Creating a Virtual Cognitive Eye Model with Artificial Leaky-Fire-Integrate Neurons Organised in Complex Networks

Introduction

One of the biggest mysteries of the 21st century is the understanding of the human brain. How is it able to process information so complexly and yet see, hear, smell, feel and visualize itself seamlessly? How can it think, move and react to its surroundings in less than an instant? These specific mechanisms are known as cognitive control (Gu, et al., 2015). Multiple endeavours are currently being carried out to understand this phenomenon, as well as replicate it in order to create machines with cognitive abilities.

Thus, the understanding of the brain has two major objectives. The first is to produce an accurate map of the brain in order simulate accurately brain dynamics while carrying out actions as well as the brain's behaviour when infected by a disease. This goal is being carried by the human connectome project (Shi & Toga, 2017) and the human brain project (Markram, et al., 2011), which intend to create a brain map to understand how the brain processes information, makes decisions and acts. The second endeavour consists of building artificial brains to create an artificial intelligence (A.I.) able to steer and understand their system dynamics intuitively like the humans and animals (Pan, 2016).

Recent progress in the fields of deep learning and computer vision brought new possibilities in the fields of AI and neuroscience. On the one hand, deep learning allowed us to create new bio-inspired algorithms inspired from how neurons process information. Neurons first receive incoming signals from dendrites, process it in their cellular bodies and respond via their axon. The strengths of this process are harnessed through the design of artificial neural networks (ANN) with one input, one output and one or more hidden layers. Each layer contains a finite number of nodes with each being connected to all nodes in the following layer. Once the output is found it is evaluated by a cost function and fed back into the network inputs this will then cause the alteration of the weights of the connections between nodes (Patterson & Gibson, 2017). This computing framework allows a machine to learn features driving the computation of the results. On the other hand, computer vision enables us to build systems able to use camera vision to navigate environments by using image filters and convolutional neural networks to process incoming optical flow information from a camera. This way the robot would be able to navigate the environment by simply seeing what is around him (Forsyth, et al., 2012).

This paper will attempt to build a computerized model with software written in python able to recreate the brain response of an animal seeing an image. However, instead of using the conventional computer vision and deep learning models, I will try a different approach, which attempts to create artificial vision by designing a visual processing unit (V.P.U.) simulating a complete visual animal apparatus that can observe images by creating a neural response that represents the image itself. To do this I will use two-dimensional (2D) wavelet transforms (W.T.) to extract time series (TS) from the image and use the dominant features of these discrete sequences to stimulate sets of artificial leaky-integral-fire (LIF) neurons.

Methods

Wavelet Transforms

For an ANN to process information, we first need extract features from the data. In this case the data will consist of two images provided by the free to use PyWavelets package online. These two images consist of the aero and camera image in the "pywt.data" data attribute of the software package (PyWavelets Developers, 2017). I then used PyWavelets to create my own customised software wavelet tools, which can be found under the file "WaveletWrapper.py". Features will be extracted from the image with the WT. There are two types of WT the continuous WT (CWT) and the discrete WT (DWT). The CWT is expressed by the following equation:
$W(a,b) = \int_{-\infty}^{\infty} x(t) \frac{1}{\sqrt{|a|}} \Psi * \left(\frac{t-b}{a}\right) dt$, where **b** is the translator or wavelet time window, **a** the time scaler, **x** a function of time **t**, and **Ψ** the mother wavelet or probing function (Semmlow, 2009). Though this concept is somewhat analogue to the Fourier transform (FT), it is not the same. While, the FT describes a signal in terms of frequency and intensity, the WT in contrast can be seen as describing a 2D signal as three-dimensional (3D) signal in terms of intensity, time and frequency resolution. At this point trade-offs come in as the more frequency resolution is available the less time resolution we have and inversely the more time resolution there is the less frequency resolution is present (Semmlow, 2009). Though, the CWT can be computed discreetly, it extracts too many features from images making it computationally inefficient. Thus, the CWT is not appropriate for the task and we will have to resort to the DWT. The DWT is defined by the following equation:
$x(t) = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} d\,(k,l)\, 2^{-\frac{k}{2}} \Psi(2^{-kt} - l)$, where **k** is the time scaler, **l** is the time window, **d** is the wavelet sampler function and **Ψ** the mother wavelet. The DWT's result divides the data into two parts the trend (low pass data) and the fluctuations (high pass data) (Semmlow, 2009). To compress the data as effectively as possible we will use multilevel wavelet decomposition (MWD), which consists of applying the DWT multiple times on the trends to get the clearest signal possible. As I was working with an image or 2D data the 2D MWD was used to smoothen the trend multiple times as much as possible in conjunction with the most complex discrete wavelet possible (Aziz & Pham, 2012). My MWD was done with the Daubechies 38 wavelet as it has the most vanishing moments, which is equivalent to each point where the wavelet's value and it's integral are zero. The more vanishing moments the wavelet has the more precise the MWD will be but the less times the signal can be sampled (Aziz & Pham, 2012). I carried out 2D MWD to its maximum level. Then, I obtained four sets of TS one for the trend, one for the horizontal fluctuations (HF), one for the vertical fluctuations (VF) and one for diagonal fluctuations (DF) (Aziz & Pham, 2012). The code's output in this case will consist of four NumPy (common python library used for arrays) arrays (special python type ndarray).

Time Series

After obtaining the data from our NumPy arrays we can notice this information consists of a set of sampled TS, where each data point can be represented in the form: $x_{t+1} = x_t + \Delta x_t \equiv x_0 + \sum_{i=0}^{t} \Delta x_i$ (Chatfield, 2004).When plotting the TS set for each fluctuation and trend we notice that all the TS datasets form an envelope with the maximum intensities at the top and minimum intensities at the bottom. I concluded it is then sensible to represent the stimulation of photoreceptor cells with the photons carrying the most energy. It will be assumed the greatest sensory neural response will eclipse all the other sensory responses. In the file "SignalEncoding.py", I coded some simple signal processing tools designed to recreate the TS set envelope with two new TS, with one carrying the minimal light intensities and the other the maximal light intensities. The data must be scaled between zero and one in order for the LIF ANN from the Nengo2.6 library package to accept the input, which has been coded as well in the same file.

The Leaky Integrate and Fire (LIF) Artificial Neuron Model

The python library Nengo2.6 (Eliasmith, 2017), I used to construct my virtual visual apparatus uses LIF neurons to simulate the artificial neuron instead of the classic sigmoid, rectifier or step function used to simulate a neural response (Patterson & Gibson, 2017). In contrast to the classic artificial neuron models, the LIF artificial neurons simulate the biological neural response resulting from biophysical phenomena (Hunsberger & Eliasmith, 2015). The LIF neuron dynamics are governed by the following equations: $\tau_{RC}\frac{dv(t)}{dt} = -v(t) + J(t)$ and $\begin{cases}\left[\tau_{ref} - \tau_{RC}\log\left(1-\frac{V_{th}}{J(t)}\right)\right]^{-1} & if\ j > V_{th}\\ 0\ otherwise & \end{cases}$, where v(t) is the membrane voltage as a function of time, J(t) is the input current, τ_{RC} is the membrane time constant and V_{th} is a constant. This means our model will be realistic and robust, as it will emulate the behaviour pattern of biological neurons (Hunsberger & Eliasmith, 2015).

VPU Design

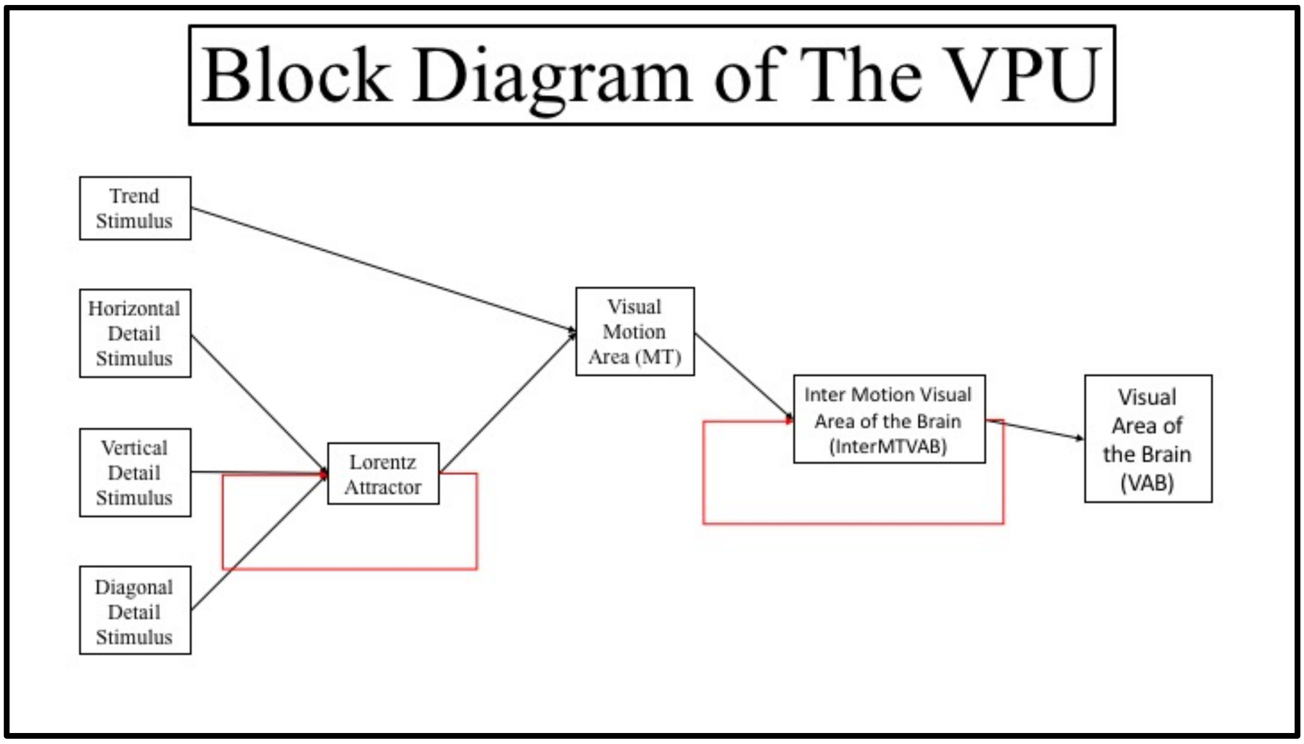

Figure 1 Visual Processing Unit (VPU) Design.

As the 2D MWD and signal encoding give us eight inputs the VPU needs to handle four sets of scaled TS each containing one maximum TS and a minimum TS. This means the TS sets will contain only the "virtual photons" carrying the most energy from the image. The main assumption of this model is that only the "virtual photons" with the most energy will stimulate the photoreceptors. This then results in four virtual stimuli, with one for each set. These stimuli are represented on the block diagram above and are pre-processed by an artificial afferent neural network (AANN) composed of three sets of artificial LIF neurons with two sets acting as the sensory visual cells of the entity, which both have 100 neurons. The third set, which is connected to the two previous neuron sets has 200 neurons and processes the minimum and maximal TS into net signal. The VPU contains four sets of artificial LIF neurons, which each have their own use. The Lorentz Attractor (LA) a set of two thousand LIF neurons and is connected to three AANN, which carry the net stimulus for a detail type from the initial scaled TS. We then have a stimulus for the HF, one for the VF and another for DF each provided by a separate AANN, which are all connected to one of the LA's three channels. The LA has a feedback loop, which introduces chaotic dynamics in the system via the following differential equations: $\begin{cases} \frac{dx_1(t)}{dt} = a(x_2(t) - x_1(t)) \\ \frac{dx_2(t)}{dt} = cx_1(t) - x_1(t)x_3(t) - x_2(t) \\ \frac{dx_3(t)}{dt} = x_1(t)x_2(t) - bx_3(t) \end{cases}$, with the x_1, x_2, x_3, provided by the net TS signals from the three AANN and the free parameters a, b and c respectively set to 10, 8/3 and 28 (Li & Ge, 2009). The LA can be seen as the chaotic movement of the animal, human or entity, which changes constantly the way the image is seen. The model then assumes only a perception of the image is seen rather than the whole image, which mean the VPU will infer the image content instead of keeping the whole image in its "memory". Another assumption is that the details of the image are sampled chaotically, while the trend remains static during the process and is always acquired by the visual system as it contains the images dominant features. The data originating from the LA is then combined with the trend in what can be seen as the area controlling the way the visual apparatus moves (MT on the diagram and in the code). In the case of humans, this can be seen as the area of the brain controlling the way the muscles of the eyes move. These informations are then processed by interneurons, which filter the data. In the case of my VPU model, the filtering effect is simulated with an integrator consisting of a feedback loop in the inter motion-visual area (InterMTVAB on the diagram and in the code) of the entity's brain. Finally, this data is given to the visual area of the artificial brain (VAB), where the image inference is stocked in the memory.

Results

WaveWrapper.py:

The figures below showcase the outputs of the file WaveWrapper.py's main.

