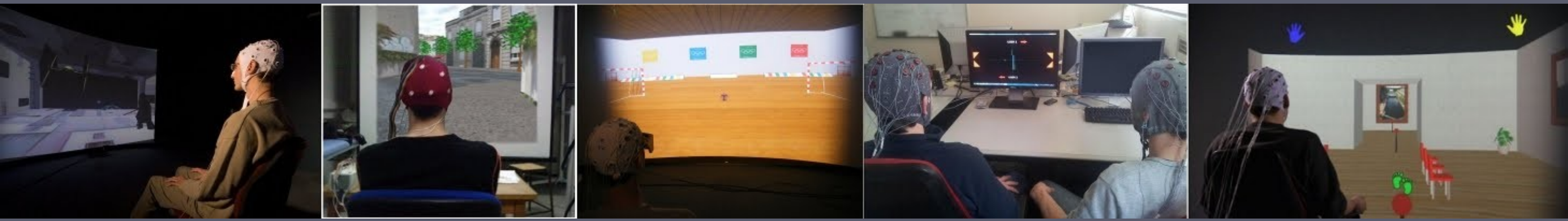
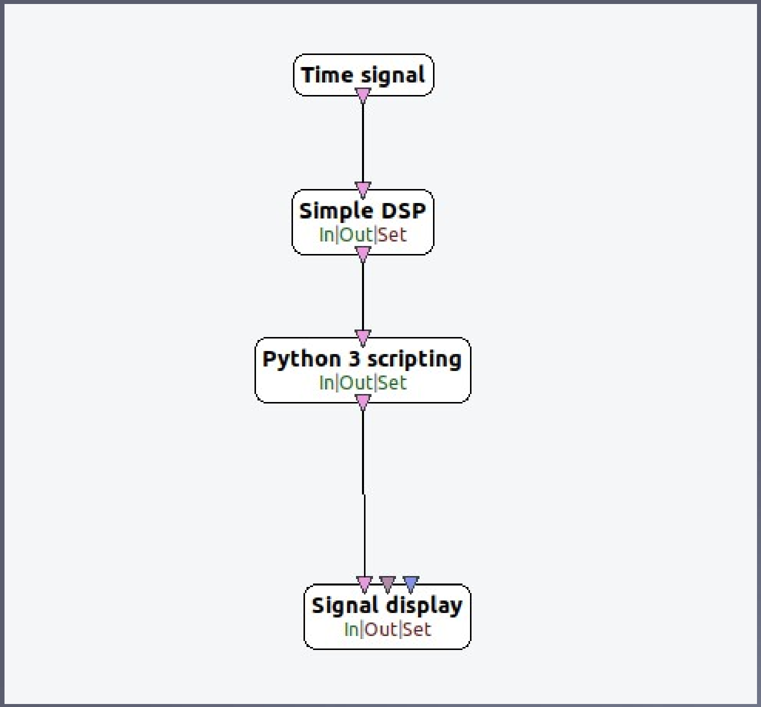




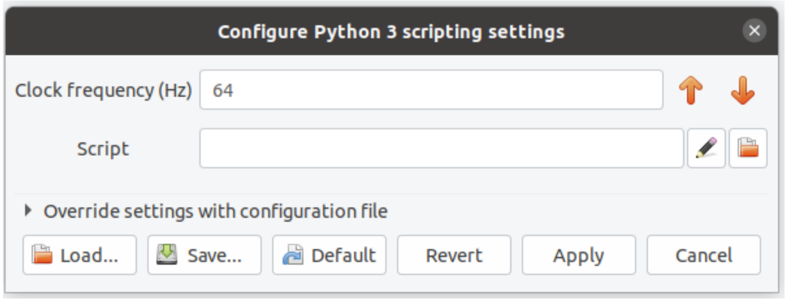
Prototyping with Python in OpenViBE

Thomas Prampart,
Inria Rennes Bretagne Atlantique

Why Python in OpenViBE ?

