Convis: A Toolbox To Fit and Simulate Filter-based Models of Early Visual Processing

Jacob Huth [1,*], Timothée Masquelier [2] and Angelo Arleo [1]

July 27, 2017

[*] the corresponding author can be reached via: jahuth@uos.de
[1] Sorbonne Universits, UPMC Univ Paris 06, INSERM, CNRS, Paris, France
[2] CERCO UMR5549, CNRS, University Toulouse 3, France

Abstract

1

We developed *Convis*, a Python simulation toolbox for large scale neural populations which offers arbitrary receptive fields by 1D, 2D and 3D convolutions executed on a graphics card. The resulting software proves to be flexible and easily extensible in Python, while building on the Theano library, previously used successfully in deep learning applications, for just-in-time optimization and compilation of the model onto CPU or GPU architectures.

Through automatic differentiation, any parameter of a specified model can be optimized to approach a desired output which is a significant improvement over e.g. Monte Carlo or particle optimizations without gradients. We show that a number of models including even complex non-linearities such as contrast gain control and spiking mechanisms can be implemented easily.

We show in this paper that we can recreate the simulation results of a popular retina simulation software VirtualRetina (Wohrer and Kornprobst, 2009), with the added benefit of providing (1) arbitrary linear filters instead of the product of Gaussian and exponential filters and (2) use optimization routines utilizing the gradients of the model. We demonstrate the utility of 3d convolution filters with a simple direction selective filter. Also we show that it is possible to optimize the input for a certain goal, rather than the parameters, which can aid the design of experiments as well as closed-loop online stimulus generation.

Convis is open source under the GPL-3.0 license and available from `https://github.com/jahuth/convis/`

2 Keywords:

vision model toolbox, retina model, primary visual cortex model, Python, GPU, Theano

3 Introduction

We developed Convis, an extension to the popular Theano toolbox (Theano Development Team, 2016) that can implement models of responses to visual inputs on a large scale and, using automatic differentiation, derive the gradient of the output with respect to any input parameter. Linear filtering is done as convolutions, giving full flexibility in the shape and temporal structure of receptive fields. As an example, we show the utility of this method by examining models of retinal ganglion cells (RGCs), ranging from a very simple linear-non-linear model to a many-parameter mechanistic model which includes recurrent contrast gain control. Further, we show that this toolbox can simulate cell populations on a similar scale and in comparable time to previous large-scale retina simulators, even though the computations refrain from simplifying the shape of receptive fields.

3.1 Large Scale Retina Models

The retina is the first stage of neural computation in vision. Models of different complexity exist that predict retinal ganglion cell responses for a wide range of ganglion cell types (from simple ON and OFF alpha cells up to direction sensitive cells). Some efforts were made to create large scale population simulators (eg. VirtualRetina (Wohrer and Kornprobst, 2009) and COREM (Martínez-Cañada et al., 2016)) that can create the responses of thousands of retinal ganglion cells with efficient computations. These can be used as input to higher visual functions which generally need a large number of cells as input. However, these simulators are not very useful for experimentalists who want to model the behavior of cells recorded in their experiments. The parameters of the model can not be directly inferred from an experiment and fitting the simulation to the data can only be done via parameter grid search or Monte Carlo methods - both being very computation intense. Retinal processing is - in contrast to the rest of the visual system - easily accessible to multi-array recordings. Despite the relative consistency of retinal circuitry, a high number of ganglion-cell response classes exists (Masland, 2001). To characterize the properties of each class of cells, linear-non-linear cascade models are the current gold standard. However, the properties of contrast gain control are not easily extracted from data: Experiments usually explore only distinct levels of contrast, for instance Garvert and Gollisch (2013). Shapley and Victor (1978) proposed a mechanism for contrast gain control, which accounts for the phase advance and transfer function recorded in RGC cells. It is incorporated into large scale retina population simulators such as VirtualRetina in a formulation that takes local contrast, rather than global contrast, into account. To keep computations tractable, VirtualRetina and COREM use recursive filtering rather than actual convolution, which limits the possible applications of these models to circular receptive fields and exponential temporal filters. In contrast Convis uses dense numerical filters, which are convolved with the input on a graphics card. Cell responses with detailed receptive fields can thus be simulated efficiently and the model can be fitted to experimental recordings.

3.2 The theano computational graph

Theano (Theano Development Team, 2016) is a Python library originally developed for deep learning. A computational graph can be specified with a set of abstract symbols, which are then optimized and compiled into concrete code to run on CPU (as compiled C/C++ code) or GPU (as CUDA or OpenCL code). The computational graph is defined by connecting Operations ("Ops") and Variables using "ApplyNodes". For instance three variables and the Op "Addition" can be connected via an `ApplyNode` such that adding the first two variables (inputs of the `ApplyNode`) will yield the third variable (the output of the `ApplyNode`). This graph can then be compiled into a function that accepts two input arguments and returns their sum. Variables are typed such that compiling a graph will choose an appropriate method (e.g. either adding two scalars or adding two tensors). Before the graph is compiled onto the target device, the graph is optimized according to a set of optimization rules that can be configured and extended.

3.3 Computations in the Retina

RGCs transmit the output of the retina to different brain regions. It is estimated that there exist about 30 different RGC types (Sanes and Masland, 2015), each population forming a tiling coverage of the visual space, responding uniquely to light stimuli and possesses unique proteins making it possible to distinguish between different types histochemically. The visual system, even at this low level, is complex. A model that predicts, for a given set of stimuli, the corresponding responses accurately might not be able to do so for some novel stimuli, due to over-fitting. The responses to "natural images" lead to more complex models than the responses high-luminance on-off pulses, moving bars or even random checkerboard stimuli (Simoncelli and Olshausen, 2001). Still, a "natural stimulus" will also not cover all modes of behavior and even more complex models are conceivable. Choosing the complexity of a model is a hard problem in itself. The model should be as simple as possible while still explaining the data to avoid over-fitting as data is always limited. The simplest approximation of RGC responses can be obtained with a linear-non-linear model that predicts the firing probability of a cell by multiplying the input with a spatial or spatio-temporal receptive field and then applying a non-linearity (Korenberg and Hunter, 1986). When a stimulus is spatially more complex and has a finer resolution, it becomes apparent that subunits exist that integrate the stimulus independently under a first non-linearity before being summed and subject to a second non-linearity. Responses to these stimuli can be predicted with subunit models (also called LN-cascade models as they stack multiple levels of linear and non-linear processing). We implemented a range of models on this spectrum of complexity and assessed whether parameters could be estimated efficiently.

3.4 A complex retina model: Virtual Retina

We implemented a retina model close to the mathematical definition of VirtualRetina (Wohrer and Kornprobst, 2009), but replaced the recursive method to compute linear filters with convolution operations.

The recursive definition of filters makes VirtualRetina very fast and efficient. Two-dimensional Gaussians

for instance are computed independently for the x and y direction, such that the time complexity is scaling essentially linear with each dimension. But recursive filters have the drawback that their shape is very limited: the x-y separability is a huge constraint for receptive fields. A radially symmetric filter is a good approximation for a diffusion processes, but neural connections tend to be sparser and more selective. To simulate non-round, non-Gaussian and non-continuous spatio-temporal filters we used 1d, 2d or 3d convolutions, which have the drawback of being very inefficient when implemented in single threaded programs. To counter this flaw and to keep the code comprehensible and maintainable, we used the Theano library to define computations of the model on an abstract level and create optimized code that can run on a GPU. Spatio-temporal filters with finite impulse responses can be implemented effectively with either a recursive definition that keeps N previous states or higher moments in memory (as used in VirtualRetina and COREM) or as a convolution with a 3D kernel. Although a convolution is computationally costly because it has to calculate the crossproduct of filter and image at each position of the 3D image, the use of GPUs for cheap parallel computations can speed up this process dramatically.