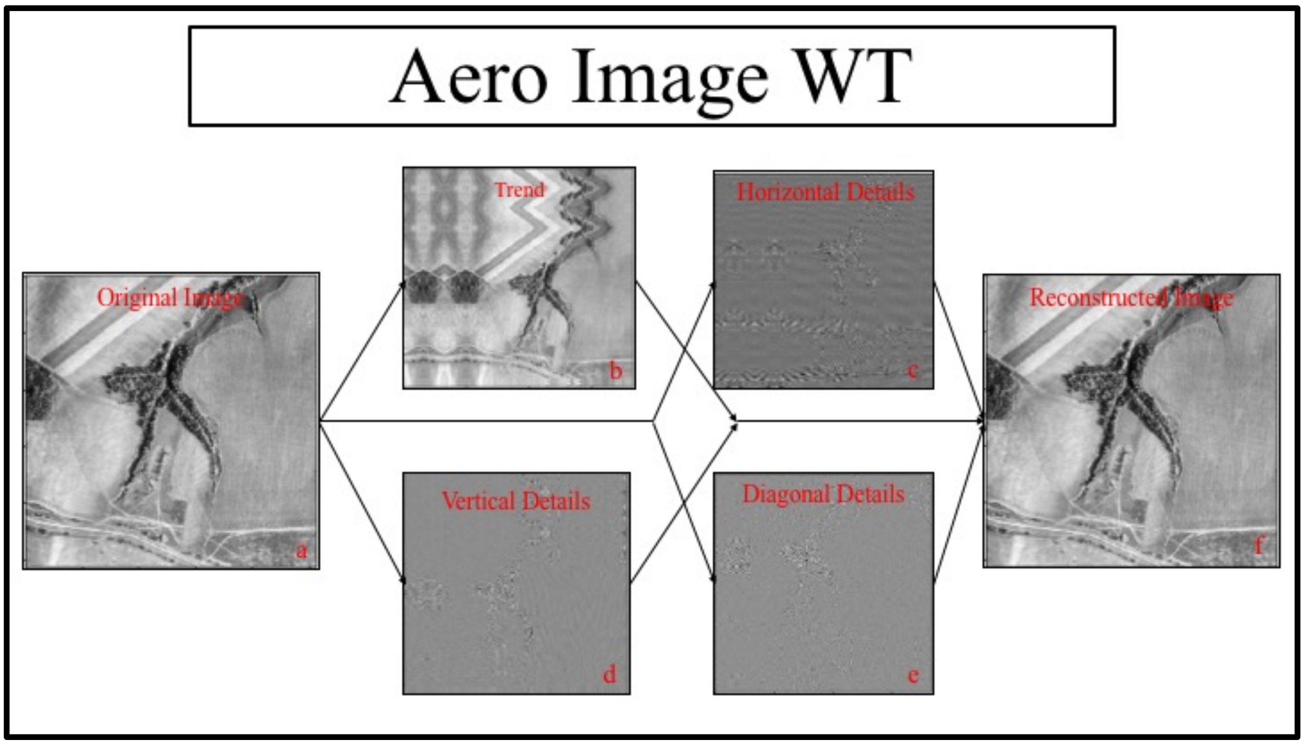

Figure 2 2D MWD of the pywt.data.aero() image. (a) Original Aero Image. (b) Aero Image Trends. (c) Aero Image HF, which is the same as horizontal details (HD). (d) Aero Image VF, which is the same as Vertical Details (VD). Aero Image DF, which is the as Diagonal Details (DD). (f) Reconstructed Aero Image with 2D multilevel wavelet reconstruction (MWR).

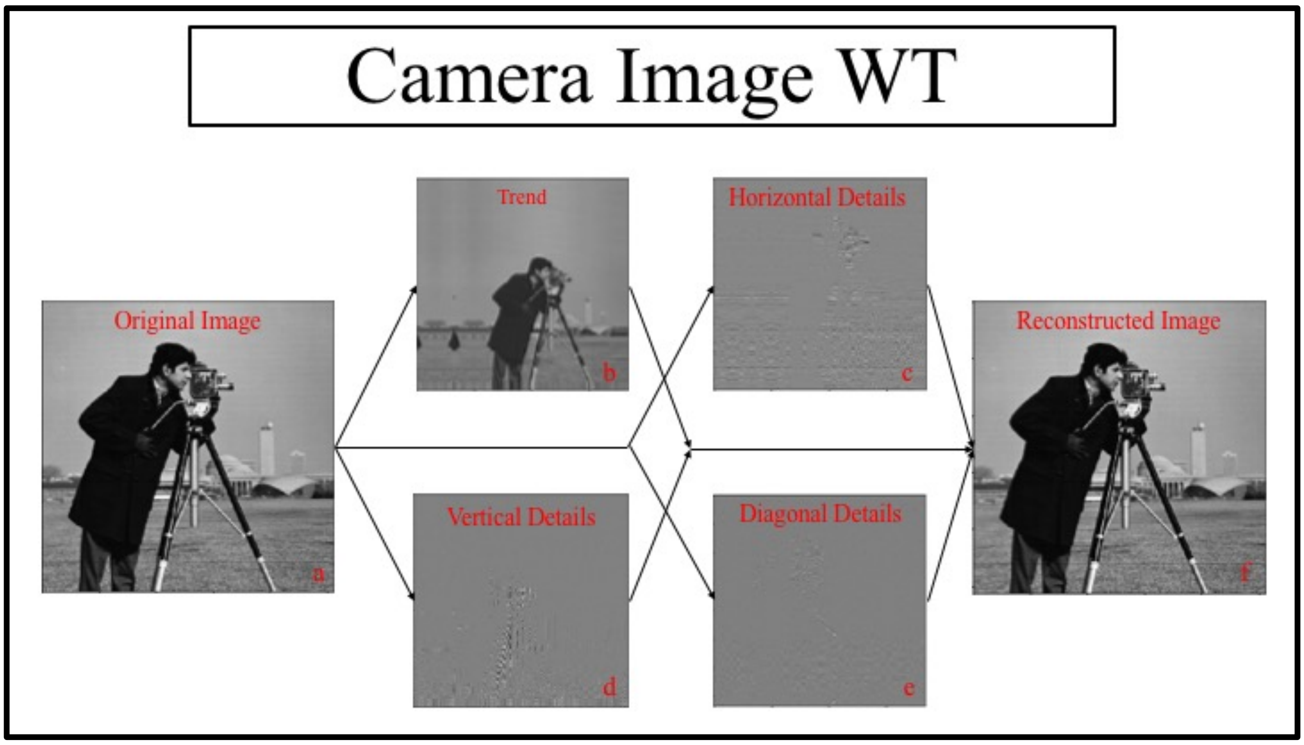

Figure 3 2 MWD of the pywt.data.camera() image. (a) Original Camera Image. (b) Camera Image Trends. (c) Camera Image HF. (d) Camera Image VF. Camera Image DF. (f) Reconstructed Camera Image with 2D MWR.

The main in the code was designed to showcase the effects of the 2D MWD. Via the MWD each image, has been converted into a set of TS with each TS containing features of the image. I chose to give two examples of wavelet analysis as it allows to later compare the outputs from the VPU. Furthermore, this also proves an image can be represented as well as reconstructed with MWR.

SignalEncoding.py:

The figures below showcase the outputs of the file SignalEncoding.py's main.

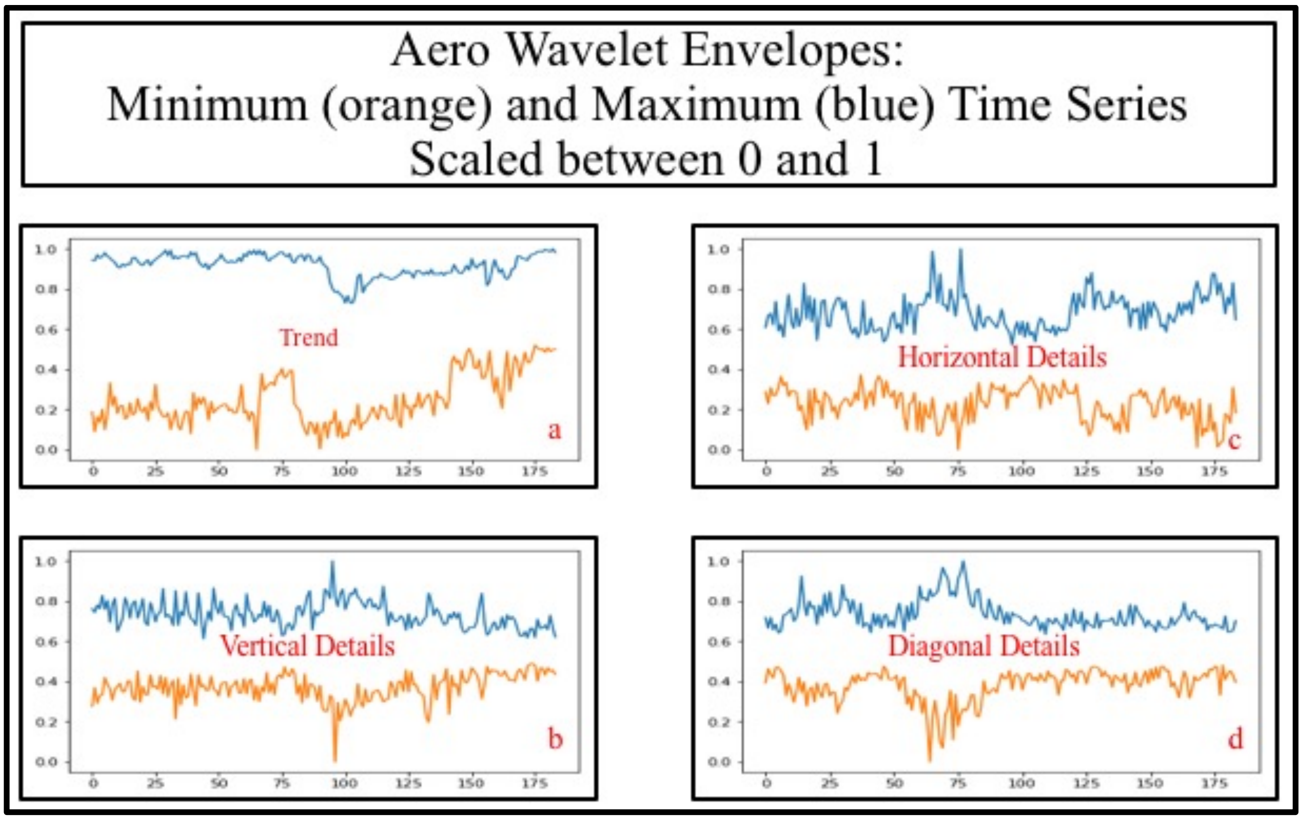

Figure 4 Scaled Aero TS Sets Envelopes. (a) Scaled Aero Trend TS Envelope. (b) Scaled Aero VF TS Envelope. (c) Scaled Aero HF TS Envelope. (d) Scaled Aero DF Envelope.

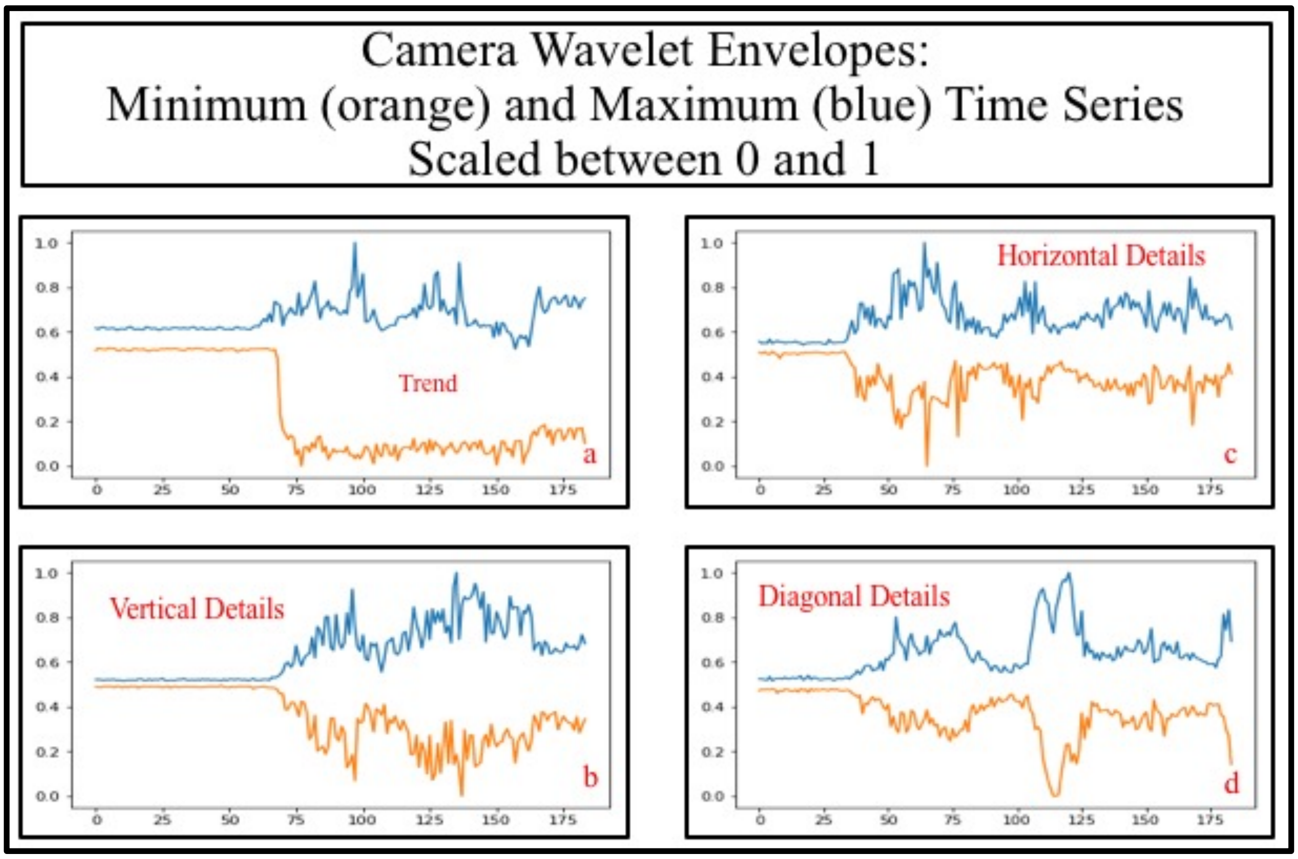

Figure 5 Scaled Camera TS Sets Envelopes. (a) Scaled Camera Trend TS Envelope. (b) Scaled Camera VF TS Envelope. (c) Scaled Camera HF TS Envelope. (d) Scaled Aero DF Envelope.