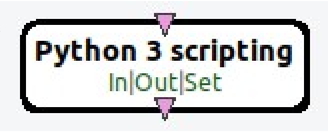
- Prototyping:
  - Implement a quickly a new box, that can be ported later to C++
- Extending:
  - You can't do what you want with the exisiting boxes
- Reuse of code:
  - You may already have the python implementation of a specific algorithm
- *Beware: Python processing will be much slower than its C++ equivalent*

Getting started - Designer



Double-click

Right-click to add inputs, outputs, settings...



Ready to plug and play !

Getting started - Script

What is needed :

- One class inheriting from OVBox
- 3 override methods that OpenViBE will call
- A 'box' variable for the class instance

Note : OpenViBE specific modules are imported automatically.

```
class MyOVBox(OVBox):
    def __init__(self):
        OVBox.__init__(self)

    def initialize(self):
        # This method is called once when the scenario is started
        # Initialize class members
        # Maybe send streams Headers

    def process(self):
        # Get Inputs (stimulations, signal or matrix)
        # Process the inputs
        # Generate output (signal, stimulation, display...)

    def uninitialize(self):
        # Release data
        # Maybe send streams Enders


box = MyOVBox()
```

Getting started – Python objects

Stream objects – Inherit OVChunk:

- Signal
  - *OVSignalHeader, Buffer, End*
- StreamedMatrix
  - *OVStreamedMatrixHeader, Buffer, End*
- Stimulation
  - *OVStimulationHeader, Set, End*
- OVBuffer :
  - *Deque containing OVChunk objects*

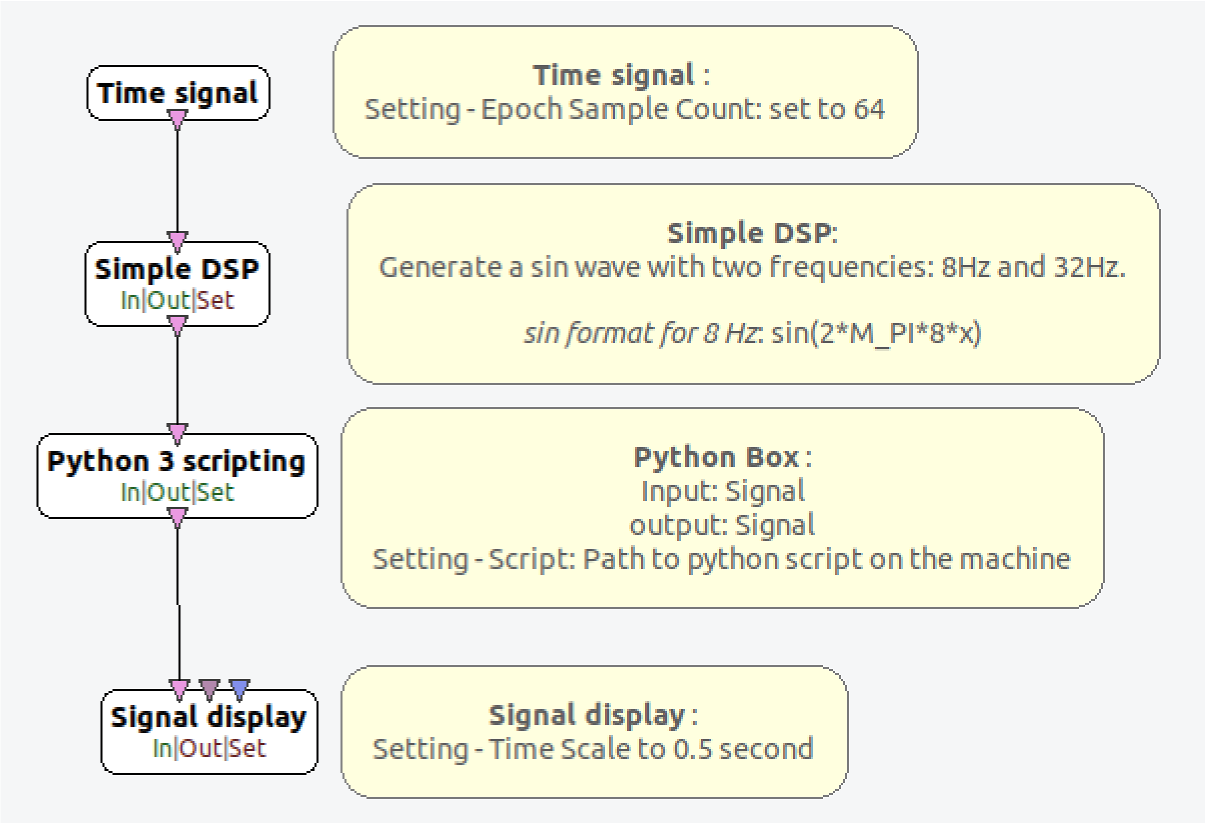
Class members :

