A COMPUTATIONAL FRAMEWORK FOR REALISTIC RETINA MODELING (COREM)

Pablo Martínez-Cañada
pablomc@ugr.es
University of Granada (Spain)

December 17, 2015

This document provides an overview of COREM simulator. It describes the installation process and how to configure a retina model through the simulation script. The goal of this document is not to explain the different retina models that have been evaluated, details of them can be found in related papers[1, 2, 3, 4].

Contents

1 Introduction

Computational simulations of the retina have led to valuable insights about the biophysics of its neuronal activity and processing principles. A great number of retina models have been proposed to reproduce the behavioral diversity of the different visual processing pathways. While many of these models share common

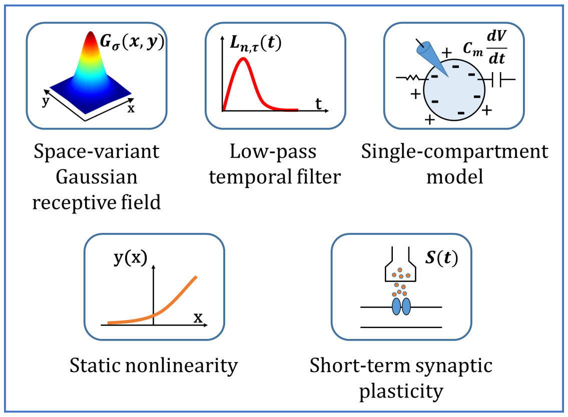

Figure 1: Computational retinal microcircuits that are used as basic building blocks in COREM. They consist of one spatial processing module, a space-variant Gaussian filter, two temporal modules, a low-pass temporal filter and a single-compartment model, a configurable time-independent nonlinearity and a STP function.

computational stages, previous efforts have been more focused on fitting specific retina functions rather than generalizing them beyond a particular model. In COREM, we have defined a set of computational retinal microcircuits that can be used as basic building blocks for the modeling of different retina mechanisms (see Fig. 1).
With this set of microcircuits our aim was to summarize the basic principles of the retina modeling and embrace some of the most significant algorithms proposed in the literature, without going into depth on the description of the neural morphology. They can be combined in different ways to reproduce single-cell and large-scale retina models at different abstraction levels.

2 Installation

COREM compiles under Linux with gcc (last tested version: 4.6.3). It makes use of some external and standard libraries in C++:

- CImg, an image processing toolbox (included in the software package)
- The X11 client-side library (used by CImg)
- OpenMP (to speed up processing of the space-variant Gaussian filter)

2.1 Stand-alone program

To compile COREM as a stand-alone program, in directory **build**, just type:

```
make
```

The simulator can be executed now using the default retina script:

```
./COREM
```

or a different retina script located in directory **Retina_scripts**:

```
./COREM example_1.py
```

2.2 NEST extension module

2.2.1 Prerequisites

Detected as an issue, loading an extension module fails when NEST 2.8.0 [5] is built with gcc 4.7.2 under Linux. Defining a linkage for time constants in c file fixes the problem. In **nestkernel/nest_time.cpp**, add the following lines (marked by +):

```
nest::double_t Time::Range::MS_PER_STEP = TICS_PER_STEP / TICS_PER_MS;
nest::double_t Time::Range::STEPS_PER_MS = 1 / Time::Range::MS_PER_STEP;

// define for unit -- const'ness is in the header
// should only be necessary when not folded away
// by the compiler as compile time consts
(+)const tic_t Time::LimitPosInf::tics;
(+)const delay Time::LimitPosInf::steps;
(+)const tic_t Time::LimitNegInf::tics;
(+)const delay Time::LimitNegInf::steps;

tic_t
Time::compute_max()
```

Define also the environment variable **NEST_INSTALL_DIR** to contain the path to which you have installed NEST, e.g. using bash:

```
export NEST_INSTALL_DIR=/home/pablo/nest/
```

2.2.2 Build and install

To build COREM as an extension module of NEST, first change to **NEST_Module /COREM_Module** and prepare by typing:

```
./bootstrap.sh
```

Leave **COREM_Module** and go to **build** directory:

```
cd ../build/
```

Configure:

```
../COREM_Module/configure --with-nest=$NEST_INSTALL_DIR/bin/nest-config
```

The module will then be installed to **${NEST_INSTALL_DIR}**.Compile:

```
make
make install
```