The strongest virtual photons in the images have been harnessed by finding the photons with the most "negative energy" (TS set minima) and the ones with the most "positive energy" (TS set maxima). In the methods section, it is assumed that the "virtual photoreceptors" are represented as LIF artificial neurons triggered by the photons with the most energy. For this reason, all the other photons of the TS sets were neglected. Furthermore, it would be highly impractical to do a full image simulation on my laptop as this would take too much time (each AANN simulation takes about two minutes to complete and 2D MWD gives us more than a hundred TS). We can notice as well that the TS patterns showcased in both images are not the same, which means we can conclude that two different images are represented by two distinct TS sets.

NeuralVisualProcessingUnit.py:

The figures below showcase the outputs of the file NeuralVisualProcessingUnit.py's main.

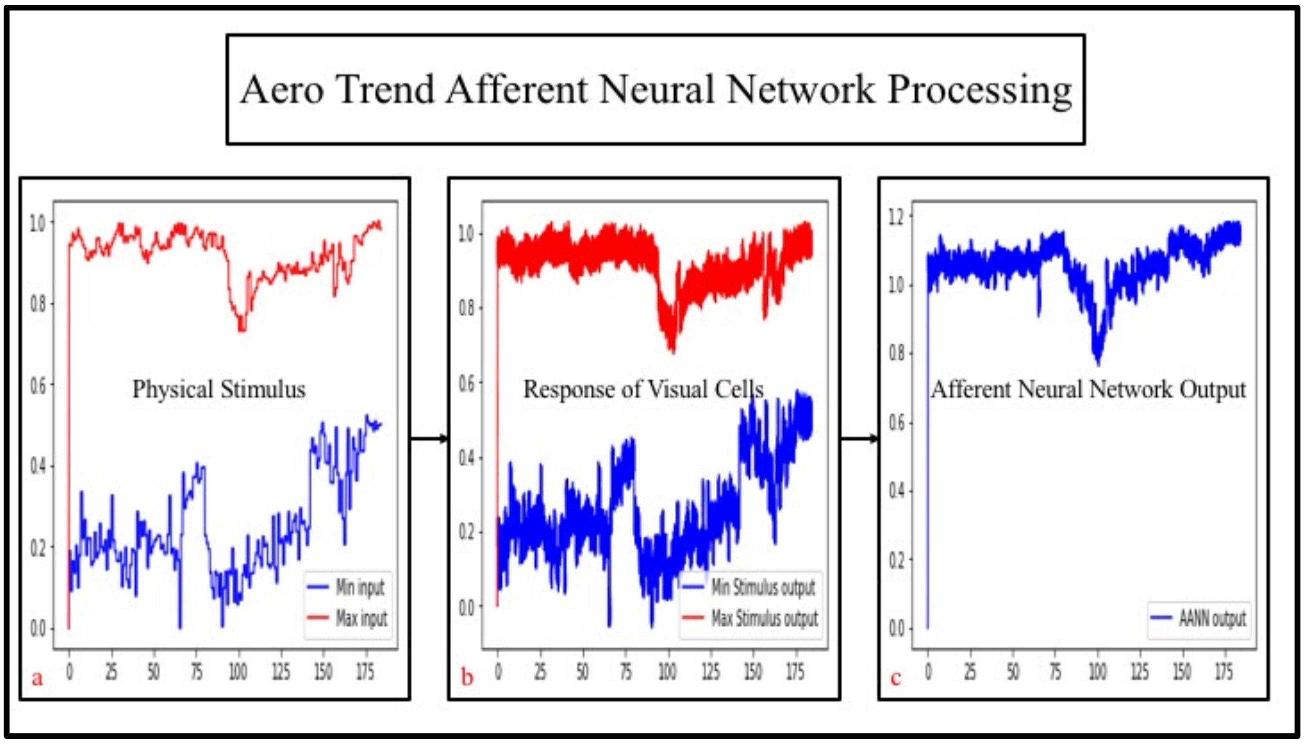

Figure 6 Data Processing of the Aero Image's strongest Trends. (a) Physical Stimulation Input corresponding to the Aero Image's Trends. (b) Photoreceptor Response to the Aero Image's Trends. (c) AANN Response to the Aero Image's Trends.

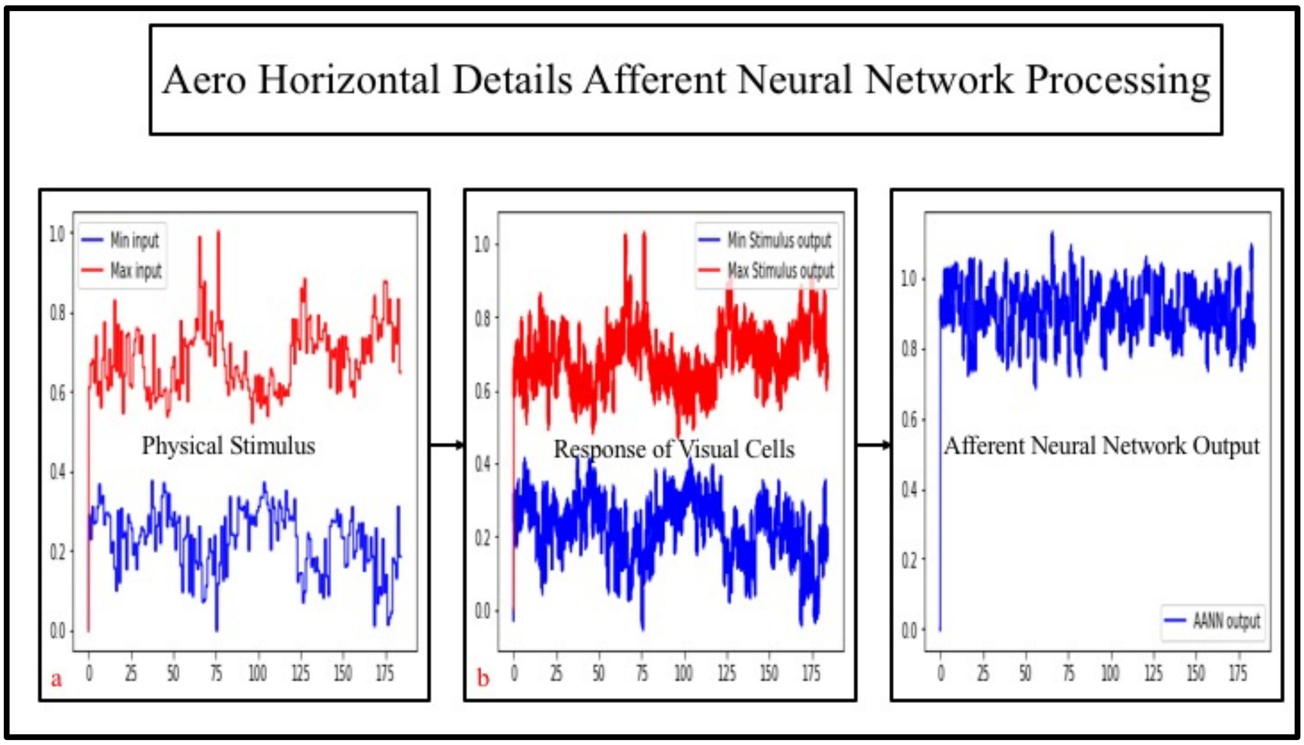

Figure 7 Data Processing of the Aero Image's strongest HFs. (a) Physical Stimulation Input corresponding to the Aero Image's HFs. (b) Photoreceptor Response to the Aero Image's HFs. (c) AANN Response to the Aero Image's HFs.

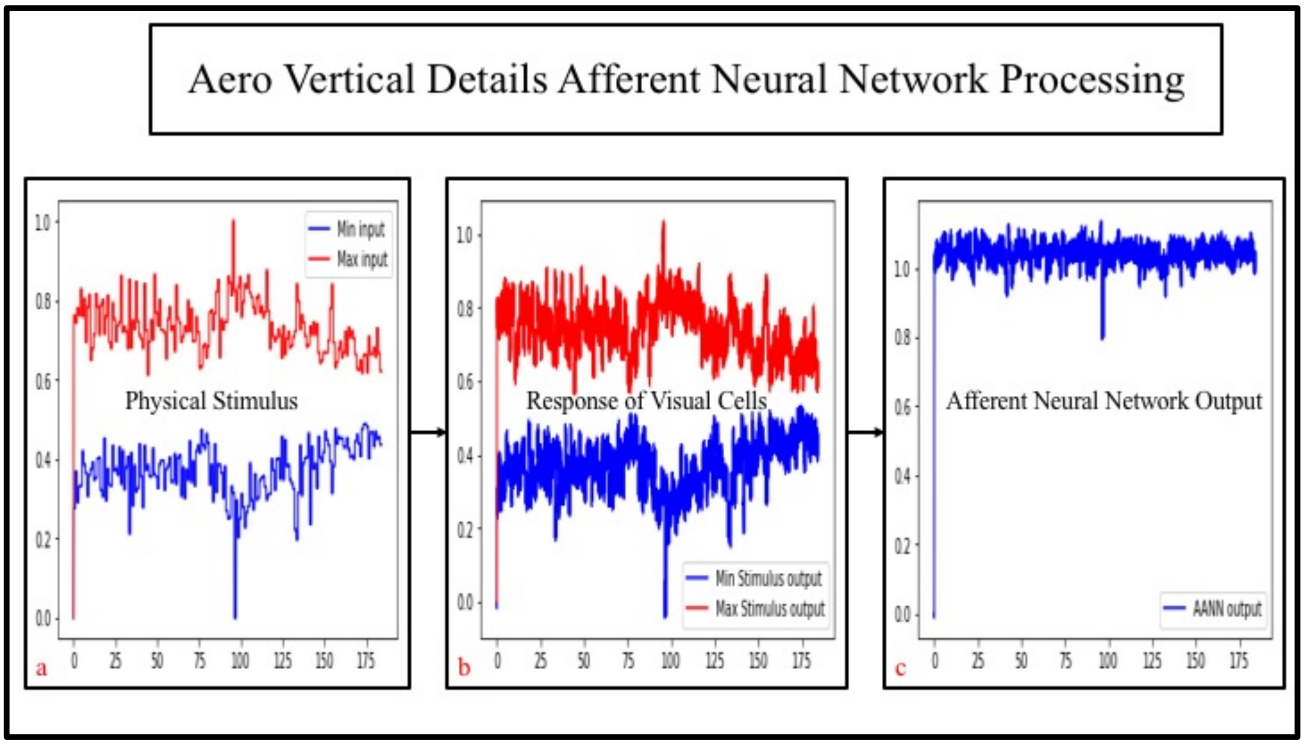

Figure 8 Data Processing of the Aero Image's strongest VFs. (a) Physical Stimulation Input corresponding to the Aero Image's VFs. (b) Photoreceptor Response to the Aero Image's VFs. (c) AANN Response to the Aero Image's VFs.