- self.input : list(OVBuffer)
  - *Get chunk of data on input 1: self.input[0].pop()*
- self.output : list(OVBuffer)
  - *Add chunk of data to output 1 : self.output[0].append(chunk)*
- self.setting : dict()
  - *Get parameter value : param = self.setting['name']*

Note: The **OVSignalBuffer** and **OVStreamedMatrixBuffer** types can be **used as lists to access the data**

More details on the data types and how to use them on the OpenVibe website.

Step 1 : Create a passthrough script



Step 1 : Create a passthrough script

```
def initialize(self):
    # Declare signal header
    self.signalHeader = None
```

```
def process(self):
    # Loop through input 1 chunks
        # Check if chunk is Header, Buffer or End
        # For each, forward it to output 1
```

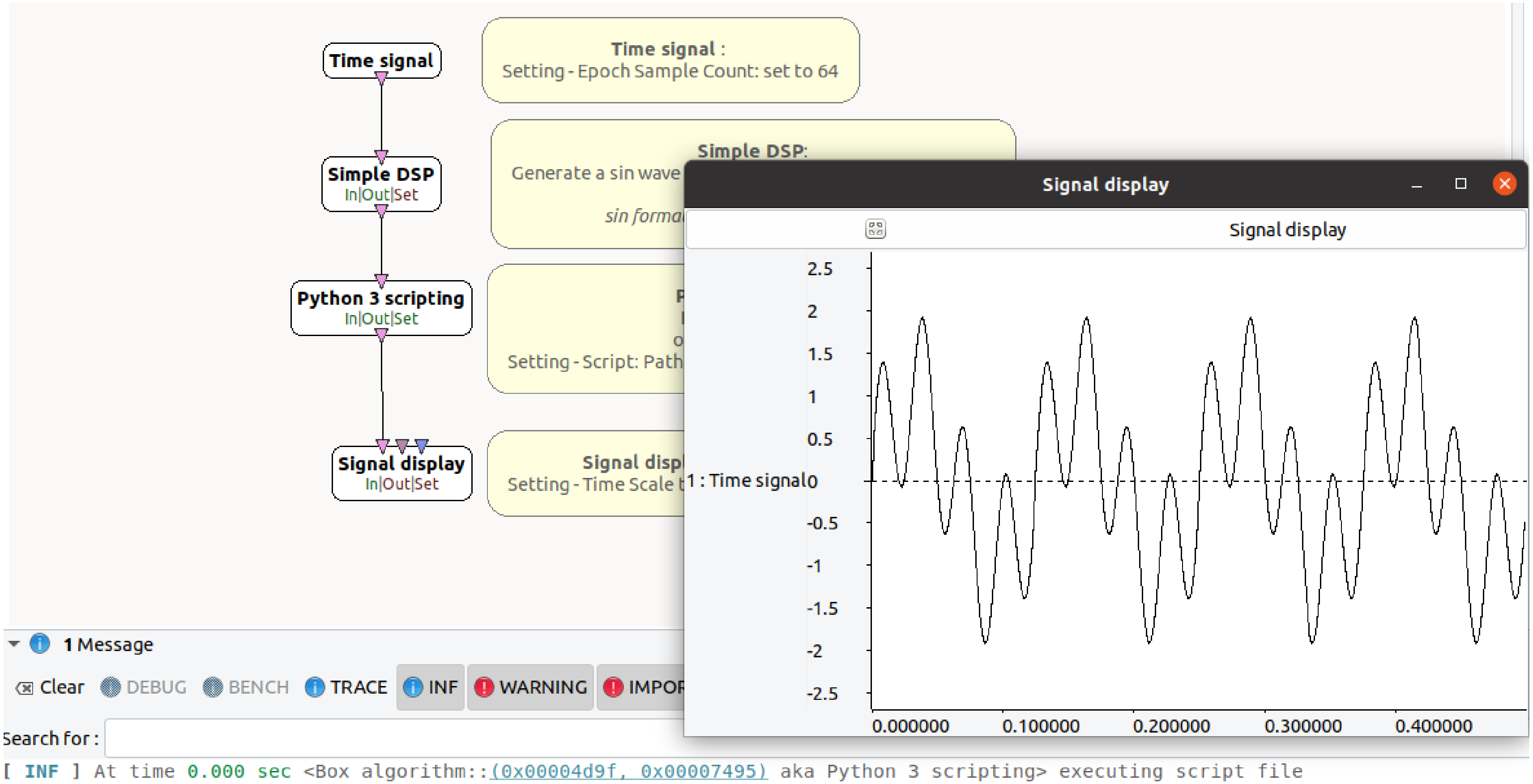
```
def uninitialize(self):
    pass
```