Set your **LD_LIBRARY_PATH** to include **$NEST_INSTALL_DIR/lib/nest**. Start NEST in PyNest and load the module by typing:

```
nest.Install("COREM")
```

3 First steps: overview of the retina script

This section presents a general example of the retina script. The aim of this example is to show an overview of the main features that can be configured in the retina script. The retina model implemented in this script processes an input sequence of images based on a general retina architecture that includes all neuron types. Temporal evolution and spatial arrangement of some neurons' membrane potentials are recorded by multimeters and displayed during and after simulation.
We will use the retina script **example_1.py** in **Retina_scripts**. The first lines are mandatory in every retina script and allow configuration of the simulation, like the simulation step, simulation time and number of repetitions of the experiment (number of trials). For random input stimuli (white noise or jittering gratings) the number of trials determines the number of samples used to average results. With image sequences this parameter is 1.

Note when copying code from this document: a command of the retina script (that begins with **retina.**) cannot be broken into two lines.

```
### Simulation parameters ###

retina.TempStep('1') # simulation step (in ms)
retina.SimTime('1200') # simulation time (in ms)
retina.NumTrials('1') # number of trials
retina.PixelsPerDegree({'5'}) # pixels per degree of visual angle
retina.NRepetitions('100') # number of simulation steps every image in the input
    sequence is repeated
retina.DisplayDelay('0') # display delay
retina.DisplayZoom({'10.0'}) # display zoom
retina.DisplayWindows('3') # Displays per row
```

The parameter **PixelsPerDegree** sets the distance between the visual input stimulus and the simulated retina. It affects spatial signal integration. When an input sequence of images is used, **NRepetitions** is the number of times (in simulation steps) that every image in the sequence is projected to the retina. In our example, the sequence contains 12 images (see below) so that every image is shown for 100 ms. For synthetic stimuli **NRepetitions** must be 1.
The last three parameters control display settings: visualization delay, display zoom and spatial arrangement of windows.
The next lines of code define the type of visual input. We use a sequence of 12 images that represents the onset (at 100 ms) and offset (at 1100 ms) of a white square over a black background. The different types of input stimuli that can be selected are described in Section 4.

```
### Visual input ###

# Folder that contains the input sequence
retina.Input('sequence',{'../input_sequences/Weberlaw/0_255/'})
```

Creation of computational retinal microcircuits follows the next syntax:

retina.Create('type','ID','list of parameters').

Every type of retinal cell is usually formed by three processing stages: spatial filtering, temporal integration and nonlinearity. However, the number of modules per cell is not a fixed structure (e.g., a nonlinearity can represent either a pre-synaptic or post-synaptic rectification). The main goal of COREM is to be sufficiently configurable to give the possibility of modelling a cell in multiple ways.

```
### Creation of computational retinal microcircuits ###

# Temporal modules
retina.Create('LinearFilter','tmp_photoreceptors',{'type','Gamma','tau','30.0','n
     ','10.0'})
retina.Create('LinearFilter','tmp_horizontal',{'type','Gamma','tau','20.0','n','
     1.0'})
retina.Create('SingleCompartment','tmp_bipolar',{'number_current_ports','1.0','
     number_conductance_ports','1.0','Rm','1.0','tau','10.0','Cm','100.0','E','
     0.0'})
retina.Create('LinearFilter','tmp_amacrine',{'type','Gamma','tau','10.0','n','1.0
     '})

# Spatial filters
retina.Create('GaussFilter','Gauss_horizontal',{'sigma','0.3','spaceVariantSigma'
     ,'False'})
retina.Create('GaussFilter','Gauss_bipolar',{'sigma','0.1','spaceVariantSigma','
     False'})
```

```
retina.Create('GaussFilter','Gauss_amacrine',{'sigma','0.3','spaceVariantSigma','
     False'})
retina.Create('GaussFilter','Gauss_ganglion',{'sigma','0.2','spaceVariantSigma','
     False'})

# Nonlinearities
retina.Create('StaticNonLinearity','SNL_photoreceptors',{'slope','-0.1','offset',
     '0.0','exponent','1.0'})
retina.Create('StaticNonLinearity','SNL_horizontal',{'slope','1.0','offset','0.0'
     ,'exponent','1.0'})
retina.Create('StaticNonLinearity','SNL_amacrine',{'slope','0.2','offset','1.0','
     exponent','2.0'})
retina.Create('StaticNonLinearity','SNL_bipolar',{'slope','10.0','offset','0.0','
     exponent','1.0','threshold','0.0'})
retina.Create('StaticNonLinearity','SNL_ganglion',{'slope','5.0','offset','0.0','
     exponent','1.0'})
```

Module **from_ID** is connected to module **to_ID** according to the next scheme:

retina.Connect('from_ID','to_ID','Current/Conductance').