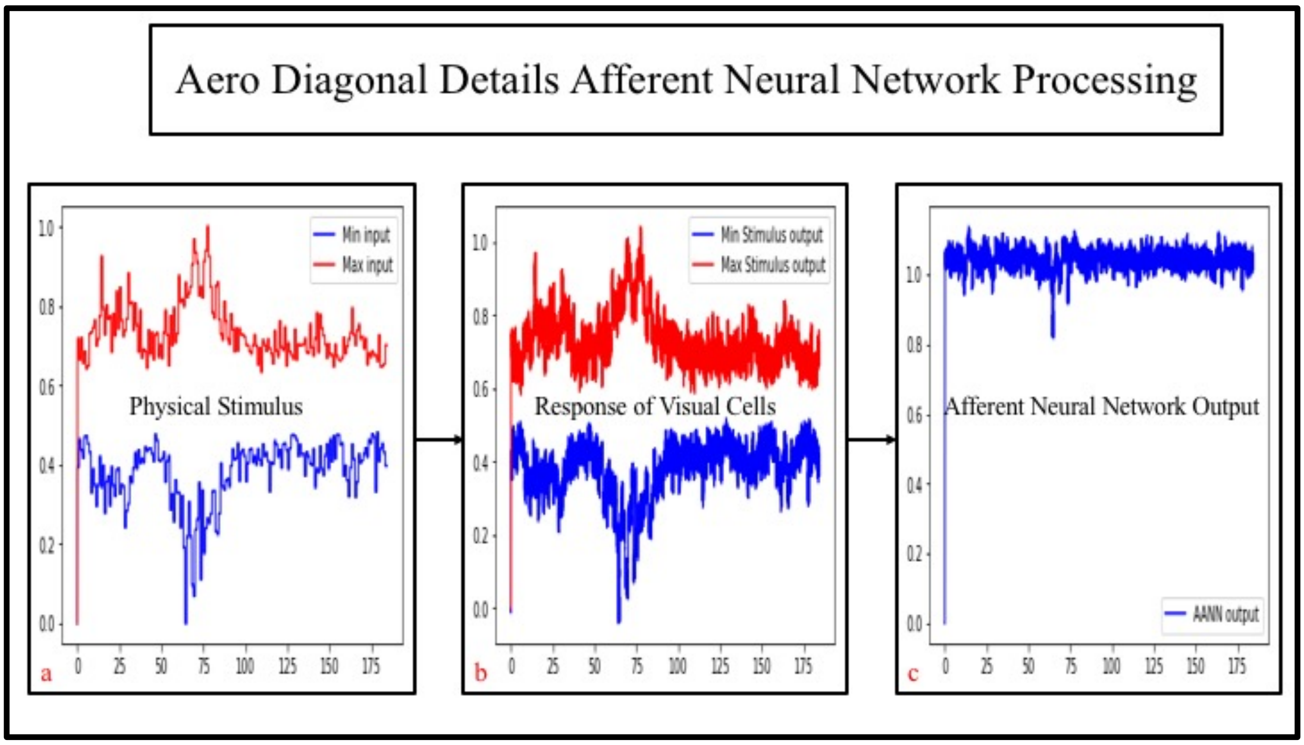

Figure 9 Data Processing of the Aero Image's strongest DFs. (a) Physical Stimulation Input corresponding to the Aero Image's DFs. (b) Photoreceptor Response to the Aero Image's DFs. (c) AANN Response to the Aero Image's DFs.

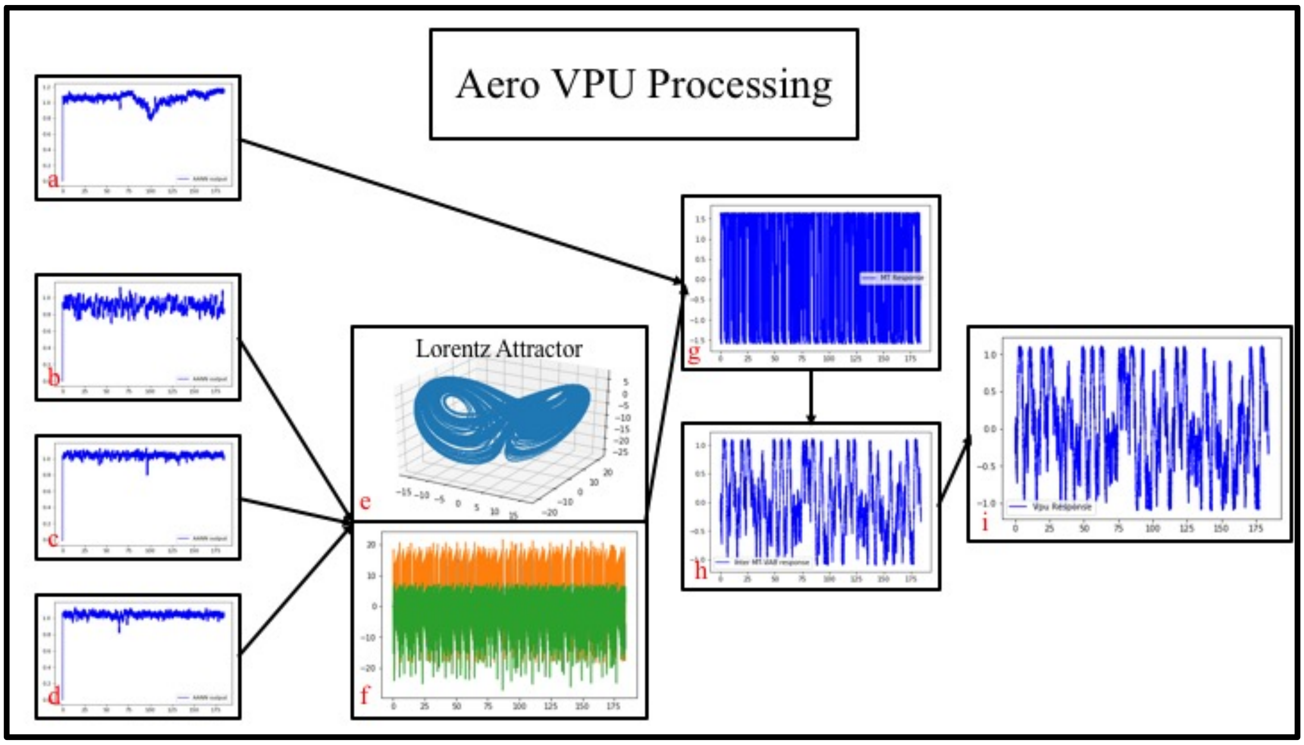

Figure 10 Data Processing of the Aero Image. (a) AANN Response to the Aero Image's Trends. (b) AANN Response to the Aero Image's HFs. (c) AANN Response to the Aero Image's VFs. (d) AANN Response to the Aero Image's DFs. (e) Aero Image LA Response 3D Plot. (f) Aero Image LA Response 2D Plot. (g) Aero Image Motion Area (MT) Response. (g) Aero Image Motion InterMTVAB Response. (i) Aero Image VAB Response, which is the VPU Response.

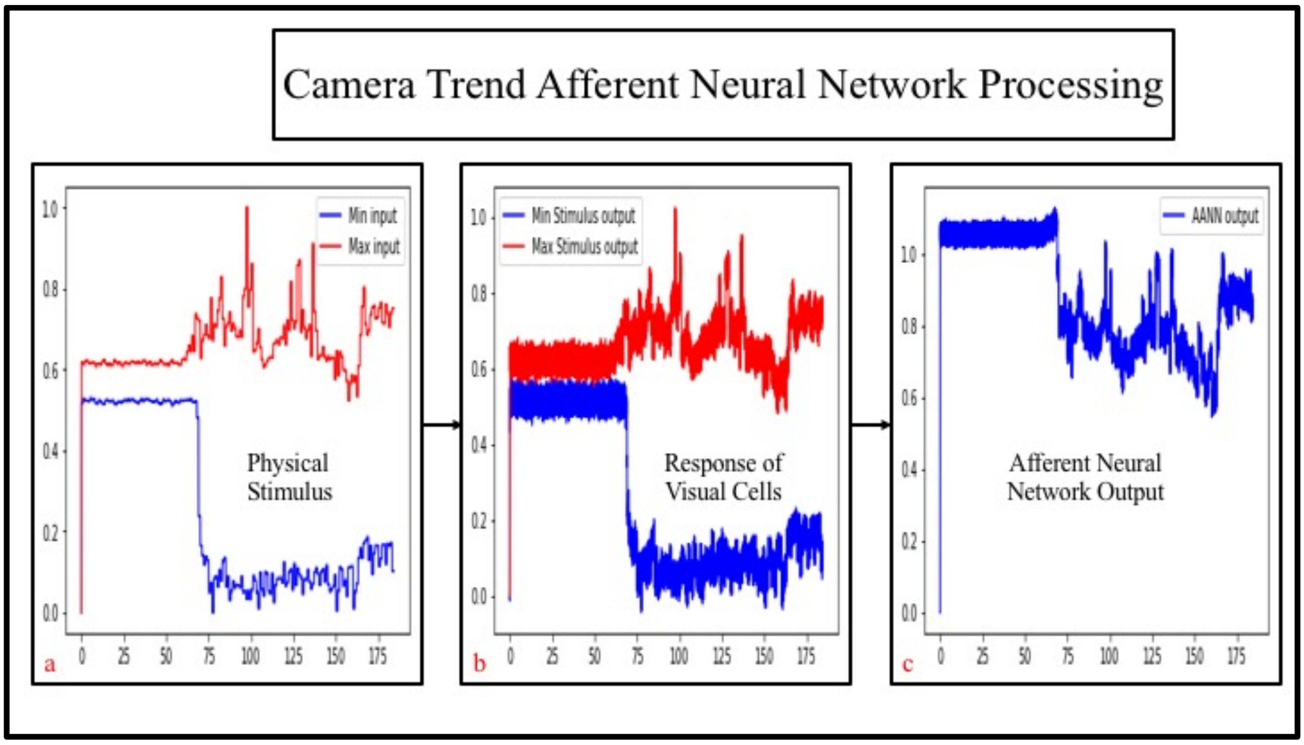

Figure 11 Data Processing of the Camera Image's strongest Trends. (a) Physical Stimulation Input corresponding to the Camera Image's Trends. (b) Photoreceptor Response to the Camera Image's Trends. (c) AANN Response to the Camera Image's Trends.

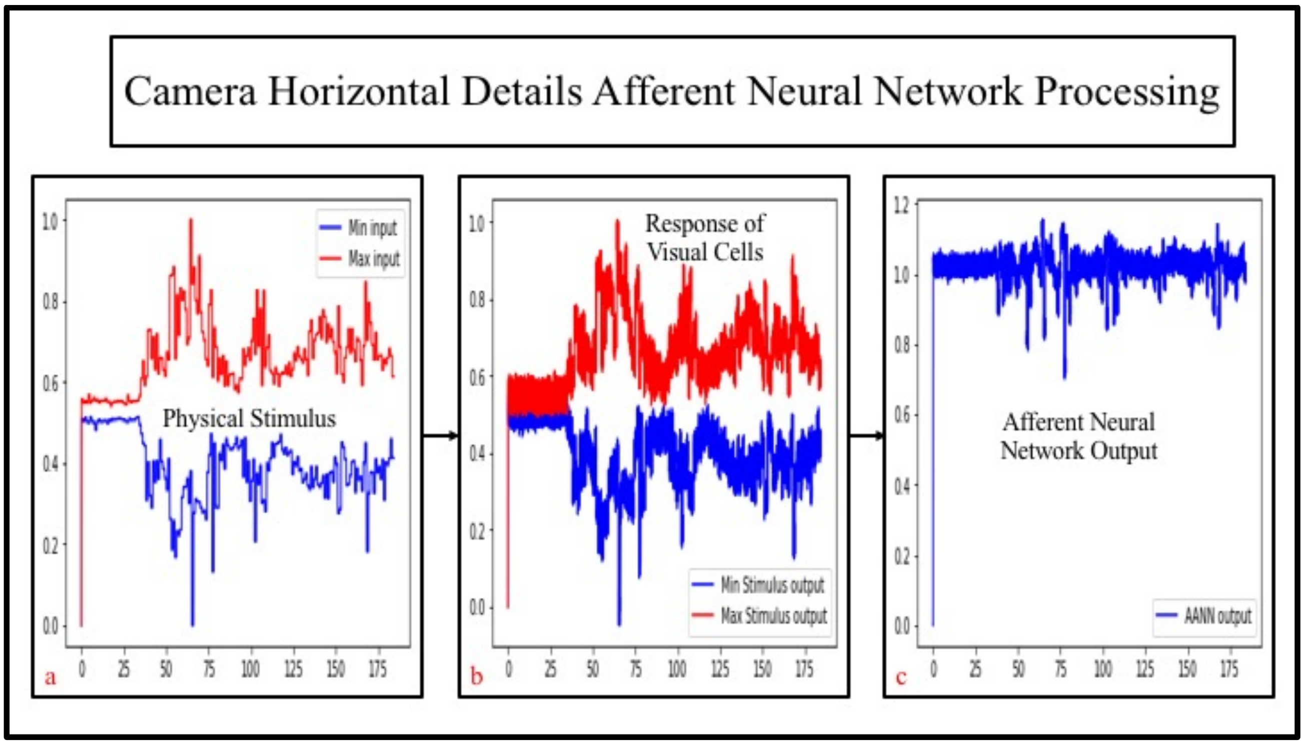

Figure 12 Data Processing of the Camera Image's strongest HFs. (a) Physical Stimulation Input corresponding to the Camera Image's HFs. (b) Photoreceptor Response to the Camera Image's HFs. (c) AANN Response to the Camera Image's HFs.