Five main types of processing happen in the layers of the retina: the input is low-pass filtered in the spatial and temporal domain, converted from luminance to contrast (and effectively spatio-temporally high pass filtered), gain controlled and rectified, spikes are generated that are then sent via the optic nerve. Each ganglion cell type differs in the parameters of these operations. Each filter operation can react differently to spatial and temporal frequencies. While the simplest possible model (a linear-non-linear model) can capture the general shape of the classic, excitatory receptive field under one specific condition, changes in illumination and surrounding contrast as well as the presentation of natural images can create responses that differ from the ones predicted by an LN model (Heitman et al., 2016). Adding subunits to the model, which compute independent non-linear functions before their output is aggregated and again subjected to a non-linearity, improves the response to some naturally structured images, while adding contrast gain control enables correct predictions when contrast is locally or globally increased. Consequently, RGC responses are most accurately modelled as a cascade linear-non-linear model with gain control. The stages of the VirtualRetina simulator for example consist of a linear, spatio-temporally non-separable filtering mimicking the Outer Plexiform Layer (OPL) which converts luminance into a contrast image via a center-surround receptive field. The second stage, modeling the contrast gain control properties of bipolar cells subject to shunting inhibition, is a leaky integrator with leak conductance that depends on a spatio-temporal neighborhood of contrast magnitude. The last stage creates spikes by applying a static non-linearity to the the output of the bipolar cells and then using a Leaky Integrate and Fire model to convert current into precise spike times. The contrast gain control as implemented in VirtualRetina has two effects in retinal processing: (1) it filters frequencies and (2) a phase advance is applied to the signal. This is typically assessed by exposing the system to a sum of sine waves and varying the contrast. Not only does the gain change, leading to sub-linear output, but with increasing contrast the phase advance decreases the response time compared with lower contrast stimuli.

Fitting the VirtualRetina model completely from recorded data is not feasible. Many of the parameters have correlated effects, the number of overall parameters is high and the non-linear nature of some parameters increases the difficulty further. Thus, the parameters used for simulations are usually taken from the literature (such as the average receptive field size and size of the suppressive surround) and

remaining parameters are tuned with Monte Carlo optimization to achieve the desired firing rates (Wohrer and Kornprobst, 2009).

The reimplementation in the Convis toolbox can in contrast be fitted to data by using the gradients of the parameters. The first gradients give information about the location of a nearby minimum (see Section 5.2), while the second derivatives can give information about the shape of the error function, specifically if there is a global minimum, and the interaction of parameters by calculating a Hessian Matrix (see Section 5.3).

4 Methods

4.1 Extension of Theano to facilitate model creation

We built the toolbox on top of Theano. Theano can define variables and combine them together with abstract operations (eg. sums, products, convolutions, etc.) into a computational graph. This graph can then be optimized and compiled to different machine languages, depending on the available devices. For deep learning models this allows for easy definition of a model that can run on different kinds of hardware, from a single desktop CPU to a cloud cluster with multiple CPUs. This graph can be manipulated, annotated with additional information and segmented to suit a specific purpose. One specific application are deep neural nets that learn the statistics of a batch of images. Toolboxes that tackle this problem generally define layers for neural network models that can be combined into optimizable models. Examples are Blocks (`https://github.com/mila-udem/blocks`), lasagne (`https://github.com/benanne/Lasagne`), Keras (`https://github.com/fchollet/keras`), Mariana (`https://github.com/tariqdaouda/Mariana`), OpenDeep (`http://opendeep.org/`), and Theanet (`https://github.com/rakeshvar/theanet`).

In our case we deal exclusively with three-dimensional or five-dimensional tensors. But the input might be larger than the available memory or even infinite in duration if we stream input from for instance a webcam. Thus, we only compute the response to a time slice at a time, the size of which depends on the available graphics card memory and image dimensions. While the streaming input is changing for each function call, the parameters (which are also inputs to the computational graph) should be fixed. Since this use case is similar for any application of the toolbox, we facilitate this by distinguishing "input" inputs, "parameter" inputs and "states". Variables are annotated with their type and other meta information when creating a graph. Also, variables are labeled to belong to a certain "subgraph" such that the overall model can be more easily understood and manipulated. We distinguish input and parameter inputs since visual inputs are bound to be three dimensional and are split up into temporal chunks, while the parameters of the model can have any dimensionality, should be save- and loadable and can have updates defined which adapt them according to some rules, for instance gradient descent.

4.2 Inspectability and Ease of Use

One goal in the design of Convis was that it can be used in an interactive Python environment and that during runtime all parameters and configuration options are discoverable - only a minimum of names

should have to be memorized. In IPython (Pérez and Granger, 2007), and other environments that inspect the attributes of Python objects, all parameters of a model can be tab-completed. Pressing Tab after typing `m.parameters.` will give a list of parameters of the model `m` in a hierarchical structure (eg. sorted by layers) and allows to get or set the values of each parameter, as well as access a description string that explains what the parameter is doing. The function `convis.describe()` also takes any object or module and provide information about it. When provided with a hierarchical structure such as `m.parameters`, it will describe all variables found in the hierarchy.

4.3 Input and Output streams

Convis supports multiple input and output formats, such as sequences of images, videos, `.npy` and `.inr` files and input via a network connection. Internally these are all represented by the same interface as stream objects from which either a number of frames can be taken and to which a number of frames can be appended. They can be automatically resampled to different frame rates (to match the discrete time step of the model) and if more than one input is required multiple streams can be synchronized using timestamps.

4.4 Annotating Variables

To manage the computational graph, Theano variables are sorted into inputs, outputs, parameters and states. Parameters are read from configuration files, but can also be optimized while running the model. State variables are saved after a chunk of input has been processed, such that subsequent invocations pick up where the previous one left of.

All annotated parameters of a model can be collected and saved to or loaded from configuration files. Initialization functions in case no value is provided are also attached to the variable itself and each parameter has to contain a doc-string with an explicit explanation of what this parameter does. While any annotated variable can have a doc-string, parameters are required to have one.

4.5 Subgraphs

A subgraph in our context is a subset of nodes of the full graph, associated with a `SubGraph` object. A function that creates connected theano variables and designates some of them as its "output", can return a `SubGraph` object, or preferably be turned into its own class that inherits from the `SubGraph` class. A reference to the object will be assigned as a label to all annotated variables within the subgraph. This way, certain filter structures, for instance a simple exponential filter or a 3D kernel filter, can be combined by linking inputs and outputs, creating a fused graph, while parameters and states still maintain their association to the object that created them. `Subgraph`s can be organized hierarchically. The hierarchical structure can be used to automatically parse or create configuration files and to replace a complete subgraph with a different functionality. If an "OPL layer" contains a "leaky heat filter" which contains a variable "tau" it can be addressed as `model.variables.opl.leakyheat.tau`. While it is still possible that the same variable is referred to with different names in the source code, the name and other meta information collected as

annotations on the theano variable centralize the information, accessible either through the hierarchically organized variable directory of the model (`model.variables.sub_graph_1.sub_graph_2.some_variable`) or a flat collection of variables (`model.variables._all`). A more specialized case of a `Subgraph` is the `Layer` class, which, in addition to outputs, also has to define one or more 3D or 5D tensor inputs, making it possible to chain these graphs together. If a `Layer` is asked to use more than one variable for a specific input slot, the inputs are summed by default.

4.6 States

Between computations the model is expected to remember a state. This corresponds to the last n time slices of input for convolutional kernels with length n in the time dimension or the n last input steps and k last output steps for recursively defined filters. If a state variable is a shared theano variable (which is accessible to the theano graph as well as the Python environment at the same time), it has to be annotated with an "update" i.e. a variable that computes the state for the next step. Otherwise the state variable has to be annotated with a reference to the variable which will make up the state for the next step and a function to initialize itself. This way, states are additional inputs and outputs of the function and are stored outside of the theano managed memory. Since the state is stable between processing steps, the model can accept very slow streaming input without a defined end and the model can also be suspended to a file.

4.7 Kernel Convolutions

In Convis, receptive fields can have an arbitrary shape. The receptive field of an RGC for example is comprised of an excitatory center and a suppressive surround. However, these two components also have different temporal dynamics making the overall receptive field space-time inseparable. While it is possible to construct it as the difference of two separable filters, Convis can use completely inseparable filters. The temporal impulse response of a cell can be approximated with a convolutional kernel only if it is finite. However, it is plausible that the response of sensory neurons to a stimulus will die out in finite time and will be replaced by the response to more recent stimuli. While the combination of recursive exponential filters can give any desired band pass behavior, the change of a model using one exponential filter to a model using two is discontinuous and is thus hard to optimize. A convolutional temporal filter can be gradually adapted into more complex shapes and then analyzed post-hoc if it is possible to approximate it with more efficient filters. Since the non-separable receptive fields are provided as full kernels, rather than as a combination of Gaussians and exponentials, any shape of receptive field can be implemented by providing the corresponding kernel. Movement sensitive receptive fields can be created for varying speeds and directions: A Gaussian that moves across the receptive field is most excited by a stimulus that travels in unison with it. This selectivity is enhanced if it is surrounded or followed by inhibitory regions (see Figure 7 as an example). The width of the speed and orientation selectivity can be adjusted by the shape of the filter.

If color or other additional channel information, such as RGC types, is part of the model, we use 5D tensors as inputs and kernels instead of the default 3D tensors. When we pass input from one layer to

the next we follow a convention for 3D convolution, which is to add one dimension for "batches", which are independent and can be processed in parallel, and one dimension for "channels", such as color, which should be treated inter-dependent. We add those dimensions as dimension 0 and 2. An input with a set number of color channels requires the convolution filter to have the exact same number of input channels and the output of all the individual channel filters will be summed. Conversely a single filter can also generate multiple output channels, which will be passed to the next layer.