Every new connection is added to either the list of current input ports or the list of conductance input ports (only for the module **SingleCompartment**). Operators + and − can be used to describe linear synaptic interactions of modules (e.g., the Outer Plexiform Layer).

```
### Connections ###

# Phototransduction
retina.Connect('L_cones','tmp_photoreceptors','Current')
retina.Connect('tmp_photoreceptors','SNL_photoreceptors','Current')

# Horizontal cells
retina.Connect('SNL_photoreceptors','Gauss_horizontal','Current')
retina.Connect('Gauss_horizontal','tmp_horizontal','Current')
retina.Connect('tmp_horizontal','SNL_horizontal','Current')

# Subtraction at Outer Plexiform Layer
retina.Connect({'SNL_horizontal',-,'SNL_photoreceptors'},'Gauss_bipolar','Current
     ')
retina.Connect('Gauss_bipolar','tmp_bipolar','Current')
retina.Connect('tmp_bipolar','SNL_bipolar','Current')

# Gain control at Inner Plexiform Layer
retina.Connect('SNL_bipolar','Gauss_amacrine','Current')
retina.Connect('Gauss_amacrine','tmp_amacrine','Current')
retina.Connect('tmp_amacrine','SNL_amacrine','Current')
retina.Connect('SNL_amacrine','tmp_bipolar','Conductance')

# Bipolar-ganglion synapse
retina.Connect('SNL_bipolar','Gauss_ganglion','Current')
retina.Connect('Gauss_ganglion','SNL_ganglion','Current')
```

```
# Connection with NEST
retina.Connect('SNL_ganglion','Output','Current')
```

To display the output of a module during simulation, we make use of the syntax **retina.Show()**. With **margin** we set a framework of visualization. In the case we are not interested in showing outer edges (because of distortions produced by the spatial filtering, for example) this parameter can be configured with values greater than 0.
Two spatial multimeters and one temporal multimeter have been added to our example script. While records of spatial multimeters are shown in the time specified by **timeStep**, temporal records are displayed when simulation stops. An example of the type of displays generated by the simulator is shown in Fig. 3. Simulation is paused every time a display pops up. They must be manually closed to continue simulation.

```
### Displays and data analysis ###

retina.Show('Input','True','margin','0')
retina.Show('SNL_photoreceptors','True','margin','0')
retina.Show('SNL_horizontal','True','margin','0')
retina.Show('SNL_bipolar','True','margin','0')
retina.Show('SNL_amacrine','True','margin','0')
retina.Show('SNL_ganglion','True','margin','0')

# Spatial multimeters of row/col 12th at 200 ms
# row selection
retina.multimeter('spatial','Horizontal cells','SNL_horizontal',{'timeStep','200'
     ,'rowcol','True','value','12'},'Show','True')
# col selection
retina.multimeter('spatial','Horizontal cells','SNL_horizontal',{'timeStep','200'
     ,'rowcol','False','value','12'},'Show','True')

# Temporal multimeter of ganglion cell at (5,5)
retina.multimeter('temporal','Ganglion cell','SNL_ganglion',{'x','5','y','5'},'
    Show','True')
```

If COREM has been compiled as an extension module of NEST, it is also possible to simulate the spike output of ganglion cells in the NEST environment. To illustrate this, a piece of code in NEST is shown below (**NEST_scripts/example_1.py**). It simply connects the retina interface of COREM with integrate-and-fire neurons and show a raster plot of spikes generated by ganglion cells.

```
#!/usr/bin/python
# coding: utf-8
```