Tips : You can use print() in order to debug. Anything you print appears in the logs.

Step 1: Create a passthrough script

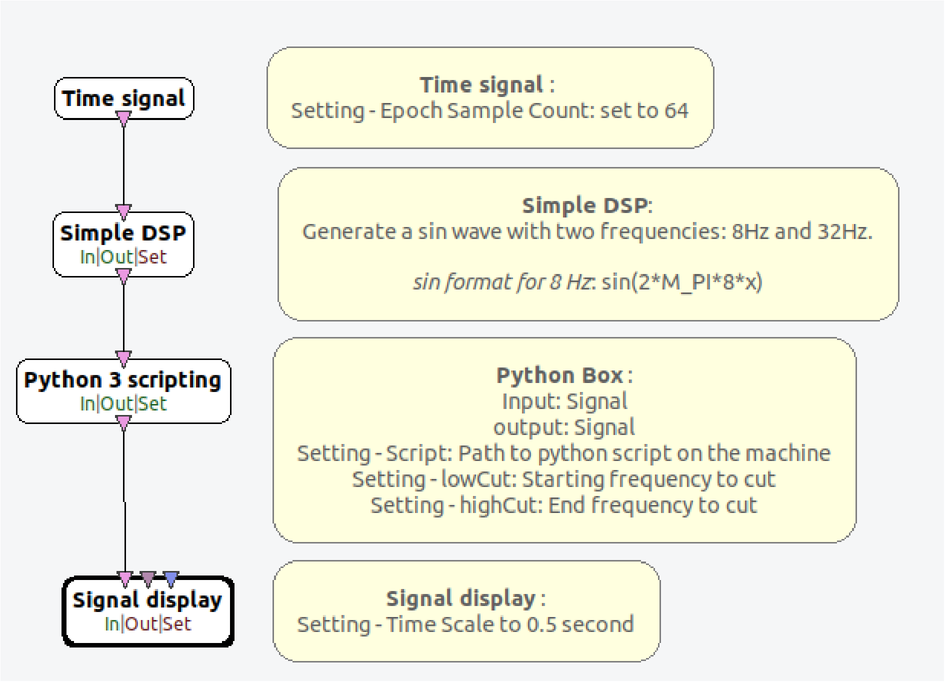
```python
class MyOVBox(OVBox):
    def __init__(self):
        OVBox.__init__(self)

    def initialize(self):
        # Declare signal header
        self.signalHeader = None

    def process(self):
        for chunkIdx in range( len(self.input[0]) ):
            if(type(self.input[0][chunkIdx]) == OVSignalHeader):
                self.signalHeader = self.input[0].pop()

                # Output the same header for signal
                self.output[0].append(self.signalHeader)

            elif(type(self.input[0][chunkIdx]) == OVSignalBuffer):
                chunk = self.input[0].pop()

                # Output signal
                outChunk = OVSignalBuffer(chunk.startTime, chunk.endTime, chunk)
                self.output[0].append(outChunk)

            elif(type(self.input[0][chunkIdx]) == OVSignalEnd):
                self.output[0].append(self.input[0].pop())

box = MyOVBox()
```