4.8 Recursive Filtering

Implementing an infinite impulse response filter for discrete time steps is not possible with a simple convolution kernel. So, in addition to convolutional filtering, we also reimplemented recursive temporal and spatial filters. A temporal recursive filter relies on a finite number of past input and output values and multiplies them each with a vector of coefficients. The recursive filter we implemented is defined as:

$$Y(k) = \sum_{j=0}^{M-1} b_j X(k-j) - \sum_{i=0}^{N} a_i Y(k-i) \tag{1}$$

with X being the input and Y being the output. M is the number of the coefficients in vector b, N is the number of coefficients in vector a. For $N = 0$ the filter has finite impulse response. The computation has to be done in temporally correct order and the previous timesteps must be accessible. In contrast to convolution, which always only depends on a small, local area of the input and is independent of all previous computations, this is a challenge to parallelizing the computations of a recursive filter. In contrast to exponential filters, Gaussians are symmetric in both directions: If we interpret them as filters in time, they have a causal and an anti-causal component, in space they extend forward as well as backwards. A two dimensional Gaussian has an additional option for optimization: since a recursive definition in two directions simultaneously is very costly, splitting the computation into two stages that are computed independently and then combined improves the time complexity significantly. Deriche (1993) uses for this the sum of a causal and an anti-causal filter while Van Vliet et al. (1998) use the product of a causal and anti-causal filter. The drawback of this optimization is that the Gaussians in x and y direction have to be independent, i.e. either a circular Gaussian or an ellipse whose major axis is aligned with either x or y. The recursive filters we implemented have exactly the same draw backs as the ones implemented in VirtualRetina.

4.9 Configuring and running a model

The level of configuration is variable when using Convis. It is possible to use the complete retina model and supplying it with a configuration xml file. Alternatively the different sub graphs can be combined by the user into a new model. The following is a code sample:

```
e = convis.filters.simple.E(name='E') # a temporal exponential filter
e.parameters.tau.set_value(0.1)
m = convis.create_model(e.output+1.0)
```

```
    # as e.output is a theano variable, the graph can be extended
    # by adding, multiplying or using any of the theano.tensor.*
    # functions such as log, exp, sin etc.
m.create_function()          # compiles the graph into executable code
some_output = m.run(some_input)
```

In this code sample, a sub graph for an exponential filter is created with the name `'E'`. It contains a parameter `tau` which can be discovered in Python environments that support tab-completion on Python objects. Then the model `m` gets as a new output `e.output+1.0`. As e.output is a Theano tensor variable, `e.output+1.0` results in a new Theano variable that represents the sum of e.output and a constant `1.0`. Many operations defined on Numpy arrays or Matlab matrices are also available for Theano variables. Before the model can be used, it has to be compiled. It is possible that this can take a few seconds for a complicated model (e.g. the retina model in Convis). After the model compiled the function, `m.run` can be used as a normal Python function and it will return the list of all the registered outputs.

4.10 Layers and Models implemented

Creating a simple LN model can be defined like this:

```
l = convis.filters.simple.K_3d_kernel_filter(
           {'kernel': rand(10,5,5)},
           name='L')
n = convis.filters.simple.Nonlinearity(name='N', inputs=l)
m = convis.make_model(n)                # creating the model
```

and subunit models accordingly with additional convolutions and nonlinearities. Models similar to the definitions from (Real et al., 2017) are already implemented: LN, LNLN (called LNSN for subunit), LNLN with feedback at the second stage (LNSNF) and both stages (LNFSNF) and LNLN models with delayed feedback (LNFDSNF). In contrast to Real et al., where the stimulus had reduced dimensionality, our filters are 3D spatio-temporal filters.

```
from convis.numerical_filters import gauss_filter_3d
m = convis.models.Real_LNSNF_model()
m.parameters.BCM.kernel.set_value(gauss_filter_3d(0.1,0.1))
```

Creating a custom nonlinearity with parameters that can be optimized - and are self-documenting - can be done like this:

```
l = convis.filters.simple.K_3d_kernel_filter(
                {'kernel': rand(5,5,5)}, name='L')
v0 = convis.as_parameter(0.0,name='v0',
                doc='''
                        The cutoff of the nonlinearity.
```

```
                        Values below v0 will be set to i0.
                        Values above will be scaled quadratically.
                '''')
i0 = convis.as_parameter(0.0,name='i0',
                doc='The minimal value of the nonlinearity.')
m = convis.make_model(i0+(1-v0).clip(0,10000)**2)
```

4.11 Automatic Optimization

Since the computational graph is represented symbolically and most of the operations provided by Theano provide a gradient, it is possible to get the gradient of any input with respect to any scalar output. Adding a single derivative to the model is almost unnoticeable in the time the model needs to compile or to run. Adding a second order derivative however can add 2 or 3 minutes to the compile time of the VirtualRetina-style model and a small increase to the time it needs to run. Depending on the task this might or might not be feasible. A small example is provided in figure 12 where the response of a target exponential filter to a random event train is approximated by a second exponential filter and figure 13 where a receptive field was recovered by approximating the response of the first filter to noise. These two examples use the simplest possible gradient descent algorithm using three additions to the static model: (1) the error term (defined for instance as the mean absolute distance to a target output), (2) the definition of a gradient with respect to the error term (provided by the model itself) and (3) an update equation (in this case a linearly scaled step in the direction of the gradient). While it is possible to define a global error term and do a gradient descent on all possible variables, the large dimensionality of the parameter space will make it very unlikely to achieve efficient improvements using gradient descent. Instead of simple gradient descent more complex optimization mechanisms can be used such as different Monte Carlo Methods (e.g. particle filters) or Hessian Free Optimization (Martens, 2010). Additionally, the complexity of the error functions can be examined before running a simulation to reveal interactions between parameters which we will show in section 5.3.

While Convis provides helper functions to create an optimization scheme, implementing a gradient descent algorithm to optimize a model according to some error term by hand can be done in a few lines of code. First, one defines the error function. This can be either the difference to a desired output or a self-organization term.

```
error = T.sum(abs(m.variables.opl.output
                - convis.as_input(dtensor3('target'))))
```

Then, the variables to be minimized can compute the gradient of this error with respect to themselves.

```
grad = T.grad(error, m.variables.opl.E.tau)
```

And finally an update function can be defined, e.g. in the simplest case gradient descent.

```
m.updates.append((m.variables.opl.E.tau,
```

```
                        - 0.05 * grad))                    # gradient descent
m.updates.append((m.variables.opl.E.tau,
                        - 0.01 * m.variables.opl.E.tau)) # normalization
```

Updates are collected at compile time and summed for each variable. After a chunk of input was processed, the update is applied. To ease the process further, helper functions can create some commonly used error functions and optimization algorithms. The given example works well for very simple models whose error function has no local minima.

When comparing a firing rate with a spike train, the error function can be the log-likelihood rather than the distance, when comparing spike trains it is sensible to convolve them with an exponential kernel such that a measure similar to the Van Rossum spike distance (van Rossum, 2001) can be minimized. Instead of gradient descent, it is possible to define for instance a gradient descent with momentum or even Hessian Free optimization. Many parameters in the model have an approximately linear effect on the output, thus the error function with respect to this parameter, when comparing the outputs of one instantiation of the model with another one with the sum of square distances, will be a quadratic function with a minimum at the true value of the parameter. In the absence of noise the minimum can be determined with either three sample points of the value of the error function, or two sample points of the derivative of the error function. Most of the scalar parameters in the model have a quadratic error function. Only parameters of the static non-linearity and gain control have non-quadratic error functions. However, they are still concave, and they have a monotonous derivative and they can be fit with a polynomial of order 3 or at worst 4. Even though each single parameter can have an easily explored error function, varying two parameters simultaneously can lead to problems. Take as an example two variables a and b which give an error of $E(a, b) = a^2 + b^2 + 10.0 * (a + b)^2$, which forms an elongated Gaussian around 0. The two dimensional gradient of E will for almost all points be orthogonal to the x=y diagonal. This means that the 2d gradient will not point towards the global minimum, as the minimum when varying x and keeping y constant is distinct from the one achieved when varying y and keeping x constant. Because of the interaction of different parameters, it can be helpful to explore the parameter space and note for each pair of parameters whether they can be deemed independent, whether they have a fixed relationship to each other (which can then be exploited) or whether they have an interdependence that relies on even further factors, such as the input stimulus or other third parameters.