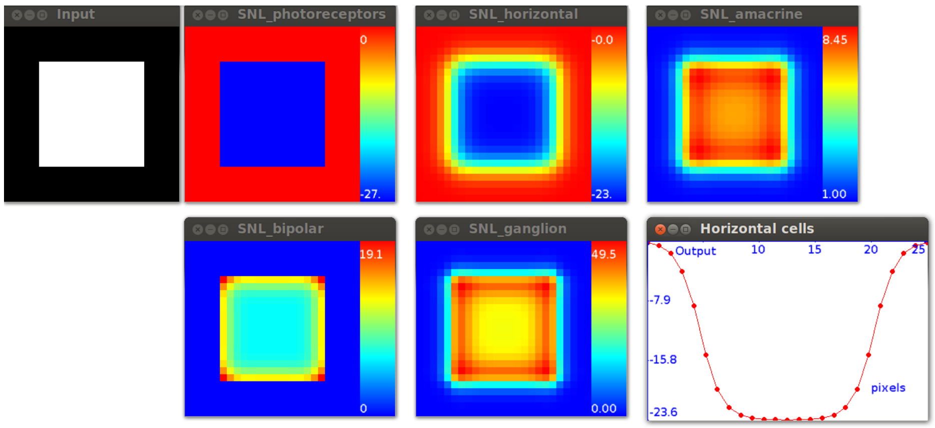

Figure 2: Displays showing the input stimulus and outputs of modules in **example_1.py** at time step 200 ms. The spatial multimeter (bottom right corner) plots output values of horizontal cells for row number 12. Note that values represented in these displays do not necessarily correspond with membrane potentials. Their meaning depends on the interpretation that the user wants to give to them. In this example, the photoreceptor display represents membrane potential of cones but the ganglion display refers to pre-synaptic input current of ganglion cells. Note as well that values of membrane potentials and synaptic currents may not be scaled to real values, this is just an example.

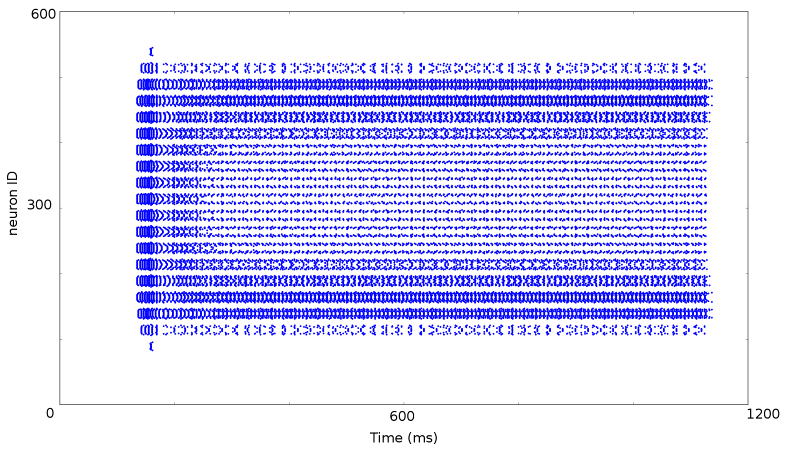

Figure 3: Raster plot of spikes generated by ganglion cells. The different firing patterns across cells represent three distinct regions of the image: enhanced edges of the square, attenuated center of the square and inhibited background.

```python
import nest
import nest.raster_plot
import numpy as np
import matplotlib.pyplot as plt

# simulation parameters
ganglionCells = 25*25
simTime = 1200.0

# Kernel and Network reset
nest.ResetKernel()
nest.ResetNetwork()

# Number of threads (must be 1) and resolution
nest.SetKernelStatus({"local_num_threads": 1,'resolution': 1.0})

# Load COREM
nest.Install("COREM")

# Create spike detector and spiking nodes
mult = nest.Create('spike_detector',1)
spikingGanglion=nest.Create('iaf_neuron',ganglionCells,{'C_m':10.0,'tau_m':10.0})
nest.Connect(spikingGanglion,mult)

# COREM nodes
for i in range(0,ganglionCells):
       g=nest.Create('COREM',1,{'port':float(i),'file':'../Retina_scripts/
            example_1.py'})
       nest.Connect(g,[spikingGanglion[i]])

# Simulation
nest.Simulate(simTime)

# Raster plot
nest.raster_plot.from_device(mult,hist=False)
plt.show()
```

4 Types of visual inputs

There are two main groups of visual stimuli that are provided in COREM: image sequences and synthetic patterns. Synthetic patterns are simple images commonly used in electrophysiological experiments to study low-level processing of the visual system. Below we explain how to configure image sequences and synthetic patterns (sinusoidal drifting grating, jittered grating, white noise and impulse).

```python
retina.Input('sequence',{'path_to_seq'})
```

- Read in alphabetical order all images in directory **path_to_seq**. Simulation parameter **NRepetitions** (time in ms every image is shown) sets the frame rate of the video sequence. For example, a rate of 10 frames/s is indicated by:

retina.NRepetitions('100').