Step 1: Create a passthrough script

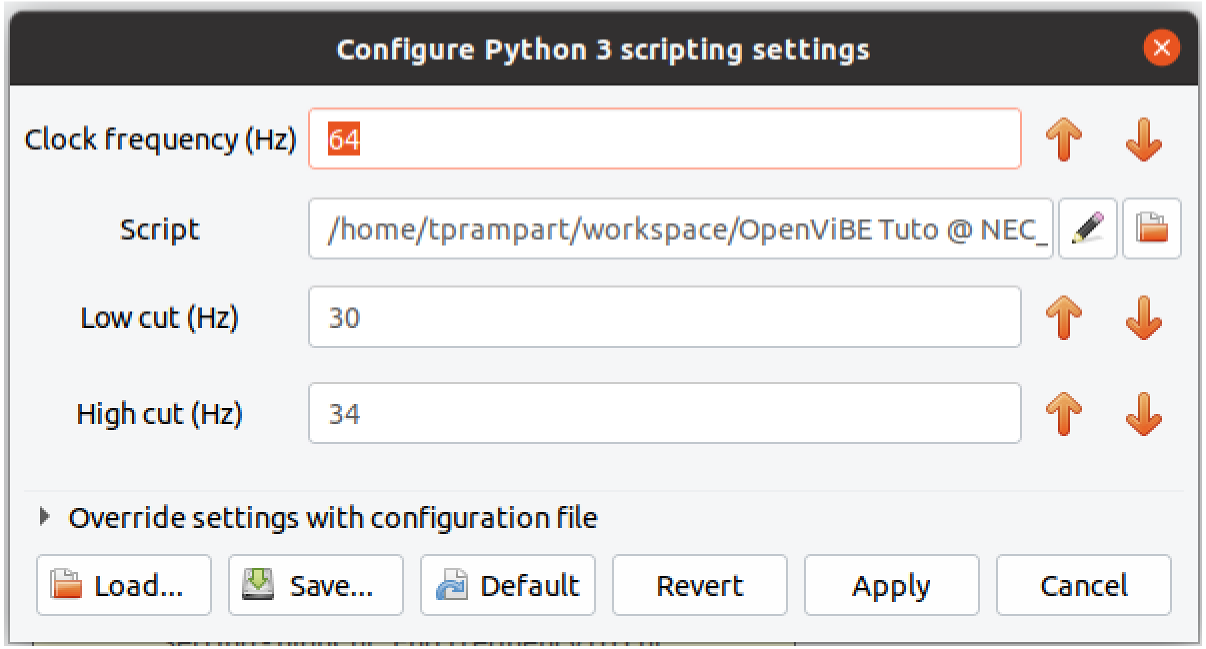


Step 2: Frequency band cut

- Goal : Remove the 32 Hz component of the signal
- Box update: Add lowCut and highCut Settings to python box to provide a range of frequencies to remove



Step 2: Frequency band cut



Step 2: Frequency band cut

- Use FFT functionalities from numpy:
  - *install numpy in terminal: pip3 install numpy*
  - *Useful functions : numpy.fft.fft(), numpy.fft.fftfreq() & numpy.fft.ifft()*

How does a FFT work :

- Estimates spectrum of signal
- The amount of frequency bins in the spectrum depends on the amount of samples processed :

$$FFT_{bins} = \frac{N_{Samples}}{2}$$

- With a sample count of 64, we will have 32 freq. bins
- The actual spectrum will have 64 bins, but split into 32 positive bins and 32 negative bins which mirror each other.