5 Results

5.1 Simulating a population of RGCs

To verify that our new implementation of the VirtualRetina model still replicates old results we made one-to-one comparisons using different stimuli and retina configurations. Then we extended the simulation to show that movement sensitive cells and heterogeneous cells can be simulated as well.

5.1.1 Comparing convis.retina and VirtualRetina

To verify that the temporal responses are identical to the same model implemented in VirtualRetina, we used a full field chirp stimulus as used in Baden et al. (2016) (see Figure 1) to characterize the temporal characteristics of retinal ganglion cells and a zooming view of a fractal (see Figure 2) to compare the spatial responses. The chirp features an OFF-ON-OFF pulse followed by oscillations increasing in frequency and then oscillations increasing in amplitude. The configuration was supplied as an xml file, as is typically used for VirtualRetina, and both simulations were configured with the same file. Yet the Convis version created corresponding convolutional filters as opposed to recursive filers. As it can be seen in Figures 3,4, and 5, while the bipolar and ganglion cell stages replicate the results of VirtualRetina with high precision, the OPL layer has some numerical differences for very abrupt changes due to the finite precision of the different filter implementation. This can be remedied by either enabling training, using longer (and therefore more accurate) convolution filters or using the recursive filtering implementation which is again identical to VirtualRetina. In Figure 6 we show a convolution kernel that is fitted to match the response of VirtualRetina, showing that when ignoring the configuration values which can cause numerical errors, the response can be faithfully reproduced by a linear convolution filter. Overall, the output of the models is close enough that the difference is unobservable when spikes are generated and compared as either instantaneous rates or spike times.

5.1.2 Direction Selective Filters

Since the non-separable receptive fields are provided as full kernels, rather than as a combination of Gaussians and exponentials, any shape of receptive field can be efficiently implemented by providing the corresponding kernel. Movement sensitive receptive fields can be created for varying speeds and directions, an example can be seen in Figure 7: A Gaussian that moves across the receptive field is most excited by a stimulus that travels in unison with it. To make the response ignore any other stimulus characteristics, a negative Gaussian closely follows the first, such that only an edge can excite the cell. To respond to both negative and positive edges, the absolute value of the response is taken. Figure 8 shows the resulting direction tuning curve.

5.1.3 Heterogeneous Populations

Figure 9 shows how two different outputs can be spatially integrated. We used a weighted sum of the On and Off RGC output of a VirtualRetina model to define a new model. Some locations of the visual space only show On behavior while others show only Off behavior, according to a spatial map. This mechanism can be used to integrate any combination of cell populations spatially: healthy and diseased cells in models of pathologies such as retinitis pigmentosa or age-related macular degeneration, cells of different sized receptive fields or cortical cells responding to different orientations.

To create this model, the graph of an already existing model can be used:

```
retina = convis.retina.Retina()
ssf1 = convis.filters.simple.SoftSelectFilter(
```

```
                    config={'a': spatial_map},
                    name='select',
                    inputs={
                        'input_a': retina.ganglion_input_layers[0],
                        'input_b': retina.ganglion_input_layers[1]
                    })
m = convis.make_model([ssf1])
```

5.2 Estimating Parameters from Synthetic data

5.2.1 Fitting a spatial receptive field

When we simulate a single cell with a randomly generated receptive field, this cell will respond to some white noise stimuli more than to others. If we know the response to a specific sequence of white noise pictures, we can define an optimization on a spatial kernel such that the output of a second model tries to approach the output of the first. Figure 13 shows how a convolutional receptive field with delicate structure (the letters "con" and "vis") can be recovered from a noisy goal activity. With naïve gradient descent this process is not efficient, as with decreasing distance to the true kernel the error gets weaker and thus the convergence gets slower. However, the direction of the high dimensional parameter space (16x16 pixel) is very stable over the whole trajectory and the error function is close to a squared function we can estimate the minimum after observing the magnitude of two gradients along the linear extension of the initially observed gradient. In case of high noise, the gradient estimates will be noisy as well, requiring more than two samples.

5.2.2 Fitting a recursive exponential Filter

To show a simple example of optimization in the temporal domain we created a Poisson spike train which is convolved with an exponential filter of a certain time constant. Then we used the gradient with respect to the time constant of the recursively defined exponential filter to optimize this variable. As this is a fairly simple task, the only possible complication that can arise comes from choosing the learning rate parameter: convergence can be too slow or oscillate around the true value. This can be remedied by introducing a momentum term or adapting the learning rate over time. Figure 12 shows how the response of the model approaches the target. Again, naïve gradient descent works, but it is not an efficient method as the direction of the gradient does not change, but the magnitude is proportional to the distance to the ground truth parameter and thus slows down the closer it is to the true value.

5.2.3 Fitting non-linear Parameters

The error function over non-linear parameters (Figure 14) does not look like a square function. Still the function is concave and the gradient points toward the minimum, allowing gradient descent methods. While it is possible to fit a polynomial to all points so far encountered, the quality of the solution depends on the location in the parameter space and the noise in the fitting process.

5.3 Using the Second Derivative

The gradients in Figure 13,C show that the gradients lie on a straight line. If the linear filter would have been subject to a stronger non-linearity, this would no longer be the case. The second derivative of the error with respect to twice the same parameter will give rapid information about this. While in the simple case the 2nd derivative will be almost constant, it will vary more strongly for more complex non-linearities. Additional information about the derivatives of the error will decrease the number of samples needed to estimate the shape of the error function. As an example, in the simple case of a noise-less linear parameter, the error function is a 2nd degree polynomial. To estimate it, we would either need three samples of the error at different parameter values, or only two samples of the gradient, since we only have to fit a line rather than a parabola. If noise is present (as is the case in Figure 13), the advantage of estimating a lower-degree function can be even larger.

Another application of 2nd derivatives is to capture the interaction of parameters. Figure 10 shows the interaction between two linearly interdependent parameters in the Bipolar stage of the retina model, given a random checker-board flicker stimulus and a ground-truth model with default parameters. We show the gradient with respect to each parameter and a resulting flow field over a visualization of the error. A long corridor of almost identical solutions exists on the line $g_{leak} = \lambda_{bip}$. From this plot we can already assume that gradient descent will be very inefficient, but using a Hessian based descent method is more effective.

5.4 Optimizing the Input to a Model

We set up a small simulation in which we added two retina models (A and B) and took the difference of their outputs. The configuration between the models differed slightly. Then we created a Theano graph, which created a moving bar stimulus, dependent on a few parameters, such as direction and speed. We added this graph as an input to the model. When we now vary the speed of the gratings, the difference between the two models will change, for one, there is only a certain range where the moving gratings are actually visible for the model due to bandpass filtering (and the limitations of resolution). Figure 11 shows the optimal speed to discern the two models. The function is not a simple relation, and even shows more than one local maximum, possibly due to harmonics. We can compute the gradient of the difference with respect to the speed parameter of the gratings, which fits well with the actual error observed. If this were an actual experiment and we were trying to decide whether model A or B is closer to the actual mechanism we were observing, we could now choose the stimulus that will give us the greatest amount of information.

5.5 Comparison to other simulation software

As we have shown in section 5.1.1, the response of Convis is identical to the VirtualRetina if an identical configuration is used. The linear filtering and gain control properties are identical within numerical errors the magnitude of which is configurable. In addition to those response characteristics, Convis can also simulate non-circular, non-Gaussian and non-continuous receptive field shapes. Similarly to COREM,

Convis can be configured more flexibly, combining basic filters. However as the output of each sub graph is a Theano variable, simple alterations such as adding or multiplying scalars, taking the logarithm etc. can create new behavior without creating any new subgraphs for the model.

Table 1 shows a comparison between different retina simulation software.

5.5.1 Calculation Speed

Convolutions are remarkably slower than recursive filtering. This has to do with circular Gaussian spatial filtering being separable into x and y components, which can be efficiently computed in four separate phases while convolutions have to process $x_{image} \times y_{image} \times x_{filter} \times y_{filter}$ single multiplications. Including a time dimension into the filters which are not separable further increases the complexity drastically. We compared VirtualRetina and Convis on a Dell PC with a 6-core Intel(R) Xeon(R) CPU E5 v3 running at 2.40GHz, 32gb of RAM and a NVIDIA Quadro K620 GPU. While for the above simulations the calculation speeds happen to be 45 seconds for the VirtualRetina implementation and 159.2 s for our retina model (plus 9.8s compilation time), this comparison is not valid under all circumstances. For one, it depends on the hardware, specifically the graphics card, as the limitation in memory or number of cores can severely decrease or increase the efficiency of GPU calculations. In the same manner, a faster single thread CPU can speed up VirtualRetina. But even more crucial are the simulation parameters, namely the time step of the simulation and time constants of each filter. While the time step acts linearly on recursively implemented filters (time constants do not affect the speed of recursive processing at all), for a convolutional implementation, increasing the temporal resolution by a factor of two will not only double the length of the input sequence, but also the length of all filters and thus the number of multiplications performed on the GPU. For systems with low-end or no GPUs, the resolution can be decreased or, if it is acceptable that filters are constraint to exponentials and Gaussians, the recursive re-implementation of the filters can be used instead.