Parameters	**path_to_seq**: directory that contains the image sequence
Example	`retina.Input('sequence'{'../input_sequences/Weberlaw/0_255/'})`

```
retina.Input('grating',{'type','ptype','step','pstep','length1','plength1','
     length2','plength2','length3','plength3','sizeX','psizeX','sizeY','psizeY','
     freq','pfreq','period','pperiod','Lum','pLum','Contr','pContr','phi_s','
     pphi_s','phi_t','pphi_t','orientation','porientation','red_weight','
     pred_weight','green_weight','pgreen_weight','blue_weight','pblue_weight','
     red_phase','pred_phase','green_phase','pgreen_phase','blue_phase','
     pblue_phase'})
```

- Create a sinusoidal drifting grating.

Parameters	**ptype**: type of grating (drifting = 0, static and oscillating = 1, static and reversing = 2) **pstep**: time step of one frame (in seconds) **plength1**: length of time before grating apparition (in seconds) **plength2**: length of first grating apparition (in seconds) **plength3**: length of reversed grating apparition (in case of type=2, in seconds) **psizeX,psizeY**: frame size (in pixels) **pfreq**: temporal frequency (in Hertz) **pperiod**: spatial period (in pixels) **pLum**: mean luminance (unitless) **pContr**: normalized contrast (between 0 and 1) **pphi_s**: spatial phase of central pixel at time zero (in multiples of π) **pphi_t**: temporal phase of signal at time zero (in multiples of π) **porientation**: orientation of the grating (in rad) **pred_weight,pgreen_weight,pblue_weight**: weight of each color channel **pred_phase,pgreen_phase,pblue_phase**:: initial phase for each color channel
Example	`retina.Input('grating',` `{'type','0','step','0.001','length1','0.0','length2','4.0','length3',` `'0.0','sizeX','200','sizeY','200','freq','2.5','period','40.0','Lum',` `'100.0','Contr','0.5','phi_s','0.0','phi_t','0.0','orientation','0.0',` `'red_weight','1.0','green_weight','1.0','blue_weight','1.0','red_phase',` `'0.0','green_phase','0.0','blue_phase','0.0'})`

```
retina.Input('fixationalMovGrating'{'sizeX','psizeX','sizeY','psizeY','
     circle_radius','pcircle_radius','jitter_period','pjitter_period','
     spatial_period','pspatial_period','step_size','pstep_size','Lum','pLum','
     Contr','pContr','orientation','porientation','red_weight','pred_weight','
     green_weight','pgreen_weight','blue_weight','pblue_weight','type1','ptype1',
     'type2','ptype2','switch','pswitch'})
```

- Create a jittering grating to reproduce retinal adaptation to object motion as described in [6, 7].

Parameters	**psizeX,psizeY**: frame size (in pixels) **pcircle_radius**: radius of the center circular grating (in pixels) **pjitter_period**: temporal period of the jitter (in ms) **pspatial_period**: spatial period (in pixels) **pstep_size**: step size of the jitter (in pixels) **pLum**: mean luminance (unitless) **pContr**: normalized contrast (between 0 and 1) **porientation**: orientation of the grating (in rad) **pred_weight,pgreen_weight,pblue_weight**: weight of each color channel **ptype1**: relative jittering between the center and the surround during the first interval (global motion = 1, differential motion = 0) **ptype2**: relative jittering between the center and the surround during the second interval (global motion = 1, differential motion = 0) **pswitch**: time to switch between the different types of jittering (in ms)
Example	`retina.Input('fixationalMovGrating',{'sizeX','200','sizeY','200', 'circle_radius','40','jitter_period','15.0','spatial_period','40', 'step_size','5.0','Lum','100.0','Contr','0.5','orientation','0.0', 'red_weight','1.0','green_weight','1.0','blue_weight','1.0','type1','0', 'type2','1','switch','500'})`

```
retina.Input('whiteNoise'{'mean','pmean','contrast1','pcontrast1','contrast2','
     pcontrast2','period','pperiod','switch','pswitch','sizeX','psizeX','sizeY','
     psizeY'})
```

- Spatially-uniform white noise generator. It is usually used in combination with the Linear-Nonlinear (LN) multimeter (see Section 6).