Step 2: Frequency band cut

```python
def initialize(self):
    # Initialize parameters for frequency cuts:
    # self.lowCut = ...
    # self.highCut = ...
```

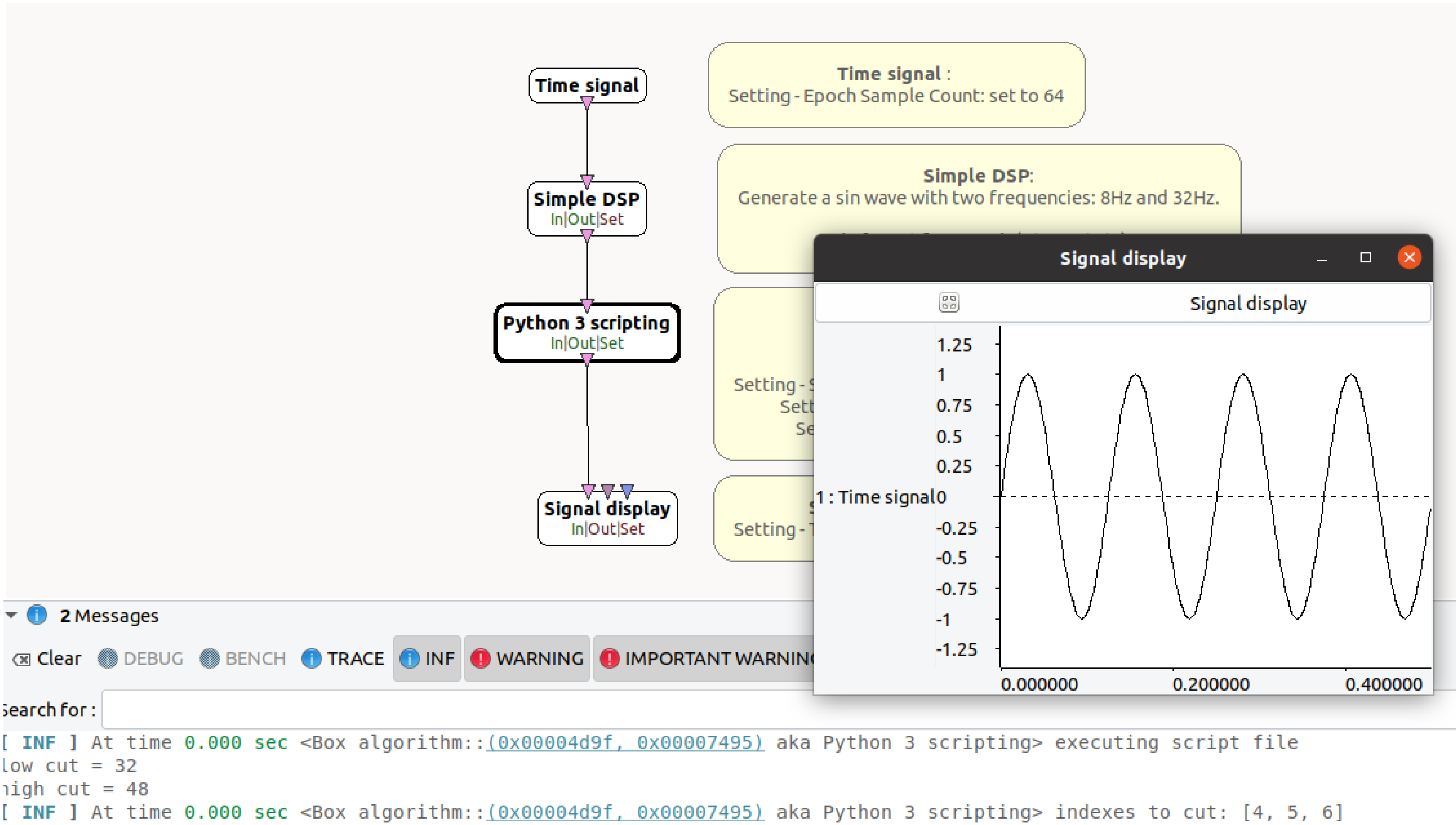
```python
def process(self):
    # Loop through input 1 chunks
        # Check if chunk is Header, Buffer or End
        # If Header:
            # Define FFT frequency bins from Header using the number of samples provided by dimensionSizes member.
            # Filter positive frequencies
            # Establish frequency bins indexes that need removing
            # Forward Header
        # If Buffer:
            # Process FFT on Buffer
            # Remove frequency bins needed
            # Process inverse FFT
            # Output filtered chunk on output 1
        # If End:
            # Forward End
```