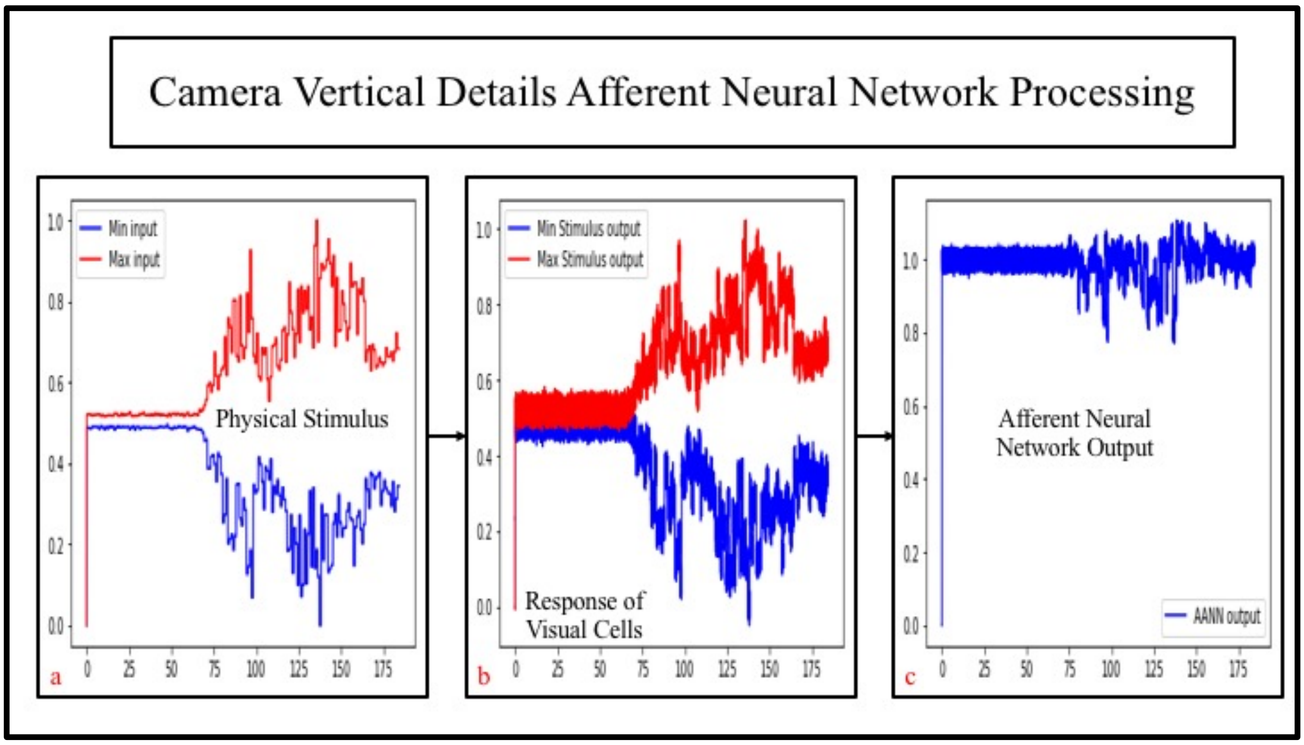

Figure 13 Data Processing of the Camera Image's strongest VFs. (a) Physical Stimulation Input corresponding to the Camera Image's VFs. (b) Photoreceptor Response to the Camera Image's VFs. (c) AANN Response to the Camera Image's VFs.

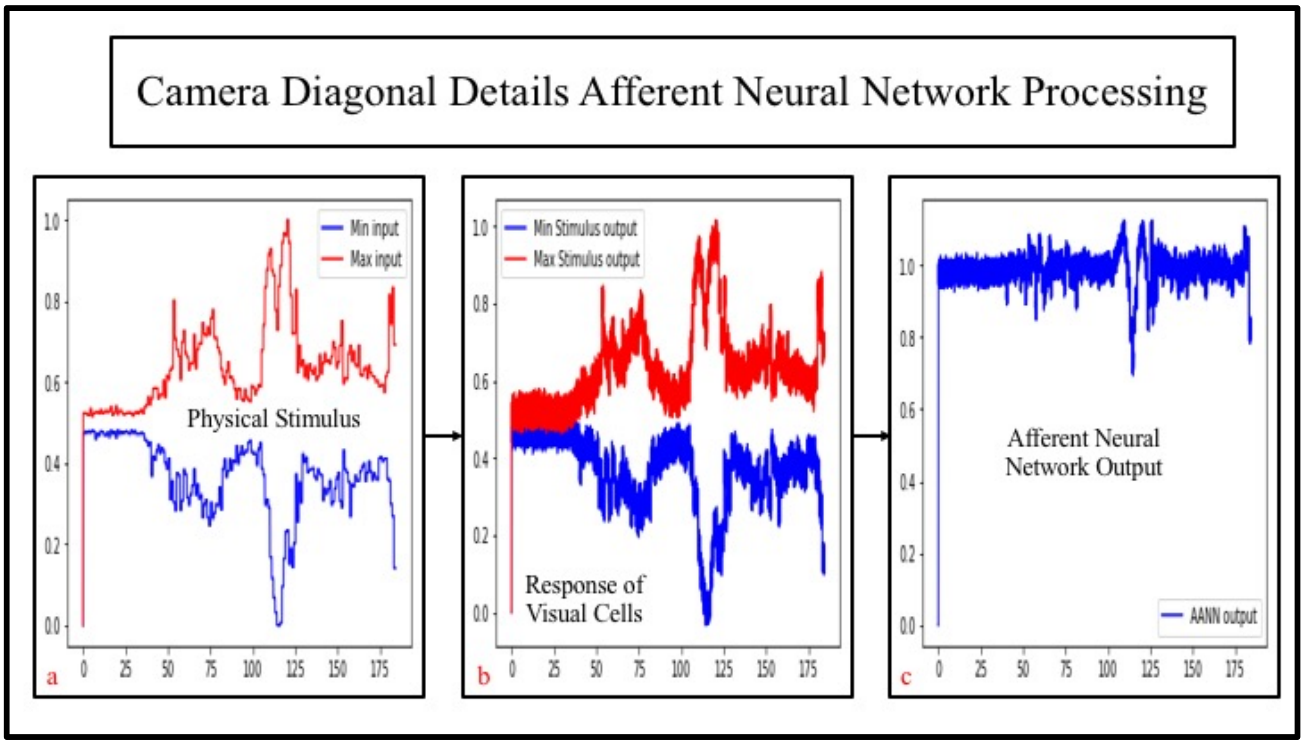

Figure 14 Data Processing of the Camera Image's strongest DFs. (a) Physical Stimulation Input corresponding to the Camera Image's DFs. (b) Photoreceptor Response to the Camera Image's DFs. (c) AANN Response to the Camera Image's DFs.

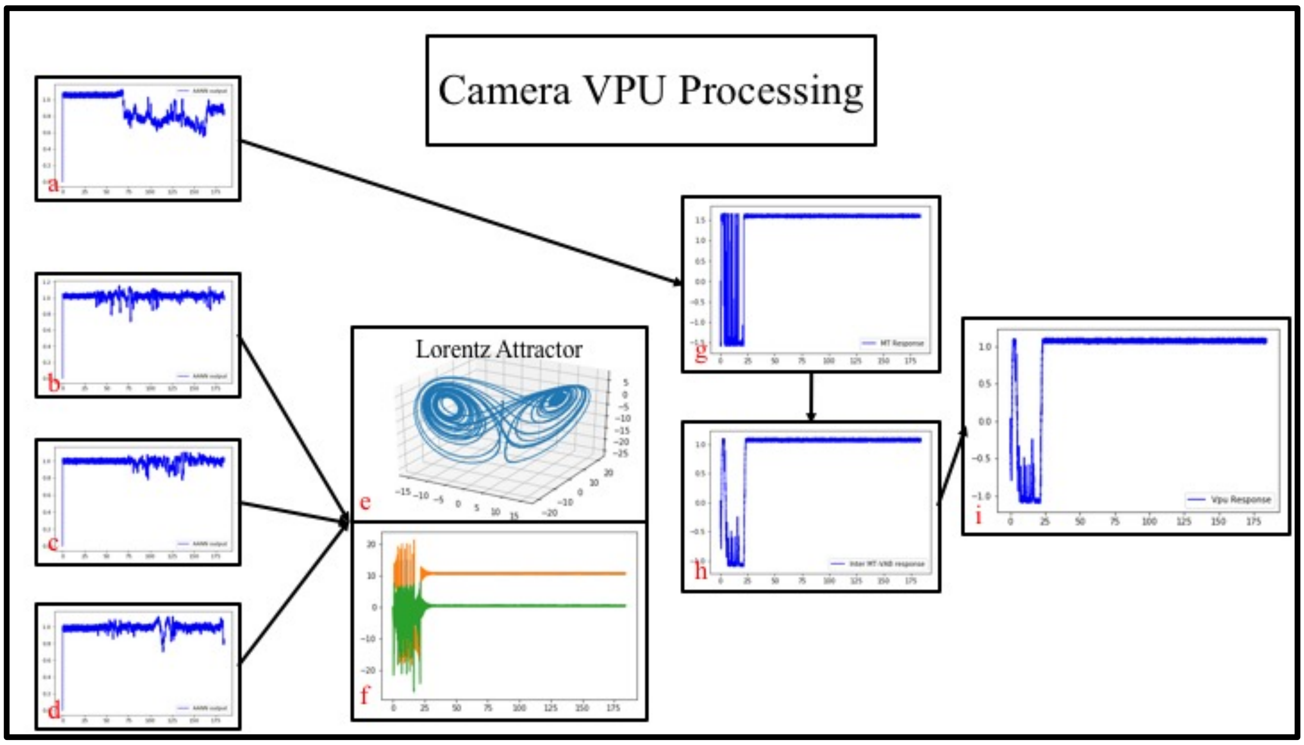

Figure 15 Data Processing of the Camera Image. (a) AANN Response to the Camera Image's Trends. (b) AANN Response to the Camera Image's HFs. (c) AANN Response to the Camera Image's VFs. (d) AANN Response to the Camera Image's DFs. (e) Camera Image LA Response 3D Plot. (f) Camera Image LA Response 2D Plot. (g) Camera Image Motion Area (MT) Response. (g) Camera Image Motion InterMTVAB Response. (i) Camera Image VAB Response, which is the VPU Response.

The results prove LIF ANNs process the data accurately as we see that they manage to create each time the appropriate firing patterns corresponding to the input signal as shown by figures 6 (b), 7 (b), 8 (b), 9 (b),11 (b), 12 (b),13 (b) and 14 (b). The complete image processing done with the VPU composed of four AANNs, one LA, one MT, one InterMTVAB and the VAB, prove it is possible to encode a complete image with chaotic visual data sampling. We can also notice a time window on the final results, which is the time gap before the system reaches steady-state on figure 15 (i), which suggests it may be possible to track the details the system is acquiring within this time lapse. Furthermore, the results prove it is possible to simulate looking at an image non-linearly, while combining the image's details with the main features in the image. The detail sampling is chaotic and therefore the results are not reproducible. Note the main of the NeuralVisualProcessingUnit.py file takes between ten and twenty minutes to run.

Discussion

The software managed to create a model able to process an image with TS. Standard deep learning and computer vision techniques can be very robotic at times, which means they necessarily each time result in the same outcome. Humans and animals do not always react to a visual situation with same sequence of actions. By chaotically sampling the details inside an image and adding the information to the trend. I managed to create a VPU response creating an inference of the image inside the virtual brain rather than the image itself. I assumed during this process the brain immediately captured the image's main features (trends), then inspected the details in the image in a chaotic fashion. This means the brain added the details as if it was filling in the gaps in its virtual image perception with new puzzle pieces on top of the original trend inference. Thus, the process ensures the brain updates its perception of the image depending on where the details fit in the image and how long the brain inspects the image. This means the VPU perceives the world adaptively rather than statically. The natural follow-up to this experiment would be to attempt to use this model to sample optical flow information in real-time, which implies using the virtual brain to analyse images in movement instead of static ones. A concreate application of this model may be possible in bionic eyes as it would be able to manage hardware components "cognitively" to get adaptive vision. This would be possible perhaps with one of the bionic eyes presented by the visual implants described in the article "Sight Restoration Comes into Focus: Versions of Visual Prostheses" (Mertz, 2012).

Though the model may seem realistic and robust at first glance, it has flaws resulting from the assumptions stated in the methods' section. Firstly, the full wavelet spectrum of the image is not processed and processes governing the photoreceptors do not get all the photons, which result in the impossibility to know with absolute certainty, which photons are going to hit the photoreceptors as the "photon sampling" is a stochastic process. In addition, results are non-reproducible due to the LA, creating non-predictable fluctuation inputs. This results it in the human observer to be unable to concretely observe any characteristic patterns in the output signal. A practical flaw may also result in real-time data processing being impossible as large amounts of data are treated via 2D MWD running on large number of artificial LIF neurons, which can lead to long computing times. Finally, the LIF ANN processes cannot be tracked qualitatively, which means it is unknown to us which features are detected by the VPU.

My model is designed to see an image and collapse its features together to create a neural response. However, other similarly neuro-inspired biological models exist such as the Reichardt detector, which is used in motion detection specifically and based on the optomotor behaviour of insects. This tracks the way the image moves by capturing the brightness of certain pixels in the image. The brightness intensities measured are then low-pass filtered at the same specific cut-off frequency in two subunits. The respective outputs are finally subtracted, which results in a signal creating motion detection (Borst & Euler, 2011). Though his detector is proven to work and is much less computation greedy, it can only harness the two brightest points in an image. Whereas the virtual eye model created chaotic detail sampling in the virtual eye ensures data is taken all across the image instead of only one spot and then the image is inferred by the VPU. The Reichardt model lacks the feature inclusion from the virtual eye but has the benefit of knowing the selected features.

In addition, an interesting cyber security opportunity may be born out of this process using neural-encoding in conjunction with chaotic systems to design new ways of encrypting and securing data (Morteza , et al., 2014). This means the LA will make it impossible for anyone but the proper LIF ANN to decrypt the information, which means it could potentially be used to make data much safer in systems as each system would encode the data with its own characteristic key. The hacker will then have to find a way to recreate exactly the same LIF ANN to access the encrypted information, which is very unlikely to happen.