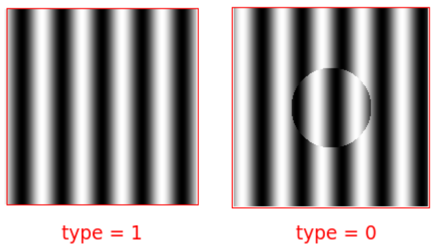

Figure 4: Parameters **ptype1** and **ptype2** define the relative jittering between the center and the surround (global motion = 1, differential motion = 0)

.

Parameters	**pmean**: mean luminance (this value is multiplied by 255) **pcontrast1**: normalized contrast (between 0 and 1) for the first time interval **pcontrast2**: normalized contrast (between 0 and 1) for the second time interval **pperiod**: a new value is drawn from a Gaussian distribution every pperiod ms **pswitch**: time to switch between the different contrasts (in ms) **psizeX,psizeY**: frame size (in pixels)
Example	`retina.Input('whiteNoise',{'mean','0.5','contrast1','0.5', 'contrast2','0.1','period','1.0','switch','500','sizeX','200', 'sizeY','200'})`

```
retina.Input('impulse'{'start','pstart','stop','pstop','amplitude','pamplitude','
     offset','poffset','sizeX','psizeX','sizeY','pSizeY'})
```

- Spatially-uniform impulse pattern.

Parameters	**pstart**: onset of the impulse (in ms) **pstop**: offset of the impulse (in ms) **amplitude**: amplitude of the impulse (unitless) **poffset**: mean luminance (unitless) **psizeX,psizeY**: frame size (in pixels)
Example	`retina.Input('impulse',{'start','100.0','stop','300.0', 'amplitude','255.0','offset','0.0','sizeX','200','sizeY','200'})`

5 Creating computational retinal microcircuits

This section enumerates the main properties and configuration parameters of the 5 computational retinal modules implemented in COREM. They can be interconnected in several different ways to reproduce a wide range of retina models.
A retinal neuron in COREM is typically formed by three processing stages: spatial filtering, temporal integration and nonlinearity. The spatial filtering represents synaptic integration of incoming synapses and gap-junctions between cells of the same type. Temporal integration of the neural signal can be modelled by a linear filter (recommended for experiments in which cells behave linearly) or a single-compartment model. Current and conductance input ports of the single-compartment model simulate synaptic and ionic channels, respectively. The last two modules, static non-linearity (time-independent) and short term plasticity (time-dependent), account for some well-known input-output relationships observed in cells, such as rectification or sigmoidal functions.

```
retina.Create('GaussFilter','module_ID',{'sigma','psigma','spaceVariantSigma','
    pspaceVariantSigma','K','pK','R0','pR0'})
```

- Space-variant Gaussian filter. If the filter is configured with space-variant constants the density function used to simulate retinal cell density is as follows:

  $if(r < R0) : density = 1$

  $if(r \geq R0) : density = 1/(1 + K * (r - R0))$

  with r the radius, $R0$ the foveal radius and K the density constant that determines the decay rate. The kernel size of the Gaussian filter is modified along the retina inversely proportional to cell density. The consequence, from the point of view of retina processing, is a higher factor of signal averaging in the retina surround (see Fig. 5).

Parameters	**psigma**: sigma value of the Gaussian filter at the center of the image **pspaceVariantSigma**: (True or False) to configure a standard spatially-uniform or a space-variant Gaussian processing **pK**: density constant (only for the space-variant case) **pR0**: foveal radius (only for the space-variant case)
Example	`retina.Create('GaussFilter','Gauss_horizontal',{'sigma','0.3','spaceVariantSigma','True','K','0.2','R0','5.0'})`

```
retina.Create('LinearFilter','module_ID',{'type','ptype','tau','ptau','n','pn'})
```

- Temporal recursive linear filtering whose kernel can be either exponential or gamma.

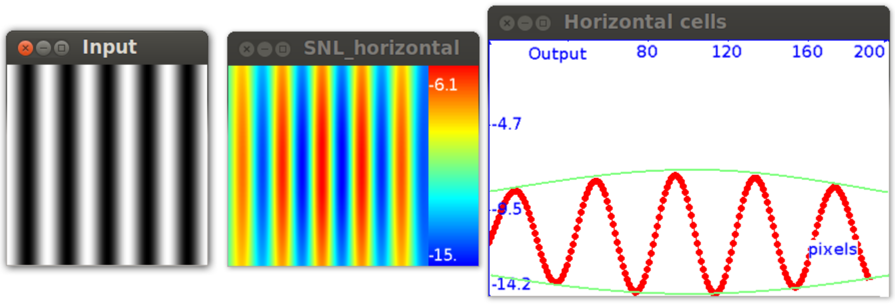

Figure 5: An example of the resulting spatial profile in horizontal cells when the Gaussian filter is configured with space-variant constants. The green lines have been manually added to highlight the higher factor of signal averaging in the retina surround

.