5.5.2 Features of VirtualRetina not implemented

We did not implement code to parse radially varying parameters. While the general idea of retinotopic varying parameters can be useful, we found that we were interested in rather local retinal populations ($< 2\,\mathrm{deg}$) which will be fairly uniform. Radially varying blur can still be used when filtering with recursive spatial filters, but the configuration values for radially varying blurs in VirtualRetina xml files will be ignored by default. Also, we did not implement cell sampling schemes. One pixel is one cell for us, however selecting a subset of cells from the output (before compiling the model) is possible and it can reduce the computations actually performed by the model due to the optimization mechanisms.

6 Discussion

Using the Convis toolbox we can simulate ganglion cell responses similar to VirtualRetina on an equally large scale (Section 5.1.1) while being able to switch between Gaussian receptive fields and arbitrary receptive field shapes, for instance obtained directly from experiments. Convis can use non-separable

convolution filters to implement motion sensitive filters in a very straightforward manner (Section 5.1.2). Large, linear convolution filters also have the advantage that their parameters, although numerous, are easy to fit. Convis models are implemented as abstract computational graphs. Through automatic differentiation, a gradient on the output can be defined with respect to a model parameter. This gradient can be used to fit a linear filter (section 5.2) or non-linear parameters (5.2.3). Even for complex, custom models, gradient expressions can be created easily and the model parameters can be explored to find redundant parameters or choose a suitable optimization algorithm (Section 5.3). The stimulus plays an important role when comparing a model and a physical system. To aid experimenters in choosing a good stimulus, we demonstrated that also the parameters of a stimulus can be optimized with gradient descent to maximize the difference between two very similar models and thus the information that new experiments will provide for the model fitting process (Section 5.4).

Conflict of Interest Statement

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Funding

This research was supported by ANR - Essilor SilverSight Chair ANR-14-CHIN-0001.

Acknowledgments

Thanks to Adrien Wohrer for helpful insights into the workings of VirtualRetina and thanks to Richard Carrillo for insightful comments.

References

Baden, T., Berens, P., Franke, K., Rosón, M. R., Bethge, M., and Euler, T. (2016). The functional diversity of retinal ganglion cells in the mouse. *Nature*, 529(7586):345–350.

Bednar, J. A. (2012). Building a mechanistic model of the development and function of the primary visual cortex. *Journal of Physiology-Paris*, 106(5):194–211.

Cessac, B. and Palacios, A. G. (2013). Spike train statistics from empirical facts to theory: The case of the retina. *Modeling in Computational Biology and Biomedicine: A Multidisciplinary Endeavor*, 9783642312:261–302.

Deriche, R. (1993). *Recursively implementating the Gaussian and its derivatives.* PhD thesis, INRIA.

Garvert, M. M. and Gollisch, T. (2013). Local and global contrast adaptation in retinal ganglion cells. *Neuron*, 77(5):915–928.

Heitman, A., Brackbill, N., Greschner, M., Sher, A., Litke, A. M., and Chichilnisky, E. (2016). Testing pseudo-linear models of responses to natural scenes in primate retina. *bioRxiv*, page 045336.

Korenberg, M. J. and Hunter, I. W. (1986). The identification of nonlinear biological systems: Lnl cascade models. *Biological cybernetics*, 55(2):125–134.

Lorach, H., Benosman, R., Marre, O., Ieng, S.-H., Sahel, J. A., and Picaud, S. (2012). Artificial retina: the multichannel processing of the mammalian retina achieved with a neuromorphic asynchronous light acquisition device. *Journal of Neural Engineering*, 9(6):066004.

Martens, J. (2010). Deep learning via Hessian-free optimization. *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 951:735–742.

Martínez-Álvarez, A., Olmedo-Payá, A., Cuenca-Asensi, S., Ferrández, J. M., and Fernández, E. (2013). RetinaStudio: A bioinspired framework to encode visual information. *Neurocomputing*, 114:45–53.

Martínez-Cañada, P., Morillas, C., Pino, B., Ros, E., and Pelayo, F. (2016). A Computational Framework for Realistic Retina Modeling. *International Journal of Neural Systems*, 26(7):1650030.

Masland, R. H. (2001). The fundamental plan of the retina. *Nature neuroscience*, 4(9):877–886.

Pérez, F. and Granger, B. E. (2007). Ipython: a system for interactive scientific computing. *Computing in Science & Engineering*, 9(3).

Real, E., Asari, H., Gollisch, T., and Meister, M. (2017). Neural circuit inference from function to structure. *Current Biology*, 27(2):189–198.

Sanes, J. R. and Masland, R. H. (2015). The types of retinal ganglion cells: current status and implications for neuronal classification. *Annual review of neuroscience*, 38:221–246.

Shapley, R. and Victor, J. (1978). The effect of contrast on the transfer properties of cat retinal ganglion cells. *The Journal of physiology*, 285:275.

Simoncelli, E. P. and Olshausen, B. A. (2001). Natural image statistics and neural representation. *Annual review of neuroscience*, 24(1):1193–1216.

Theano Development Team (2016). Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688.

van Rossum, M. C. (2001). A novel spike distance. *Neural computation*, 13(4):751–763.

Van Vliet, L. J., Young, I. T., and Verbeek, P. W. (1998). Recursive gaussian derivative filters. In *Pattern Recognition, 1998. Proceedings. Fourteenth International Conference on*, volume 1, pages 509–514. IEEE.

Wohrer, A. and Kornprobst, P. (2009). Virtual Retina: a biological retina model and simulator, with contrast gain control. *Journal of computational neuroscience*, 26(2):219–49.

Wohrer, A., Kornprobst, P., and Vi, T. (2007). Virtual Retina : a biological retina model and simulator, with contrast gain control [Research Report]. *RR-6243*, pages 32 ¡inria–00160716v2¿.

Tables

Table 1: Comparing different retina simulation software: VirtualRetina (Wohrer and Kornprobst, 2009), Topographica (Bednar, 2012), Virtual Retina++ (continued development of VR in the ENAS / PRANAS package Cessac and Palacios (2013)), Lorach et al. (Lorach et al., 2012), and RetinaStudio (Martínez-Álvarez et al., 2013) . ≈VR signifies that the gain control was implemented similarly to VirtualRetina: Contrast gain control through shunting inhibition, using a local estimate of spatio-temporal contrast (Wohrer et al., 2007).

Name	Application	Contrast Gain Control	Language of Source Code, Configuration	Open Source	Filters	continuous in/output	Optimization/ Plasticity
VirtualRetina	RGC responses	Local Shunting	C++, xml	Yes (CeCILL-C)	recursive	No	No
Topographica	Primary Visual Cortex / Neural Maps	Naka-Rushton	C, Python		convolution		Yes
ENAS	Model Verification	≈VR	C++, xml/GUI	No	recursive	No	No
Lorach et al.	Neuroprosthetics					No	No
COREM	RGC responses	≈VR	C++, scripts	Yes (CeCILL-C)	recursive	Yes	No
RetinaStudio	Neuroprosthetics		C#/Flowlang	Flowlang		No	No
Convis	RGC responses	≈VR	Python / theano	xml or script	recursive convolution	Yes	Yes

Figures

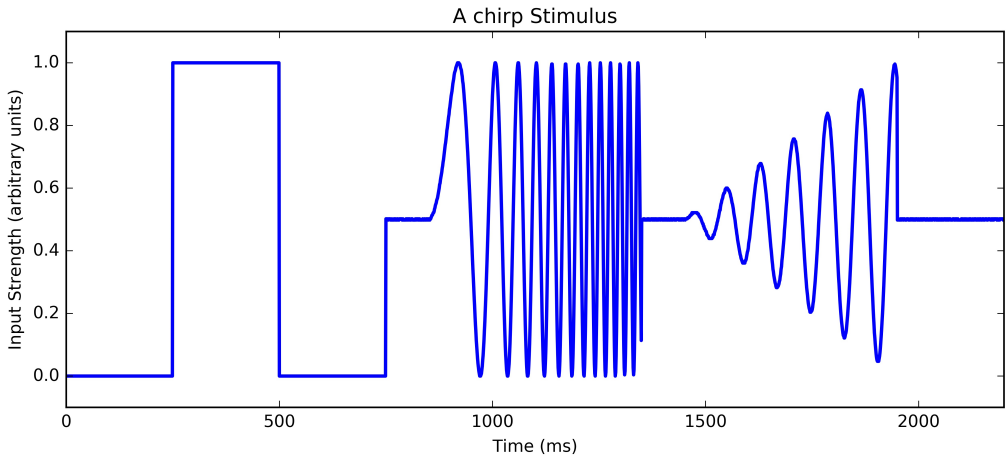

Figure 1: A "chirp" stimulus: the stimulus is comprised of an Off-On-Off pulse, then oscillations with increase in frequency, then oscillations with increase in amplitude.