Step 2: Frequency band cut

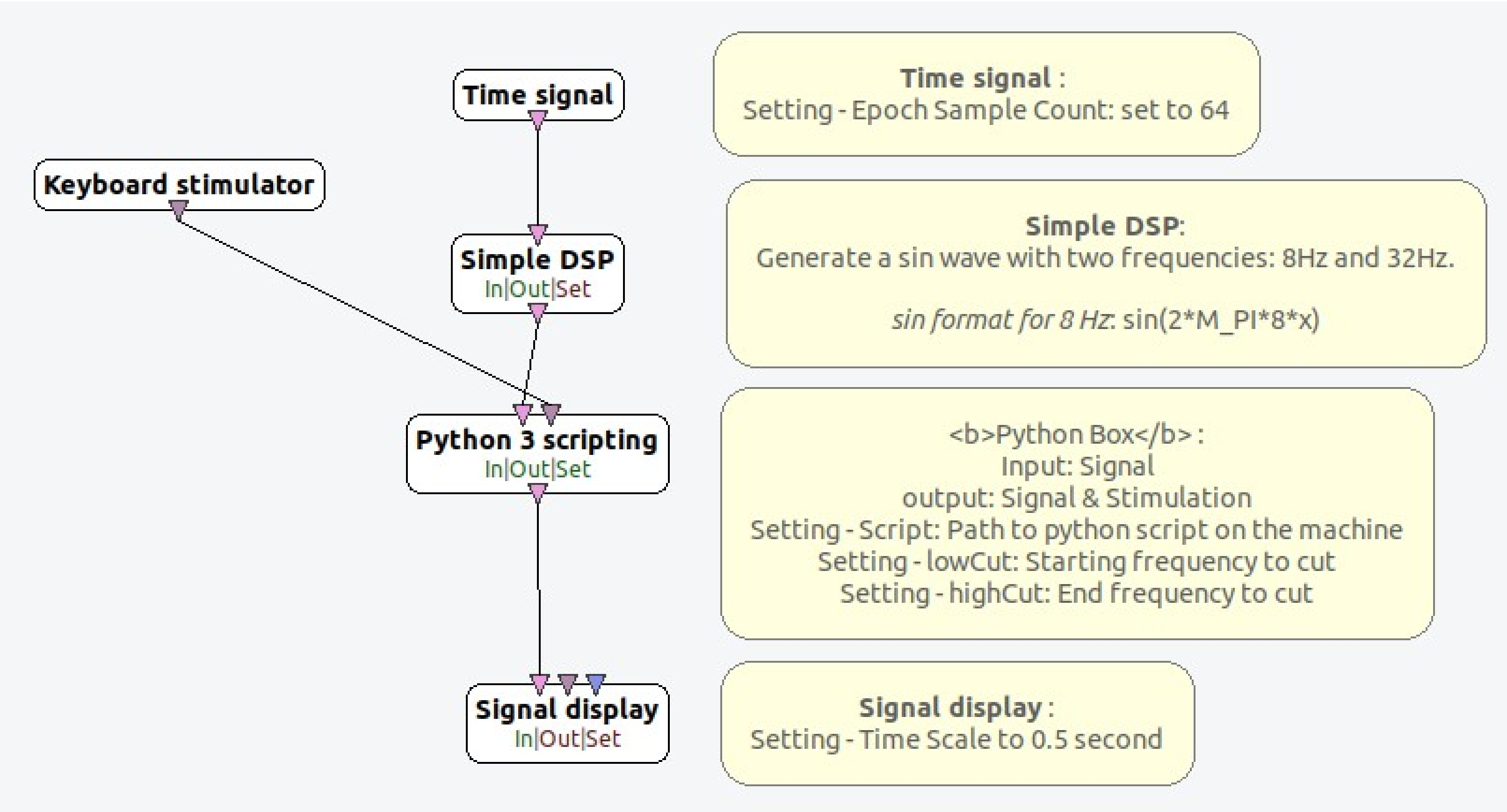
```python
def initialize(self):
    # Declare signal header
    self.signalHeader = None

    # Initialize parameters for frequency cuts:
    self.lowCut = int(self.setting['Low cut (Hz)'])
    self.highCut = int(self.setting['High cut (Hz)'])
    print(f"low cut = {self.lowCut}")
    print(f"high cut = {self.highCut}")

    # initialize FFT frequency bins indexes to cut
    self.freqIdxToCut = []
```

```python
def process(self):
    for chunkIdx in range( len(self.input[0]) ):
        if(type(self.input[0][chunkIdx]) == OVSignalHeader):
            self.signalHeader = self.input[0].pop()

            # Initialize frequency bins and initialize which will be cut
            freq = np.fft.fftfreq(self.signalHeader.dimensionSizes[1])
            freq = [f * self.signalHeader.samplingRate for f in freq[:len(freq)//2]]

            for idx, f in enumerate(freq):
                if int(f) >= self.lowCut and int(f) <= self.highCut:
                    self.freqIdxToCut.append(idx)
            print(f"indexes to cut: {self.freqIdxToCut}")

            # Output the same header for signal
            self.output[0].append(self.signalHeader)

        elif(type(self.input[0][chunkIdx]) == OVSignalBuffer):
            chunk = self.input[0].pop()

            # Process FFT: numpy.fft.fft()
            fft = np.fft.fft(chunk)

            # Remove frequencies
            for i in self.freqIdxToCut:
                fft[i] = 0.0
                fft[-i] = 0.0

            # Inverse FFT: numpy.fft.ifft()
            filteredSignal = np.fft.ifft(fft)

            # Output signal
            outChunk = OVSignalBuffer(chunk.startTime, chunk.endTime, filteredSignal)
            self.output[0].append(outChunk)

        elif(type(self.input[0][chunkIdx]) == OVSignalEnd):
            self.output[0].append(self.input[0].pop())
```