Conclusion

In conclusion, I managed to create a model able to give "cognitive vision" to a machine, which can create different perceptions of an image as different signals are perceived each time the image is seen. The model proved LIF ANNs were able to completely produce a stimulus and process it. Then, the VPU responded to the image with its very own response depending on the time it will use to inspect the details. This model represents a new way of giving vision to a machine and uses a cognitive approach instead of an AI approach to the problem. This model replicates the biological phenomenon of image inspection based on biomimicry. However, it has several drawbacks as it does not replicate the biological phenomenon completely and none of the original patterns are perceivable in the signal. Previous models of this type have been tested and proven such as the Reichardt motion detector. This cognitive VPU may have some applications in cybersecurity to encrypt information. To expand this project further, an attempt to combine this virtual eye model with a motion detector such as Reichardt's detector can be made in order to create complete environmental awareness in a machine.

Appendix

General comments on the code:

- The code used for the project is written in the python programming language.
- The code runs on the spyder text editor.
- The nengo (Neural ENgineerinG Object) library was used to create the simulations.
- Instead of coding all the wavelet transforms by hand the PyWavelet package was used to create the wavelet transform tools.
- The code uses the following library packages with their most recent versions: PyWavelets 0.5.2, Nengo 2.6 and Python 2.7.14.
- The code is divided into three files "WaveletWrapper.py", "SignalEncoding.py", "NeuralVisualProcessingUnit.py".
- If you choose to try the experiment yourself beware to respect the python indentation otherwise the code will not work

Code Presentation:

- Standard code will be in Times New Roman size 12 black.
- Key words such as for, if, from, import, etc... will be in Times New Roman size 12 blue.
- Strings and quoted comments will be in Times New Roman size 12 green.
- Comments preceded by a hash '#' (including the hash) will be in grey Times New Roman size 12 grey.
- The key word None and standard Python functions such as len, range, print, etc… will be in Times New Roman size 12 purple.
- Numbers (1, 2, 3, 4, 21, etc...) will be in Times New Roman size 12 dark red.
- The "self" key word referencing attribute or functions of a class itself will be in light brown.

Code in File “WaveletWrapper.py”:

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""
Created on Mon Nov 20 20:04:04 2017

@author: Teddy Edmond Benkohen
"""

import pywt
import numpy as np
import matplotlib.pyplot as plt

class WaveWrap:

        """ The class WaveWrap is designed to contain functions that make wavelet transforms easy
        to compute as even with PyWavelets, which makes finding wavelet transforms easy it is still
        a tedious process. This is why I created a personally customized wrapper, which makes the
        task less tedious."""

        def __init__(self):

                self.complexityLevel = None
                self.waveletName = None
                self.waveletShortName = None
                self.waveletType = None
                self.waveletObj = None

        def infer_waveletObj(self):

                self.waveletObj = pywt.Wavelet(self.waveletType)
                return self.waveletObj

        def infer_waveletType(self):

                if self.complexityLevel is None:

                        self.waveletType = str(self.waveletShortName)
                        self.infer_waveletObj()

                else:

                        self.waveletType = str(self.waveletShortName + str(self.complexityLevel))
                        self.infer_waveletObj()

                return self.waveletType
```

```
    def create_wavelet(self,
                wavelet_name = None,
                wavelet_short_name = None,
                complexity_level = None):

        if wavelet_name == "Haar" or wavelet_short_name == "haar":

            self.waveletName = "Haar"
            self.waveletShortName = "haar"
            self.complexityLevel = None
            self.waveletType = self.infer_waveletType()

        if wavelet_name == "Daubechies" or wavelet_short_name == "db":

            self.waveletName = "Daubechies"
            self.waveletShortName = "db"
            self.complexityLevel = complexity_level
            self.waveletType = self.infer_waveletType()

        if self.waveletType not in pywt.wavelist('db'):

            print("Warning: This wavelet does not exist!")
            print("Use one of the following:", pywt.wavelist('db'))
            return None

        if wavelet_name == "Symlets" or wavelet_short_name == "sym":

            self.waveletName = "Symlets"
            self.waveletShortName = "sym"
            self.complexityLevel = complexity_level
            self.waveletType = self.infer_waveletType()

        if self.waveletType not in pywt.wavelist('sym'):

            print("Warning: This wavelet does not exist!")
            print("Use one of the following:", pywt.wavelist('sym'))
            return None

        if wavelet_name == "Coiflets" or wavelet_short_name == "coif":

            self.waveletName = "Coiflets"
            self.waveletShortName = "coif"
            self.complexityLevel = complexity_level
            self.waveletType = self.infer_waveletType()

        if self.waveletType not in pywt.wavelist('coif'):

            print("Warning: This wavelet does not exist!")
            print("Use one of the following:", pywt.wavelist(‘coif’))
            return None
```

```
if wavelet_name == "Biorthogonal" or wavelet_short_name == "bior":

        self.waveletName = "Biorthogonal"
        self.waveletShortName = "bior"
        self.complexityLevel = complexity_level
        self.waveletType = self.infer_waveletType()

if self.waveletType not in pywt.wavelist('bior'):

        print("Warning: This wavelet does not exist!")
        print("Use one of the following:", pywt.wavelist('bior'))
        return None

if wavelet_name == "Reverse biorthogonal" or wavelet_short_name == "rbio":

        self.waveletName = "Biorthogonal"
        self.waveletShortName = "bior"
        self.complexityLevel = complexity_level
        self.waveletType = self.infer_waveletType()

if self.waveletType not in pywt.wavelist('bior'):

        print("Warning: This wavelet does not exist!")
        print("Use one of the following:", pywt.wavelist('rbio'))
        return None

if wavelet_name == "Discrete Meyer FIR" or wavelet_short_name == "dmey":

        self.waveletName = "Discrete Meyer FIR"
        self.waveletShortName = "dmey"
        self.complexityLevel = None
        self.waveletType = self.infer_waveletType()

if wavelet_name == "Gaussian" or wavelet_short_name == "gaus":

        self.waveletName = "Gaussian"
        self.waveletShortName = "gaus"
        self.complexityLevel = complexity_level
        self.waveletType = self.infer_waveletType()

if self.waveletType not in pywt.wavelist('gaus'):

        print("Warning: This wavelet does not exist!")
        print ("Use one of the following:", pywt.wavelist('gaus'))
        return None
```

```
if wavelet_name == "Mexican Hat" or wavelet_short_name == "mexh":

        self.waveletName = "Mexican Hat"
        self.waveletShortName = "mexh"
        self.complexityLevel = None
        self.waveletType = self.infer_waveletType()

if wavelet_name == "Morlet" or wavelet_short_name == "morl":

        self.waveletName = "Morlet"
        self.waveletShortName = "morl"
        self.complexityLevel = None
        self.waveletType = self.infer_waveletType()

if wavelet_name == "Complex Gaussian" or wavelet_short_name == "cgau":

        self.waveletName = "Complex Gaussian"
        self.waveletShortName = "cgau"
        self.complexityLevel = complexity_level
        self.waveletType = self.infer_waveletType()

if self.waveletType not in pywt.wavelist('cgau'):

        print ("Warning: This wavelet does not exist!")
        print ("Use one of the following:", pywt.wavelist('cgau'))
        return None

if wavelet_name == "Shannon" or wavelet_short_name == "shan":

        self.waveletName = "Shannon"
        self.waveletShortName = "shan"
        self.complexityLevel = None
        self.waveletType = self.infer_waveletType()

if wavelet_name == "Frequency B-Spline" or wavelet_short_name == "fbsp":

        self.waveletName = "Frequency B-Spline"
        self.waveletShortName = "fbsp"
        self.complexityLevel = None
        self.waveletType = self.infer_waveletType()

if wavelet_name == "Complex Morlet" or wavelet_short_name == "cmor":

        self.waveletName = "Complex Morlet"
        self.waveletShortName = "cmor"
        self.complexityLevel = None
        self.waveletType = self.infer_waveletType()

return self.complexityLevel, self.waveletName, self.waveletShortName
```

```
def discrete_wavelet_transform(self, data):

    if self.waveletName is None or self.waveletShortName is None:

        self.create_wavelet(wavelet_short_name = 'db', complexity_level = 38)

    if self.waveletType not in pywt.wavelist(family = None, kind = 'discrete'):

        print("Warning: To use the discrete wavelet transform, your wavelet must be
            discrete. Choose a Discrete Wavelet! :")
        print(pywt.wavelist(family = None, kind = 'discrete'))
        return None

    trend, fluctuations = pywt.dwt(data, self.waveletType)
    return trend, fluctuations

def twod_discrete_wavelet_transform(self, data, concat = True):

    if self.waveletName is None or self.waveletShortName is None:

        self.create_wavelet(wavelet_short_name = 'db', complexity_level = 38)

    if self.waveletType not in pywt.wavelist(family = None, kind = 'discrete'):

        print("Warning: To use the discrete wavelet transform, your wavelet must be
            discrete. Choose a Discrete Wavelet! :")
        print(pywt.wavelist(family = None, kind = 'discrete'))
        return None

    coefficients = pywt.dwt2(data, self.waveletType)
    trend, (horizontal_detail, vertical_detail, fluctuations) = coefficients

        if concat == True:

            return coefficients

        if concat == False:

            return trend, horizontal_detail, vertical_detail, fluctuations
def inverse_discrete_wavelet_transform(self, trend, fluctuations):

    if self.waveletType not in pywt.wavelist(family = None, kind = 'discrete'):

        print("Warning: To use the discrete wavelet transform, your wavelet must be
            discrete. Choose a Discrete Wavelet! :")
        print(pywt.wavelist(family = None, kind = 'discrete'))
        return None

    reconstructed_data = pywt.idwt(trend, fluctuations, self.waveletType,
                                    mode = 'symmetric', axis = -1)
    return reconstructed_data

def continuous_wavelet_transform(self, data):

    if self.waveletName is None or self.waveletShortName is None:

        self.create_wavelet(wavelet_short_name = 'gaus', complexity_level = 8)

    if self.waveletType not in pywt.wavelist(family = None, kind = 'continuous'):

        print("Warning: To use the continuous wavelet transform, your wavelet must
            be discrete. Choose a Discrete Wavelet! :")
        print(pywt.wavelist(family = None, kind = 'discrete'))
        return None

    coefficients, frequencies = pywt.cwt(data, np.arange(1, len(data)), self.waveletType)
    return coefficients, frequencies
```

```
def multilevel_wavelet_decomposition(self, data, level=None):

    if self.waveletName is None or self.waveletShortName is None:

        self.create_wavelet(wavelet_short_name = 'db', complexity_level = 38)

    if self.waveletType not in pywt.wavelist(family = None, kind = 'discrete'):

        print("Warning: To use the discrete wavelet transform, your wavelet must be
            discrete. Choose a Discrete Wavelet! :")
        print(pywt.wavelist(family = None, kind = 'discrete'))
        return None

    coefficients = pywt.wavedec(data, self.waveletType, level = level)

    if level is None:

        print('Warning: The default level decomposition is the maximum level:',
            pywt.dwt_max_level(data_len = len(data),
            filter_len =self.waveletObj))

    return coefficients
```

```
def twod_multilevel_wavelet_decomposition(self,
                                          data,
                                          level=None,
                                          concat = True,
                                          hl_only = False):

        if self.waveletName is None or self.waveletShortName is None:

                self.create_wavelet(wavelet_short_name = 'db', complexity_level = 38)

        if self.waveletType not in pywt.wavelist(family = None, kind = 'discrete'):

                print("Warning: To use the discrete wavelet transform, your wavelet must be
                        discrete. Choose a Discrete Wavelet! :")
                print(pywt.wavelist(family = None, kind = 'discrete'))

                return None

        coefficients_list = pywt.wavedec2(data, self.waveletType, level = level)

        if level is None:

                print('Warning: The default level decomposition is the maximum level:',
                        pywt.dwt_max_level(data_len = len(data),
                                          filter_len = self.waveletObj))

        return coefficients_list
```

```
    def multilevel_wavelet_reconstruction(self, coefficients):

        if self.waveletType not in pywt.wavelist(family = None, kind = 'discrete'):

            print("Warning: To use the discrete wavelet transform, your wavelet must be
                discrete. Choose a Discrete Wavelet! :")
            print(pywt.wavelist(family = None, kind = 'discrete'))
            return None

        reconstructed_data = pywt.waverec(coefficients,
                                          self.waveletType,
                                          mode = 'symmetric',
                                          axis = -1)
        return reconstructed_data

    def twod_multilevel_wavelet_reconstruction(self, coefficients_list):

        if self.waveletType not in pywt.wavelist(family = None, kind = 'discrete'):

            print("Warning: To use the discrete wavelet transform, your wavelet must be
                discrete. Choose a Discrete Wavelet! :")
            print(pywt.wavelist(family = None, kind = 'discrete'))
            return None

        reconstructed_data = pywt.waverec2(coefficients_list,
                                           self.waveletType,
                                           mode = 'symmetric',
                                           axes = (-2, -1))
        return reconstructed_data
```

```
if __name__ == "__main__”:

        """ Analysing the effect of 2D wavelet transforms on the aero image provided by the
        pywavelets data."""