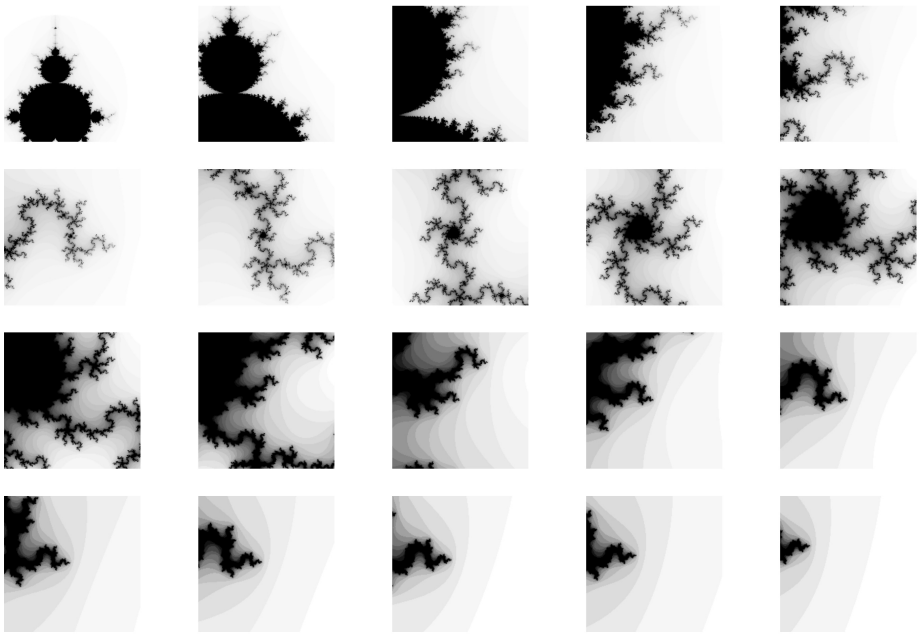

Figure 2: A zooming fractal stimulus: to compare a spatially complex stimulus, we generated a Mandelbrot fractal and created a continuous zooming motion.

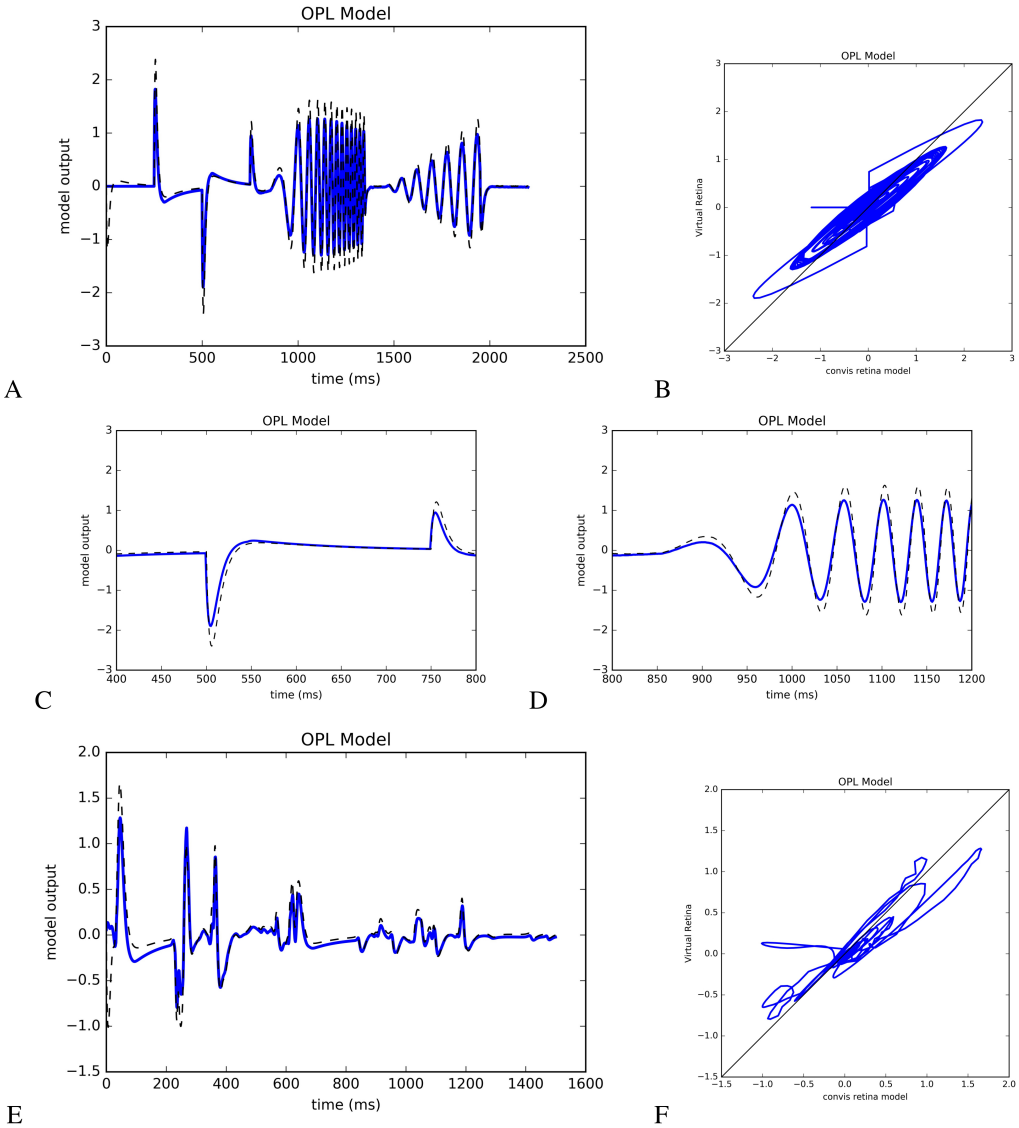

Figure 3: Comparison of Virtual Retina OPL stage and the convolutional retina model: The stimulus is a "chirp" (see figure 1) for A-D and a fractal (see figure 2) for E and F. Due to numerical differences between the method of recursive filtering and convolutional filtering, the full-convolution OPL stage differs slightly from the original trace for very abrupt changes.