Parameters	**ptype**: exponential ('Exp') or Gamma ('Gamma') **ptau**: temporal constant (in ms) **pn**: filtering stages in cascade (only for Gamma)
Example	`retina.Create('LinearFilter','tmp_horizontal',{'type','Gamma','tau','20.0','n','1.0'})`

```
retina.Create('SingleCompartment','module_ID',{'number_current_ports','
     pnumber_current_ports','number_conductance_ports',
'pnumber_conductance_ports','Rm','pRm','tau','ptau','Cm','pCm','E','pE'})
```

- Generic single-compartment model of the membrane potential, updated by Euler's method:

  $C_m \frac{dV(t)}{dt} = \sum_i I_i(t) + \sum_j g_j(t)(E_j - V(t))$

  where C_m is the membrane capacitance, $I_i(t)$ represent external currents (e.g., electrode currents or linear synaptic inputs), $g_j(t)$ are conductances of ionic channels and E_j their reversal potentials.

Parameters	**pnumber_current_ports**: number of current input ports **pnumber_conductance_ports**: number of conductance input ports **pRm**: Membrane resistance **ptau**: Membrane temporal constant **pCm**: membrane capacitance **pE**: reversal potential for all channels
Example	`retina.Create('LinearFilter','tmp_horizontal',{'type','Gamma','tau','20.0','n','1.0'})`

```
retina.Create('StaticNonLinearity','module_ID',{'slope','pslope','offset','
    poffset','exponent','pexponent','threshold','pthreshold'})
```

```
retina.Create('SigmoidNonLinearity','module_ID',{'slope','pslope','offset','
    poffset','max','pmax'})
```

```
retina.Create('CustomNonLinearity','module_ID',{'start','pstart_1','end','pend_1'
    ,'slope','pslope_1','offset','poffset_1','exponent','pexponent_1',...,'start
    ','pstart_n','end','pend_n','slope','pslope_n','offset','poffset_n','
    exponent','pexponent_n'})
```

- These three functions form a set of commonly used static nonlinearities. The standard nonlinearity is:

  $y_{x>thr}(x) = slope * x^{exponent} + offset$

  A sigmoid function is defined by:

  $y(x) = max/(1 + \exp(-slope * x + offset))$

  and **CustomNonLinearity** implements a piecewise function in which terms **start** and **stop** delimit intervals.

Parameters	**pslope**: multiplicative parameter **poffset**: additive parameter **pexponent**: exponential parameter **pthreshold**: threshold of the function **pmax**: maximum achievable by sigmoid function **pstart,pstop**: interval delimiters in the piecewise function
Example	`retina.Create('StaticNonLinearity','SNL_horizontal',{'slope','1.0', 'offset','0.0','exponent','1.0'})`

```
retina.Create('ShortTermPlasticity','module_ID',{'slope','pslope','offset','
    poffset','exponent','pexponent','kf','pkf',
'kd','pkd','VInf','pVInf','tau','ptau'})
```

- Short-term plasticity (STP). Assuming the input function is normalized to have zero mean, the offset $S(t)$ is continuously augmented an amount proportional to the variance of the input:

  $S(t) = S(t) + k_f(k_s(t)abs(input(t)) - S(t))$

  where k_f is a parameter that specifies the fast adaptation rate and $k_s(t)$ $abs(input(t))$, with $k_s(t)$ the slow adaptation term (explained below), limits the maximum value the offset can reach. The term $abs(input(t))$ provides a rectified measure of the input synaptic activity.

  Slow system transitions during prolonged periods of low and high contrast are governed then by the following equation:

  $k_s(t + \Delta t) = k_\infty(t) + (k_s(t) - k_\infty(t)) \exp(-\Delta t/\tau_s)$

where τ_s is the temporal constant of the slow adaptation mechanism and $k_\infty(t)$ is inversely modulated by the input activity of the cell:

$k_\infty(t) = V_{Inf}/(k_d * abs(input(t)))$

V_{Inf} and k_d are maintained as different constants (instead of blending them into one) because of terminology reasons.