Step 2: Frequency band cut



Step 3: Control band cut



Step 3: Control band cut

Code addition for initialize and process methods.

```
def initialize(self):
    # Initialize flag to activate the process filtering
```

```
def process(self):
    # Loop through input 2 chunks
        # Activate filtering when receiving stim ID 33025 (keyboard 'a')
        # Deactivate filtering when receiving stim ID 33026 (keyboard 'z')
```

Finally: You can the use the flag to either send filter the signal or just pass it through

Step 3: Control band cut

```python
def initialize(self):
    # Initialize flag to activate the process filtering
    self.filterOn = False

    # Declare signal header
    self.signalHeader = None

    # Initialize parameters for frequency cuts:
    self.lowCut = int(self.setting['Low cut (Hz)'])
    self.highCut = int(self.setting['High cut (Hz)'])
    print(f"low cut = {self.lowCut}")
    print(f"high cut = {self.highCut}")

    # initialize FFT frequency bins indexes to cut
    self.freqIdxToCut = []
```

```python
def process(self):
    for chunkIdx in range( len(self.input[1]) ):
        if(type(self.input[1][chunkIdx]) == OVStimulationSet):
            stimSet = self.input[1].pop()
            for stim in stimSet:
                if stim.identifier == 33025:
                    self.filterOn = True
                if stim.identifier == 33026:
                    self.filterOn = False

    for chunkIdx in range( len(self.input[0]) ):
        if(type(self.input[0][chunkIdx]) == OVSignalHeader):
            self.signalHeader = self.input[0].pop()
```

```python
elif(type(self.input[0][chunkIdx]) == OVSignalBuffer):
    chunk = self.input[0].pop()

    if self.filterOn == False:
        outChunk = chunk
    else:
        # Process FFT: numpy.fft.fft()
        fft = np.fft.fft(chunk)
```

Thank you for your attention !

Thomas Prampart
thomas.prampart@inria.fr

openvibe.inria.fr