        # Visualize the image.
        data = pywt.data.aero()
        plt.figure()
        plt.imshow(data, cmap = 'gray')
        plt.show()

        # Do 2D wavelet decomposition.
        # Compute 2D wavelet transforms with the Daubechies 38 Wavelet as it is the default value I
        # chose and maximal decomposition level.
        # The same wavelet will be used over again throughout the code and in the other files.
        wave = WaveWrap()
        coefs = wave.twod_multilevel_wavelet_decomposition(data)

        # Trends in the image.
        plt.figure()
        plt.imshow(coefs[0], cmap = 'gray')
        plt.show()

        # Horizontal details in the image.
        plt.figure()
        plt.imshow(coefs[1][0], cmap = 'gray')
        plt.show()

        # Vertical details in the image.
        plt.figure()
        plt.imshow(coefs[1][1], cmap = 'gray')
        plt.show()

        # Diagonal details in the image.
        plt.figure()
        plt.imshow(coefs[1][2], cmap = 'gray')
        plt.show()

        # Reconstruct the data.
        rdata = wave.twod_multilevel_wavelet_reconstruction(coefs)
        plt.figure()
        plt.imshow(rdata, cmap = 'gray')
        plt.show()
```

```
""" Analysing the effect of 2D wavelet transforms on the camera image provided by the
pywavelets data. """

# Visualize the image.
data = pywt.data.camera()
plt.figure()
plt.imshow(data, cmap = 'gray')
plt.show()

# Do wavelet decomposition.
wave = WaveWrap()
coefs = wave.twod_multilevel_wavelet_decomposition(data)

# Trends in the image.
plt.figure()
plt.imshow(coefs[0], cmap = 'gray')
plt.show()

# Horizontal details in the image.
plt.figure()
plt.imshow(coefs[1][0], cmap = 'gray')
plt.show()

# Vertical details in the image.
plt.figure()
plt.imshow(coefs[1][1], cmap = 'gray')
plt.show()

# Diagonal details in the image.
plt.figure()
plt.imshow(coefs[1][2], cmap = 'gray')
plt.show()

# Reconstruct data.
rdata = wave.twod_multilevel_wavelet_reconstruction(coefs)
plt.figure()
plt.imshow(rdata, cmap = 'gray')
plt.show()
```

Code in File "SignalEncoding.py":

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""
Created on Sat Nov 25 15:52:59 2017

@author: Teddy Edmond Benkohen
"""

from WaveletWrapper import WaveWrap, pywt, np, plt

class TimeSeries:

    """ The class TimeSeries is designed to manage the wavelet transforms output"""

    def __init__(self):

        self.timeSeriesSet = None
        self.maxTimeSerie = []
        self.minTimeSerie = []
        self.MaxTss = None
        self.MinTss = None
        self.scaledMaxTimeSerie = []
        self.scaledMinTimeSerie = []

    def create_timeSeriesSet(self, time_series_data):

        self.timeSeriesSet = []

        for i in range(0, len(time_series_data)):

            self.timeSeriesSet.append(time_series_data[i])

        return self.timeSeriesSet

    def find_maxTimeSerie(self, time_series_data):

        if self.timeSeriesSet is None:

            self.create_timeSeriesSet(time_series_data)

            for i in range(0, len(self.timeSeriesSet)):

                self.maxTimeSerie.append(max(self.timeSeriesSet[i]))

        return self.maxTimeSerie
    def find_minTimeSerie(self, time_series_data):

        if self.timeSeriesSet is None:

            self.create_timeSeriesSet(time_series_data)

        for i in range(0, len(self.timeSeriesSet)):

            self.minTimeSerie.append(min(self.timeSeriesSet[i]))

        return self.minTimeSerie

    def find_scalers(self, time_series_data):

        if len(self.minTimeSerie) == 0:

            self.find_minTimeSerie(time_series_data)

        if len(self.maxTimeSerie) == 0:

            self.find_maxTimeSerie(time_series_data)
            self.MinTss = min(self.minTimeSerie)

        self.MaxTss = max(self.maxTimeSerie) + abs(self.MinTss)
        return self.MaxTss, self.MinTss


    def find_scaledMinTimeSerie(self, time_series_data):

        if self.MaxTss == None or self.MinTss == None:

            self.find_scalers(time_series_data)

        for i in range(0, len(self.minTimeSerie)):

            self.scaledMinTimeSerie.append(
                                        (self.minTimeSerie[i] + abs(self.MinTss))
                                        /self.MaxTss)

        return self.scaledMinTimeSerie
    def find_scaledMaxTimeSerie(self, time_series_data):

        if self.MaxTss == None or self.MinTss == None:

            self.find_scalers(time_series_data)

        for i in range(0, len(self.maxTimeSerie)):

            self.scaledMaxTimeSerie.append(
                                        (self.maxTimeSerie[i] + abs(self.MinTss))
                                        /self.MaxTss)

        return self.scaledMaxTimeSerie
```

```
if __name__ == "__main__":

        """Encoding the aero image into time series containing the strongest intensities of the
        image."""

        # Load and inspect data.
        data = pywt.data.aero()
        print(np.shape(data))
        wave = WaveWrap()
        coefs = wave.twod_multilevel_wavelet_decomposition(data)

        # Create time series.
        ts1 = TimeSeries()
        ts2 = TimeSeries()
        ts3 = TimeSeries()
        ts4 = TimeSeries()

        # Create stimuli in the form of time series scaled between 0 and 1.
        stim1max = ts1.find_scaledMaxTimeSerie(coefs[0]) # Max trends.
        stim2max = ts2.find_scaledMaxTimeSerie(coefs[1][0]) # Max horizontal details.
        stim3max = ts3.find_scaledMaxTimeSerie(coefs[1][1]) # Max vertical details.
        stim4max = ts4.find_scaledMaxTimeSerie(coefs[1][2]) # Max diagonal details.
        stim1min = ts1.find_scaledMinTimeSerie(coefs[0]) # Min trends.
        stim2min = ts2.find_scaledMinTimeSerie(coefs[1][0]) # Min horizontal details.
        stim3min = ts3.find_scaledMinTimeSerie(coefs[1][1]) # Min vertical details.
        stim4min = ts4.find_scaledMinTimeSerie(coefs[1][2]) # Min diagonal details.

        #  Plotting the stimulus 1 enveloppe.
        plt.figure()
        plt.plot(stim1max)
        plt.plot(stim1min)
        plt.show()

        # Plotting the stimulus 2 enveloppe.
        plt.figure()
        plt.plot(stim2max)
        plt.plot(stim2min)
        plt.show()

        # Plotting the stimulus 3 enveloppe.
        plt.figure()
        plt.plot(stim3max)
        plt.plot(stim3min)
        plt.show()

        # Plotting the stimulus 4 enveloppe.
        plt.figure()
        plt.plot(stim4max)
        plt.plot(stim4min)
        plt.show()
```

```
"""Encoding the camera image into time series containing the strongest intensities of the image."""

# Load and inspect data.
data = pywt.data.camera()
print(np.shape(data))
wave = WaveWrap()
coefs = wave.twod_multilevel_wavelet_decomposition(data)

# Create time series.
ts1 = TimeSeries()
ts2 = TimeSeries()
ts3 = TimeSeries()
ts4 = TimeSeries()

# Create stimuli in the form of time series scaled between 0 and 1.
stim1max = ts1.find_scaledMaxTimeSerie(coefs[0]) # Max trends.
stim2max = ts2.find_scaledMaxTimeSerie(coefs[1][0]) # Max horizontal details.
stim3max = ts3.find_scaledMaxTimeSerie(coefs[1][1]) # Max vertical details.
stim4max = ts4.find_scaledMaxTimeSerie(coefs[1][2]) # Max diagonal details.
stim1min = ts1.find_scaledMinTimeSerie(coefs[0]) # Min trends.
stim2min = ts2.find_scaledMinTimeSerie(coefs[1][0]) # Min horizontal details.
stim3min = ts3.find_scaledMinTimeSerie(coefs[1][1]) # Min vertical details.
stim4min = ts4.find_scaledMinTimeSerie(coefs[1][2]) # Min diagonal details.

#  Plotting the stimulus 1 enveloppe.
plt.figure()
plt.plot(stim1max)
plt.plot(stim1min)
plt.show()


# Plotting the stimulus 2 enveloppe.
plt.figure()
plt.plot(stim2max)
plt.plot(stim2min)
plt.show()

# Plotting the stimulus 3 enveloppe.
plt.figure()
plt.plot(stim3max)
plt.plot(stim3min)
plt.show()

# Plotting the stimulus 4 enveloppe.
plt.figure()
plt.plot(stim4max)
plt.plot(stim4min)
plt.show()
```

Code in File “NeuralVisualProcessingUnit.py”:

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""
Created on Sun Nov 26 14:01:36 2017

@author: Teddy Edmond Benkohen
"""

import nengo
from nengo.processes import Piecewise
from WaveletWrapper import WaveWrap, pywt, plt
from SignalEncoding import TimeSeries
from nengo.processes import WhiteNoise
from nengo.dists import Uniform
from mpl_toolkits.mplot3d import Axes3D

class Stimulus:

        def __init__(self, minTimeSerie, maxTimeSerie):

                self.minTimeSerie = minTimeSerie
                self.maxTimeSerie = maxTimeSerie
                self.stimulusLength = len(self.minTimeSerie)
                self.piecewiseMinTimeSerie = None # Dictionary containing the minTimeSerie.
                self.piecewiseMaxTimeSerie = None # Dictionary containing the maxTimeSerie.
                self.afferentNeuralNet = None # Model of an artificial Afferent Neural Network.
                self.minStimulus = None
                self.maxStimulus = None
                self.minStimulus = None
                self.maxStimulus = None
                self.minStimulator = None
                self.maxStimulator = None
                self.netStimulator = None
                self.minStimulusProbe = None
                self.maxStimulusProbe = None
                self.minStimulusProbe = None
                self.maxStimulusProbe = None
                self.minStimulatoProbe = None
                self.maxStimulatorProbe = None
                self.netStimulatorProbe = None
                self.stimSim = None
```

```
def list2dict(self, time_series_data):

        keys = []
        values = time_series_data

        for i in range(0, len(time_series_data)):

                key = int(i)
                keys.append(key)

        dictionary = dict(zip(keys, values))
        return dictionary

def create_stimulus(self):

        self.piecewiseMinTimeSerie = self.list2dict(self.minTimeSerie)
        self.piecewiseMaxTimeSerie = self.list2dict(self.maxTimeSerie)
        return  self.piecewiseMinTimeSerie, self.piecewiseMaxTimeSerie

def stimulate(self, piecewiseTimeSerie):

        stimulus = Piecewise(piecewiseTimeSerie)
        return stimulus

def create_afferentNeuralNet(self, number_of_neurons = 100):

        if self.piecewiseMinTimeSerie is None or self.piecewiseMaxTimeSerie is None:

                self.create_stimulus()

        self.afferentNeuralNet = nengo.Network(label = 'Afferent Artificial Neural Network')

        with self.afferentNeuralNet:

                self.minStimulator = nengo.Ensemble(number_of_neurons, dimensions = 1)
                self.maxStimulator = nengo.Ensemble(number_of_neurons, dimensions = 1)
                self.netStimulator = nengo.Ensemble(number_of_neurons * 2,
                                                    dimensions = 1)
                self.minStimulus = nengo.Node(self.stimulate(self.piecewiseMinTimeSerie))
                self.maxStimulus = nengo.Node(self.stimulate(self.piecewiseMaxTimeSerie))
                nengo.Connection(self.minStimulus, self.minStimulator)
                nengo.Connection(self.maxStimulus, self.maxStimulator)
                nengo.Connection(self.minStimulator, self.netStimulator)
                nengo.Connection(self.maxStimulator, self.netStimulator)

        return self.afferentNeuralNet, self.netStimulator
```

```
def probe_afferentNeuralNet(self, sampling_time = 0.01):

    if self.afferentNeuralNet is None:

        self.create_afferentNeuralNet()

    with self.afferentNeuralNet:

        self.minStimulusProbe = nengo.Probe(self.minStimulus,
                                            synapse = sampling_time)
        self.maxStimulusProbe = nengo.Probe(self.maxStimulus,
                                            synapse = sampling_time)
        self.minStimulatorProbe = nengo.Probe(self.minStimulator,
                                            synapse = sampling_time)
        self.maxStimulatorProbe = nengo.Probe(self.maxStimulator,
                                            synapse = sampling_time)
        self.netStimulatorProbe = nengo.Probe(self.netStimulator,
                                            synapse = sampling_time)

    return self.minStimulusProbe,
        self.maxStimulusProbe,
        self.minStimulatorProbe,
        self.maxStimulatorProbe,
        self.netStimulatorProbe

def simulate_afferentNeuralNet(self):

    if self.minStimulusProbe is None
        or self.maxStimulusProbe is None
        or self.minStimulatorProbe is None
        or self.maxStimulatorProbe is None
        or self.netStimulatorProbe is None:

        self.probe_afferentNeuralNet()

    with nengo.Simulator(self.afferentNeuralNet) as AANNsim:

        AANNsim.run(self.stimulusLength)

    # Plotting the physical time serie stimulation of the visual sensory cells.
    plt.figure()
    plt.plot(AANNsim.trange(), AANNsim.data[self.minStimulusProbe], 'b',
        label = "Min input")
    plt.plot(AANNsim.trange(), AANNsim.data[self.maxStimulusProbe], 'r',
        label = "Max input")
    plt.legend()
    plt.show()

    # Plotting the physical time serie output of the visual sensory cells.
```

```
        plt.figure()
        plt.plot(AANNsim.trange(), AANNsim.data[self.minStimulatorProbe], 'b',
                 label = "Max Stimulus output")
        plt.plot(AANNsim.trange(), AANNsim.data[self.maxStimulatorProbe], 'r',
                 label = "Max Stimulus output")
        plt.legend()
        plt.show()

