLAGS) core_control.c
flops:
    $(CC) $(CFLAGS) core_flipflops.c
sigproc:
    $(CC) $(CFLAGS) core_signalprocessing.c
container:
    $(CC) $(CFLAGS) core_container.c
avg:
    $(CC) $(CFLAGS) core_avg.c
vdw:
    $(CC) $(CFLAGS) core_VDW.c
stm:
    $(CC) $(CFLAGS) core_STM.c
dipole:
    $(CC) $(CFLAGS) core_Dipole.c
pyc:
    $(CC) $(CFLAGS) -I$(PYINC) core_pycircuit.c
tutcirc:
    $(CC) $(CFLAGS) core_tutcirc.c

clean:
    rm -rf *.o vafmcore.so
```

And we are done!

now navigate the src/cCore and type make into the console to recompile the pyvafm!

Testing

So lets make a quick test script. The script is shown below:

```python
from vafmcircuits import Machine

machine = Machine(name='machine', dt=0.01)

machine.AddCircuit(type='waver', name='osc', freq=1, amp=1)

machine.AddCircuit(type='TutCircuit', name='TutCircuit',gain=2)

machine.Connect('osc.sin','TutCircuit.in')

logger = machine.AddCircuit(type='output', name='logger', file='tutorial_AddCircuits.out', dump=1)
logger.Register('global.time','osc.sin','TutCircuit.out')

machine.Wait(1)

```

All we are doing here is taking a sin wave and putting it through our new gain circuit. I

So only thing special here is we added our new circuit

```
machine.AddCircuit(type='TutCircuit', name='TutCircuit',gain=2)
```

It is worth noting the type argument here is equal to the same name as the circuit class we made in python way at the start. Also we have our gain argument that we have established in the python part as well.

So run this script in the console and you should get an output of two wave one which is multiplied by 2!

This tutorial has taken you through how to make a simple c circuit in the PyVAFM it is also possible to make circuits in Python which is considerably simpler but also quite a bit slower, so this method is well worth learning for repeated or heavy applications.

The operation in this tutorial was very simple but lays a basis ground for how each of the c circuits in the PyVAFM are implemented. If you want more examples simply look at the source files for the PyVAFM and you can see all the circuits are implemented like this giving the user a wealth of examples from which to build their own circuits.

Python Circuits

Python circuits are very easy to implement and follow the same ideas as with the C circuits.

In order to add a python circuit it must be added to the file vafmcircuits_pycirc.py located within the src folder.

So lets say we want to make a simple multiplication circuit. Lets make a new class called myCirc that inherits the PYCircuit class. this is done by using the below code snippet:

```
class myCirc(PYCircuit):

    def __init__(self, machine, name, **keys):

        super(self.__class__, self).__init__( machine, name )
```

Next lets add our input channels in this case we want two inputs and one output, so this is done by adding:

```
44          self.AddInput("in1")
45          self.AddInput("in2")
46          self.AddOutput("out")
47
48
49          self.Create(**keys)
```

Now our initialisation is done. If we wanted to access initialisation variables from the setup we can simply access them by using the keys dictionary. So for example if we had a initialisation parameter called gain we can access it by using keys[‘gain’].

so now we have to set up our update function. so make a new method called def Update(self). in order to access input channels we use self.I[“CHANNELNAME"].value and for output we use self.O[“CHANNELNAME"].value where CHANNELNAME is the name of the channels we just defined in the previous step. So this update function is called at every time step and we want to make the output equal the product of both the input channels. This is simply done as shown below:

```
51
52      def Update(self):
53
54          self.O["out"].value = self.I["in1"].value*self.I["in2"].value
55
56
```

And we are done! The whole Python circuit will look like this

```
class myCirc(PYCircuit):

    def __init__(self, machine, name, **keys):

        super(self.__class__, self).__init__( machine, name )

        self.AddInput("in1")
        self.AddInput("in2")
        self.AddOutput("out")


        self.Create(**keys)


    def Update(self):

        self.O["out"].value = self.I["in1"].value*self.I["in2"].value
```

Now to test it, we can add it to our simulation like any other circuit

```
machine.AddCircuit(type='myCirc',name='pytest', pushed=True )
```

So a full input script can look like this:

```python
#!/usr/bin/env python

from vafmcircuits import Machine

machine = Machine(name='machine', dt=0.01, pushed=True);


#Add Circuits

machine.AddCircuit(type='waver',name='wave', amp=1, freq=1, pushed=True )
    machine.AddCircuit(type='myCirc',name='pytest', pushed=True )

machine.Connect("wave.sin","pytest.in1")
machine.Connect("wave.cos","pytest.in2")

out1 = machine.AddCircuit(type='output',name='output',file='test_pycircuit.out', dump=1)
out1.Register('global.time', 'wave.sin', 'pytest.out')

machine.Wait(1)
```

So if you run this script you should see an output where the sin wave is multiplied by the cos wave.