Parameters	**pslope**: multiplicative parameter (same as in the static non-linearity) **poffset**: additive parameter (same as in the static non-linearity) **pexponent**: exponential parameter (same as in the static non-linearity) **pkf,pkd,pVInf,ptau**: parameters of the STP
Example	`retina.Create('ShortTermPlasticity','SNL_ganglion',{'slope','1.0','offset','-87.0','exponent','1.0','kf','0.1','kd','2.14','VInf','9.0','tau','12000'})`

6 Data analysis: multimeter

Three different types of multimeters are used to record output values from modules during simulation and to plot results. While records of spatial multimeters are shown in the simulation time specified by **ptimeStep**, temporal records are displayed when simulation stops. Linear-Nonlinear (LN) analysis has been implemented according to the method description by Baccus et al.[8]. An example of the LN analysis is shown in Fig. 6.

```
retina.multimeter('spatial','title','module_ID',{'timeStep','ptimeStep','rowcol',
    'prowcol','value','pvalue'},'Show','pShow')
```

```
retina.multimeter('temporal','title','module_ID',{'x','px','y','py'},'Show','
    pShow')
```

```
retina.multimeter('Linear-Nonlinear','title','module_ID',{'x','px','y','py','
    segment','psegment','interval','pinterval','start','pstart','stop','pstop','
    Show','pShow'}
```

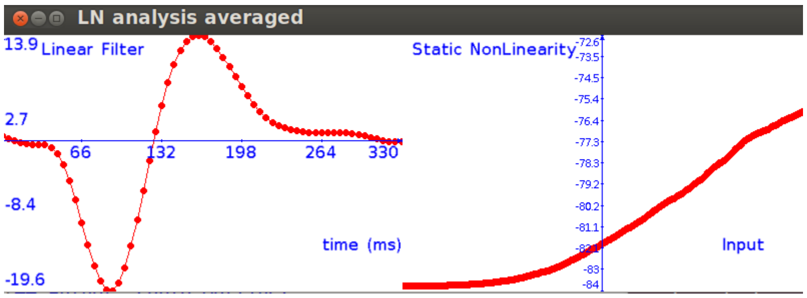

Figure 6: LN analysis (Linear Filter and Static Nonlinearity) of the retina model **contrast.py**

Parameters	**ptimeStep**: time to show spatial multimeter records **prowcol**: select to display a row or a column in the spatial multimeter display (True or False) **pvalue**: number of the row/column **px,py**: pixel coordinates to show records (temporal and LN analysis) **psegment**: segment length to average values of the input sequence (LN analysis) **pinterval**: averaging period (LN analysis) **pstart,pstop**: delimit the time interval to calculate the LN analysis **pShow**: show multimeter display (True or False)
Example	`retina.Create('ShortTermPlasticity','SNL_ganglion',{'slope','1.0', 'offset','-87.0','exponent','1.0','kf','0.1','kd','2.14','VInf', '9.0','tau','12000'})`

References

[1] P. Martínez-Cañada, C. Morillas, B. Pino, E. Ros, F. Pelayo *et al.*, A computational framework for realistic retina modeling, *International Journal of Neural Systems. (Under review*, (World Scientific, 0), pp. 0–0.

[2] P. Martínez-Cañada, C. Morillas, J. L. Nieves, B. Pino, F. Pelayo *et al.*, First stage of a human visual system simulator: The retina, *Computational Color Imaging*, (Springer, 2015), pp. 118–127.

[3] P. Martínez-Cañada, C. Morillas, B. Pino, F. Pelayo *et al.*, Towards a generic simulation tool of retina models, *Artificial Computation in Biology and Medicine*, (Springer, 2015), pp. 47–57.

[4] P. Martínez-Cañada, C. Morillas, S. Romero, F. Pelayo *et al.*, Modeling retina adaptation with multiobjective parameter fitting, *Advances in Computational Intelligence*, (Springer, 2015), pp. 175–184.

[5] M.-O. Gewaltig and M. Diesmann, Nest (neural simulation tool), *Scholarpedia* **2**(4) (2007) p. 1430.

[6] B. P. Ölveczky, S. A. Baccus and M. Meister, Segregation of object and background motion in the retina, *Nature* **423**(6938) (2003) 401–408.

[7] B. P. Ölveczky, S. A. Baccus and M. Meister, Retinal adaptation to object motion, *Neuron* **56**(4) (2007) 689–700.

[8] S. A. Baccus and M. Meister, Fast and slow contrast adaptation in retinal circuitry, *Neuron* **36**(5) (2002) 909–919.