        # Plotting the net physical output of the afferent Leaky-Integrate-Fire Neurons.
        plt.plot(AANNsim.trange(), AANNsim.data[self.netStimulatorProbe][:, 0], 'b',
                 label = "AANN output")
        plt.legend()
        plt.show()

class Vpu(Stimulus):
# V.P.U. stands for Visual Processing Unit.

    def __init__(self, stimulusTrend, stimulusHdetail, stimulusVdetail, stimulusDdetail):

        self.stimLengthList = [stimulusTrend.stimulusLength,
                               stimulusHdetail.stimulusLength,
                               stimulusVdetail.stimulusLength,
                               stimulusDdetail.stimulusLength]
        self.maxStimulusLength = max(self.stimLengthList)
        self.AANNTrend, self.AANNTout = stimulusTrend.create_afferentNeuralNet()
        self.AANNHdetail, self.AANNHout = stimulusHdetail.create_afferentNeuralNet()
        self.AANNVdetail, self.AANNVout = stimulusVdetail.create_afferentNeuralNet()
        self.AANNDdetail, self.AANNDout = stimulusDdetail.create_afferentNeuralNet()
        self.VpuModel = None
        self.LorentzAttractor = None
        self.MT = None
        self.InterMTVAB = None
        self.VAB = None
        self.LorentzAttractorProbe = None
        self.MTProbe = None
        self.InterMTVABProbe = None
        self.VABProbe = None
```

```
def create_Vpu(self,
               tau = 0.1,
               sigma = 10,
               beta = 8.0 / 3,
               rho = 28,
               number_of_neurons = 100,
               weight = 0.1,
               syn = 0.01):

    self.VpuModel = nengo.Network(label = 'Visual Processing Unit')

    # Introducing Lorentz attractor chaotic equations.

    def feedback(x):

        # Default values are set at tau = 0.1, sigma = 10, beta = 8.0/3 and rho = 28.
        Dx0 = -sigma * x[0] + sigma * x[1]
        dx1 = -x[0] * x[2] – x[1]
        dx2 = x[0] * x[1] – beta * (x[2] + rho) – rho
        return [dx0 * tau + x[0], dx1 * tau + x[1], dx2 * tau + x[2]]

    with self.VpuModel:

        # Creating the Lorentz Attractor allows us to model rapid eye movement as
        #a chaotic process for detail inspection in images.
        self.LorentzAttractor = nengo.Ensemble(20 * number_of_neurons, 3,
                                               radius = 60)

        # Creating the second part of the Vpu.
        # The eye's motion area (MT).
        self.MT = nengo.Ensemble(number_of_neurons, dimensions = 1,
                                 noise = WhiteNoise(dist = Uniform(-0.3, 0.3)))
        # The area carrying the signal from the MT to the Visual Area of the Brain
        # (InterMTVAB).
        self.InterMTVAB = nengo.Ensemble(2 * number_of_neurons,
                                 dimensions = 1,
                                 noise = WhiteNoise(dist = Uniform(-0.3, 0.3)))
        # The Visual Area of the Brain (VAB).
        self.VAB = nengo.Ensemble(number_of_neurons, dimensions = 1,
                                 noise = WhiteNoise(dist = Uniform(-0.3, 0.3)))

        # Linking image details to the Lorentz Attractor.

        # Horizontal details.
        with self.AANNHdetail:

            nengo.Connection(self.AANNHout, self.LorentzAttractor[0])

        # Vertical details.
        with self.AANNVdetail:

                nengo.Connection(self.AANNVout, self.LorentzAttractor[1])

        # Diagonal details.
        with self.AANNDdetail:

                nengo.Connection(self.AANNDout, self.LorentzAttractor[2])

        # Introducting chaotic feedback in the Lorentz Attractor.
        nengo.Connection(self.LorentzAttractor,
                         self.LorentzAttractor,
                         function = feedback,
                         synapse = tau)

        # Connecting the Lorentz Attractor to MT.
        nengo.Connection(self.LorentzAttractor[0], self.MT, synapse = syn)
        nengo.Connection(self.LorentzAttractor[1], self.MT, synapse = syn)
        nengo.Connection(self.LorentzAttractor[2], self.MT, synapse = syn)

        # Connecting the trend to MT.

        with self.AANNTrend:

                nengo.Connection(self.AANNTout, self.MT)

        # Connecting MT ensemble to InterMTVAB ensemble.
        nengo.Connection(self.MT, self.InterMTVAB, transform = weight,
                         synapse= 10 * syn)
        # Connecting InterMTVAB ensemble to itself.
        nengo.Connection(self.InterMTVAB, self.InterMTVAB, synapse = 10 * syn)
        # Connecting InterMTVAB ensemble to output.
        nengo.Connection(self.InterMTVAB, self.VAB, synapse = syn)

    return self.VpuModel
def probe_Vpu(self, sampling_time = 0.01):

    if self.VpuModel is None:

        self.create_Vpu()

    tau = 10 * sampling_time

    with self.VpuModel:

        self.LorentzAttractorProbe = nengo.Probe(self.LorentzAttractor,
                                                 synapse=tau)
        self.MTProbe = nengo.Probe(self.MT, synapse = sampling_time)
        self.InterMTVABProbe = nengo.Probe(self.InterMTVAB,
                                           synapse = sampling_time)
        self.VABProbe = nengo.Probe(self.InterMTVAB, synapse = sampling_time)

    return self.LorentzAttractorProbe,
        self.MTProbe,
        self.InterMTVABProbe,
        self.VABProbe
```

```
def simulate_Vpu(self):

        if self.LorentzAttractorProbe is None
                or self.MTProbe is None
                or self.InterMTVABProbe is None
                or self.VABProbe is None:

                self.probe_Vpu()

        with nengo.Simulator(self.VpuModel) as Vpu_sim:

                Vpu_sim.run(self.maxStimulusLength)

        # 3D plot of Chaotic Data.
        ax = plt.figure().add_subplot(111, projection = '3d')
        ax.plot(*Vpu_sim.data[self.LorentzAttractorProbe].T)

        # 2D Lorentz attractor.
        plot.plt.figure()
        plt.plot(Vpu_sim.trange(), Vpu_sim.data[self.LorentzAttractorProbe])
        plt.show()

        # Plotting the time serie virtual motion output resulting from chaotic movement.
        plt.figure()
        plt.plot(Vpu_sim.trange(), Vpu_sim.data[self.MTProbe], 'b', label = "MT Response")
        plt.legend()
        plt.show()

        # Plotting the output of virtual sensory cells.
        plt.figure()
        plt.plot(Vpu_sim.trange(), Vpu_sim.data[self.InterMTVABProbe], 'b',
                label = "Inter MT-VAB response")
        plt.legend()
        plt.show()

        # Plotting the output of the afferent Leaky-Integrate-Fire Neurons.
        plt.figure()
        plt.plot(Vpu_sim.trange(), Vpu_sim.data[self.VABProbe], 'b',
                label = "Vpu Response")
        plt.legend()
        plt.show()
```

```
if __name__ == "__main__”:

        """Simulating the neural response to the aero image."""

        # Loading and inspecting the data then finding the time Series (same code as in file
        SignalEncoding.py).
        data = pywt.data.aero()
        wave = WaveWrap()
        coefs = wave.twod_multilevel_wavelet_decomposition(data)
        ts1 = TimeSeries()
        ts2 = TimeSeries()
        ts3 = TimeSeries()
        ts4 = TimeSeries()
        ts1max = ts1.find_scaledMaxTimeSerie(coefs[0])
        ts2max = ts2.find_scaledMaxTimeSerie(coefs[1][0])
        ts3max = ts3.find_scaledMaxTimeSerie(coefs[1][1])
        ts4max = ts4.find_scaledMaxTimeSerie(coefs[1][2])
        ts1min = ts1.find_scaledMinTimeSerie(coefs[0])
        ts2min = ts2.find_scaledMinTimeSerie(coefs[1][0])
        ts3min = ts3.find_scaledMinTimeSerie(coefs[1][1])
        ts4min = ts4.find_scaledMinTimeSerie(coefs[1][2])

        # Create the stimuli.
        stim1 = Stimulus(ts1min, ts1max)
        stim2 = Stimulus(ts2min, ts2max)
        stim3 = Stimulus(ts3min, ts3max)
        stim4 = Stimulus(ts4min, ts4max)

        # Simulate the incoming stimuli from the image.
        stim1.simulate_afferentNeuralNet()
        stim2.simulate_afferentNeuralNet()
        stim3.simulate_afferentNeuralNet()
        stim4.simulate_afferentNeuralNet()

        # Simulate the Vpu's response to the stimuli.
        vpu = Vpu(stim1, stim2, stim3, stim4)
        vpu.simulate_Vpu()

        """Simulating the neural response to the camera image."""

        # Loading and inspecting the data then finding the time Series (same code as in file
        SignalEncoding.py).
        data = pywt.data.camera()
        wave = WaveWrap()
        coefs = wave.twod_multilevel_wavelet_decomposition(data)
        ts1 = TimeSeries()
        ts2 = TimeSeries()
        ts3 = TimeSeries()
        ts4 = TimeSeries()
```

```
ts1max = ts1.find_scaledMaxTimeSerie(coefs[0])
ts2max = ts2.find_scaledMaxTimeSerie(coefs[1][0])
ts3max = ts3.find_scaledMaxTimeSerie(coefs[1][1])
ts4max = ts4.find_scaledMaxTimeSerie(coefs[1][2])
ts1min = ts1.find_scaledMinTimeSerie(coefs[0])
ts2min = ts2.find_scaledMinTimeSerie(coefs[1][0])
ts3min = ts3.find_scaledMinTimeSerie(coefs[1][1])
ts4min = ts4.find_scaledMinTimeSerie(coefs[1][2])

# Create the stimuli.
stim1 = Stimulus(ts1min, ts1max)
stim2 = Stimulus(ts2min, ts2max)
stim3 = Stimulus(ts3min, ts3max)
stim4 = Stimulus(ts4min, ts4max)

# Simulate the incoming stimuli from the image.
stim1.simulate_afferentNeuralNet()
stim2.simulate_afferentNeuralNet()
stim3.simulate_afferentNeuralNet()
stim4.simulate_afferentNeuralNet()

# Simulate the Vpu's response to the stimuli.
vpu = Vpu(stim1, stim2, stim3, stim4)
vpu.simulate_Vpu()
```

Bibliography

1. Aziz, S. M. & Pham, D. M., 2012. Efficient parallel architecture for multi-level forward discrete wavelet transform processors. *Computers and Electrical Engineering,* 15 June, 38(5), p. 1325–1335.

2. Borst, A. & Euler, T., 2011. Seeing Things in Motion: Models, Circuits, and Mechanisms. *Neuron,* 22 September, 71(6), pp. 974-994.

3. Chatfield, C., 2004. *The Analysis of Time Series: An Introduction.* Sixth Edition éd. Boca Raton(Florida): CRC Press Company.

4. Eliasmith, C., 2017. *Nengo.* [Online]
Available at: https://www.nengo.ai/nengo/index.html#
[Accessed on the 3 December 2017].

5. Forsyth, D. A., Ponce, J., Mukherjee, S. & Bhattacharjee, A. K., 2012. *Computer Vision: A Modern Approach.* 2nd Edition éd. Harlow(Essex): Pearson Education Limited.

6. Gu, S. et al., 2015. Controllability of structural brain networks. *Nature Communications ,* 1 October, 6(8414), pp. 1-10.

7. Hunsberger, E. & Eliasmith, C., 2015. *Spiking Deep Networks with LIF Neurons.* [En ligne]
Available at: https://arxiv.org/pdf/1510.08829.pdf
[Accessed on the 3 December 2017].

8. Li, S.-Y. & Ge, Z.-M., 2009. A novel study of parity and attractor in the time reversed Lorentz system. *Physics Letters A,* 3 September, 373(44), pp. 4053-4059.

9. Markram, H. et al., 2011. Introducing the Human Brain Project. *Procedia Computer Science,* 22 December, 7(11), pp. 39-42.

10. Mertz, L., 2012. Sight Restoration Comes into Focus: Versions of Visual Prostheses. *IEEE Pulse,* 24 September, 3(5), pp. 10-16.

11. Morteza , S. et al., 2014. Using 3-cell chaotic map for image encryption based on biological operations. *Nature: Non-Linear Dynamics,* February, 75(3), pp. 407-416.

12. Pan, Y., 2016. Heading toward Artificial Intelligence 2.0. *Engineering,* 2 December, 2(4), pp. 409-413.

13. Patterson, J. & Gibson, A., 2017. *Deep Learning: A practitioners approach.* Sebastopol(California): O'Reilly Media Inc.

14. PyWavelets Developers, T., 2017. *PyWavelets Documentation.* [Online]
Available at: https://media.readthedocs.org/pdf/pywavelets/latest/pywavelets.pdf
[Accessed on the 2 December 2017].

15. Semmlow, J. L., 2009. *Biosignal and Medical Image Processing.* 2nd Edition éd. Boca Raton(Florida): CRC Press.

16. Shi, Y. & Toga, A. W., 2017. Connectome imaging for mapping human brain pathways. *Nature Molecular Psychiatry,* September, 22(9), pp. 1230-1240.