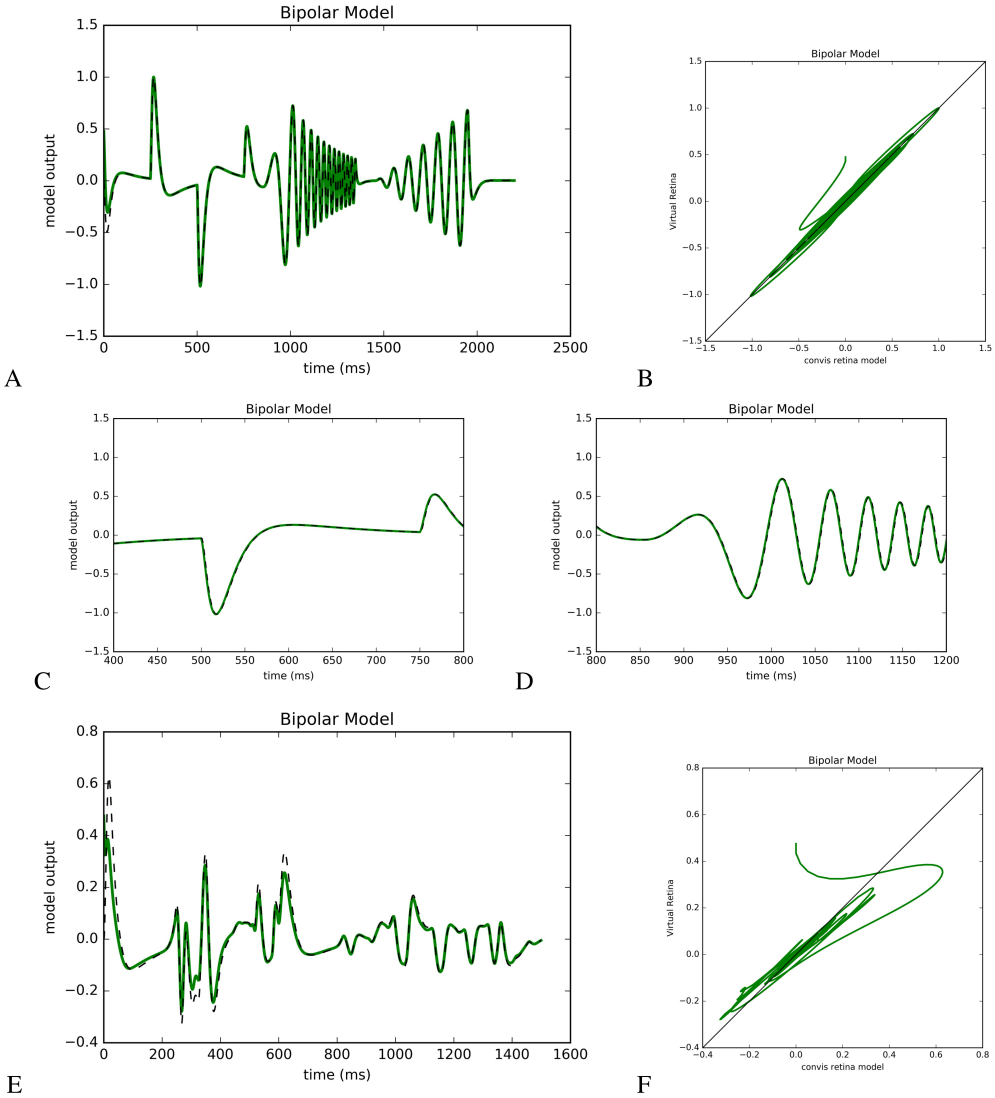

Figure 4: Comparison of Virtual Retina Bipolar stage and the convolutional retina model. The stimulus is a "chirp" (see figure 1) for A-D and a fractal (see figure 2) for E and F.

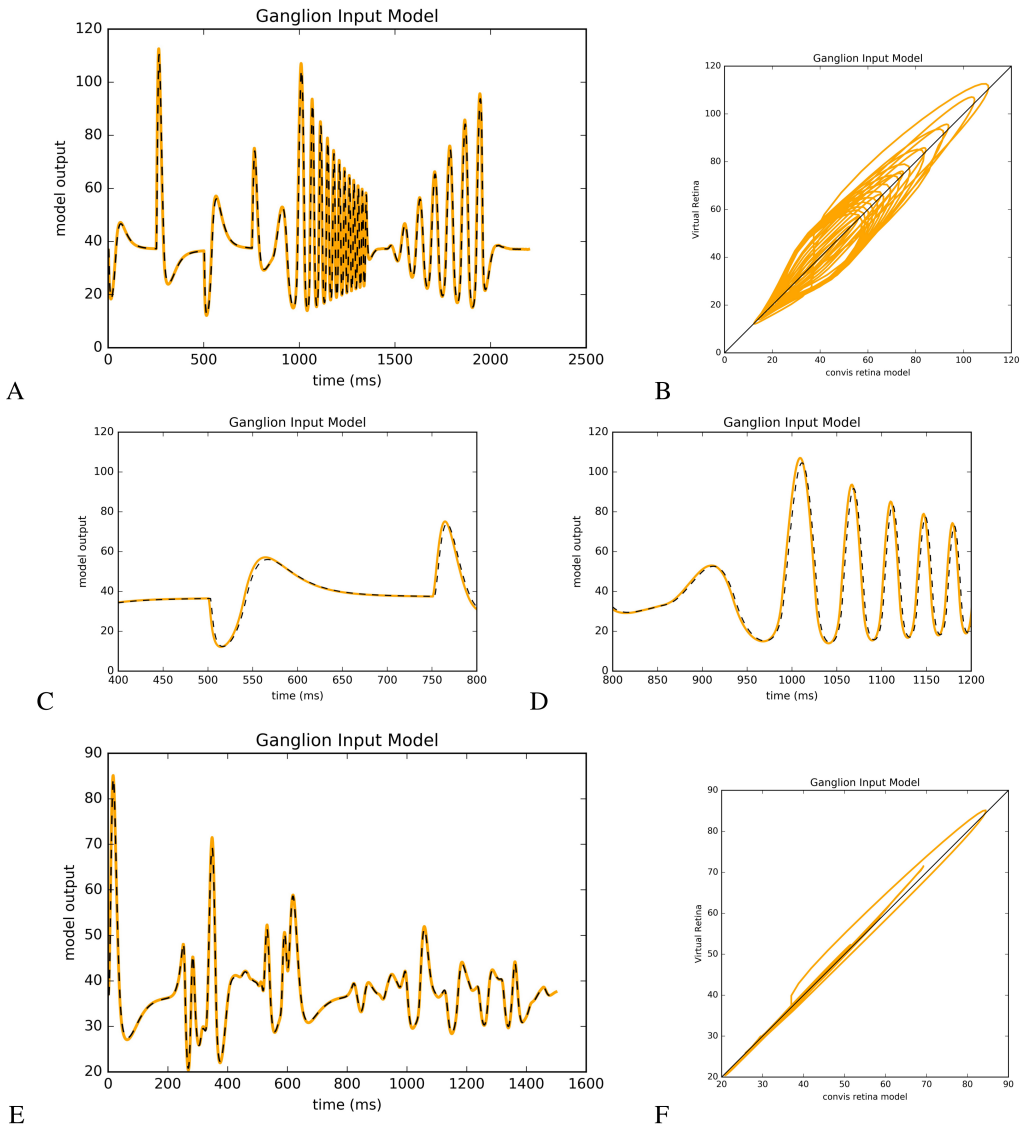

Figure 5: Comparison of Virtual Retina Ganglion Layer stage and the convolutional retina model. The stimulus is a "chirp" (see figure 1) for A-D and a fractal (see figure 2) for E and F.

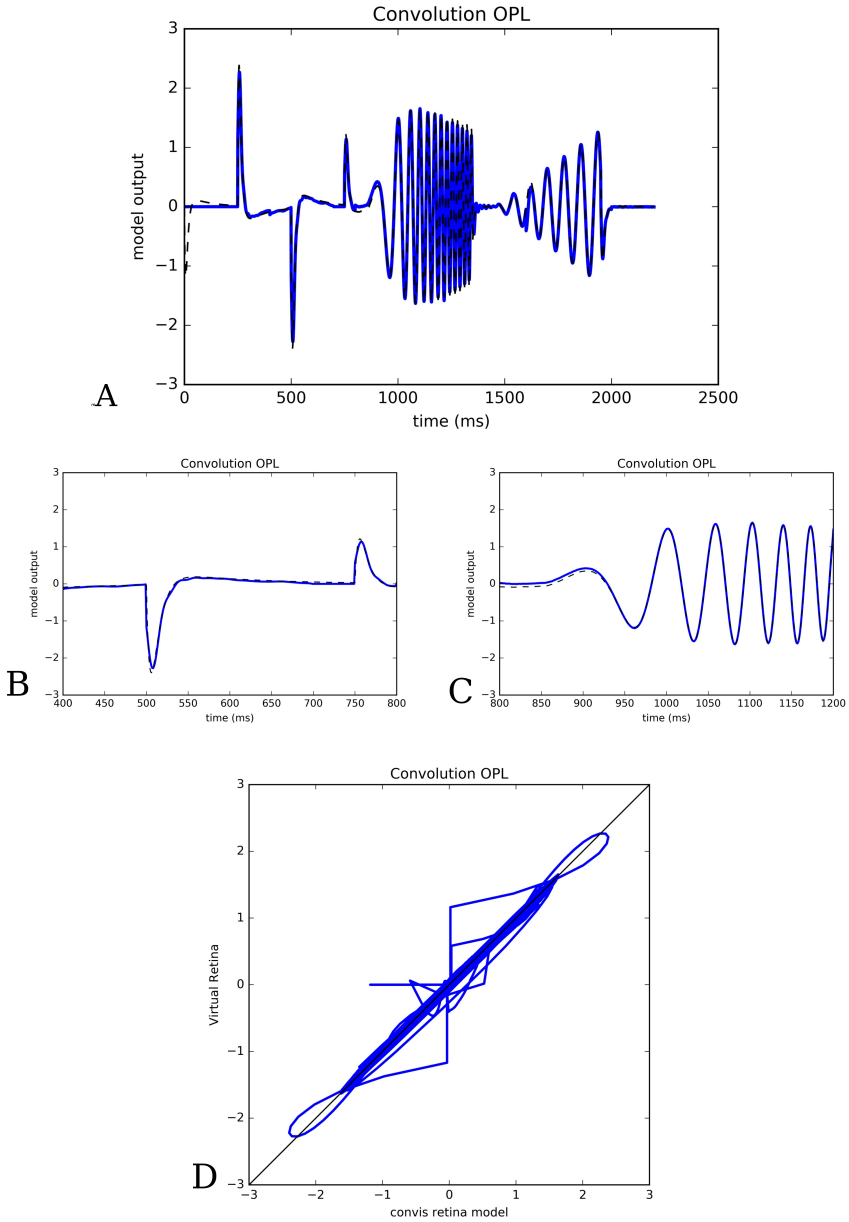

Figure 6: Comparison of Virtual Retina OPL stage and the convolutional retina model: The OPL stage fitted to the desired response instead of using configuration values can reproduce the response of the VirtualRetina OPl layer faithfully. (A) is an overview of the whole timeseries, (B) and (C) are details and (D) shows the relation of outputs of VirtualRetina and the convolution OPL as an x-y plot.

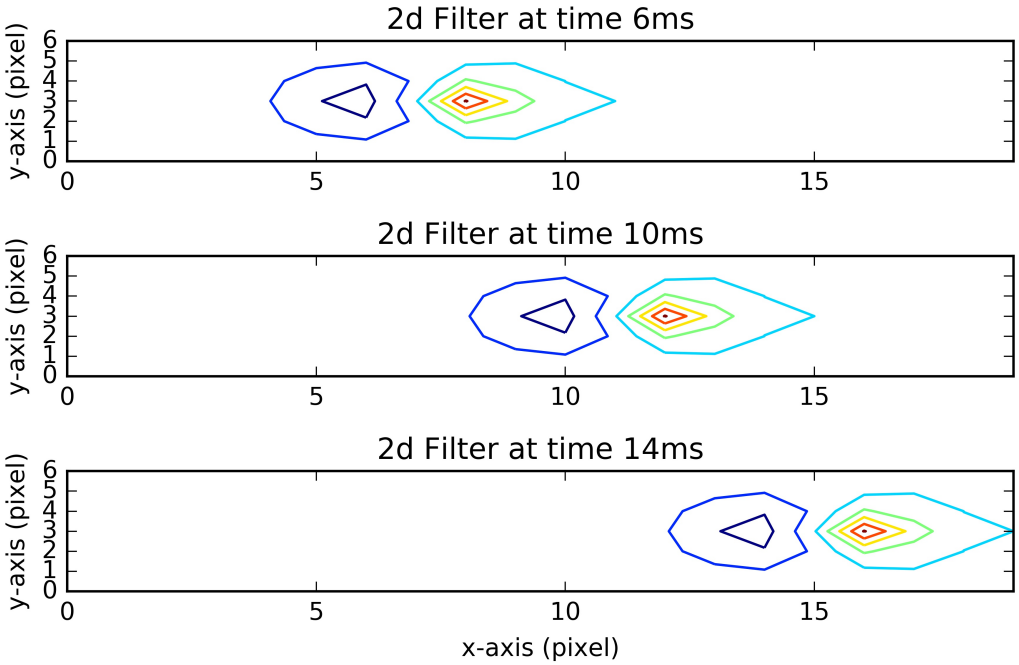

Figure 7: Direction Sensitive Receptive Field

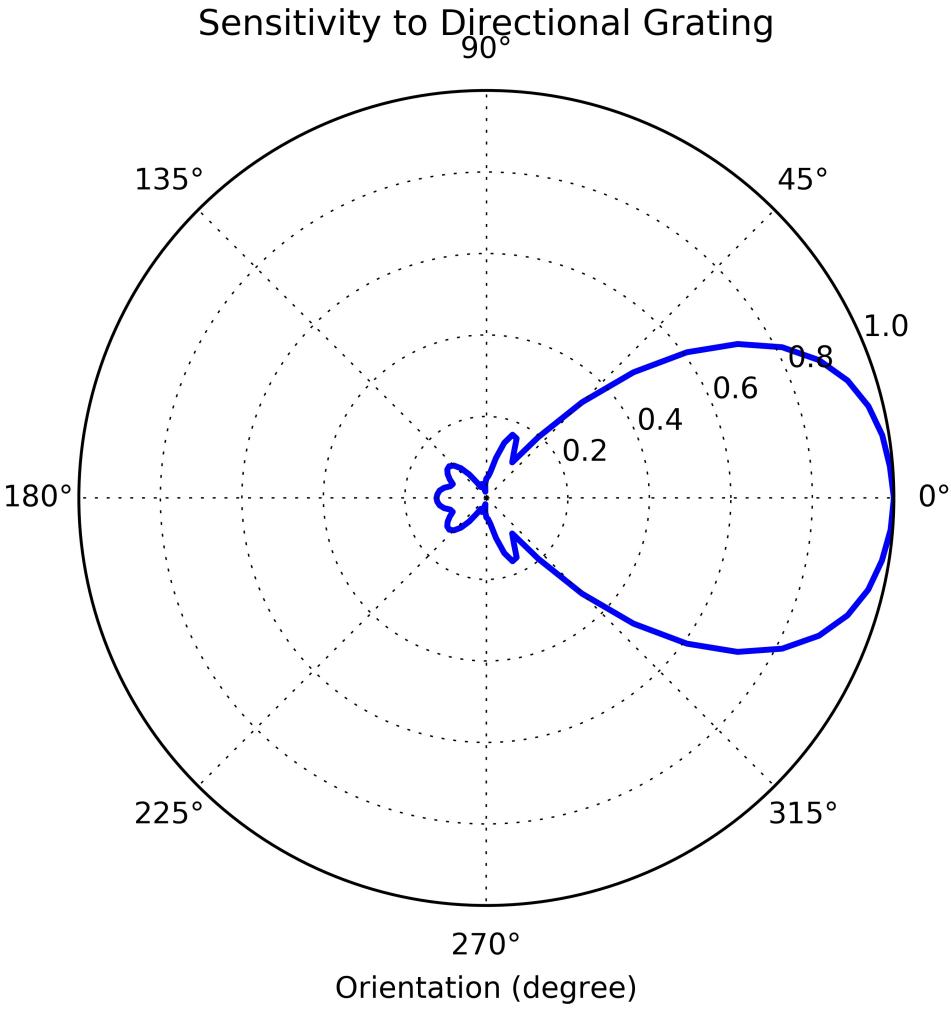

Figure 8: Direction Selectivity tuning of a DS cell

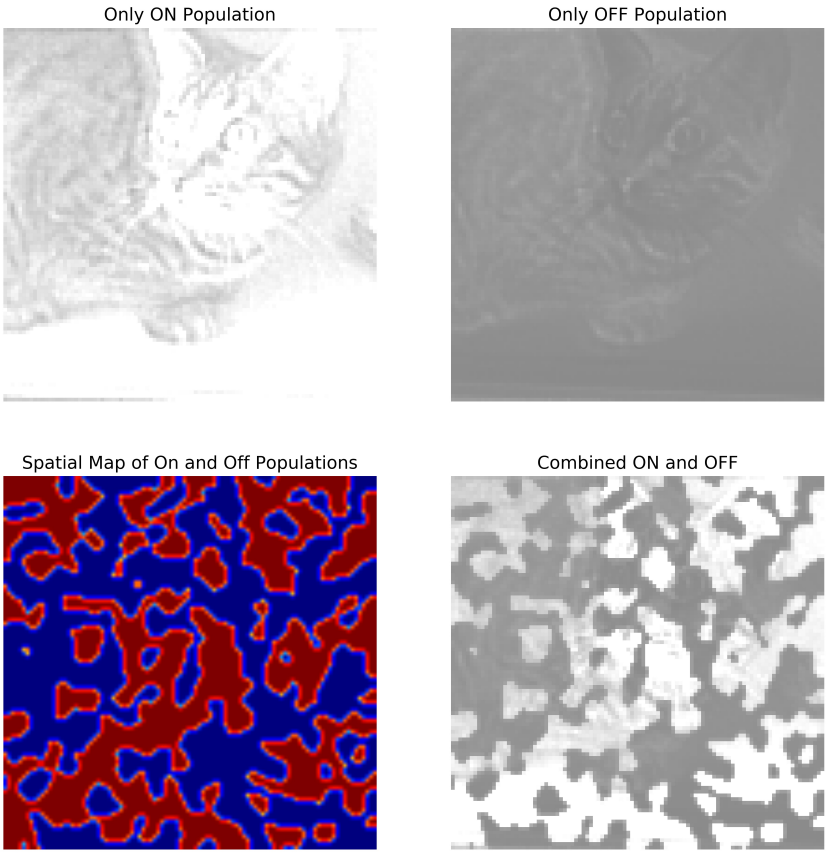

Figure 9: An example of simple modification of a model: The On and Off cell outputs of a VirtualRetina model are combined with a `SoftSelectFilter` to spatially integrate the two populations.

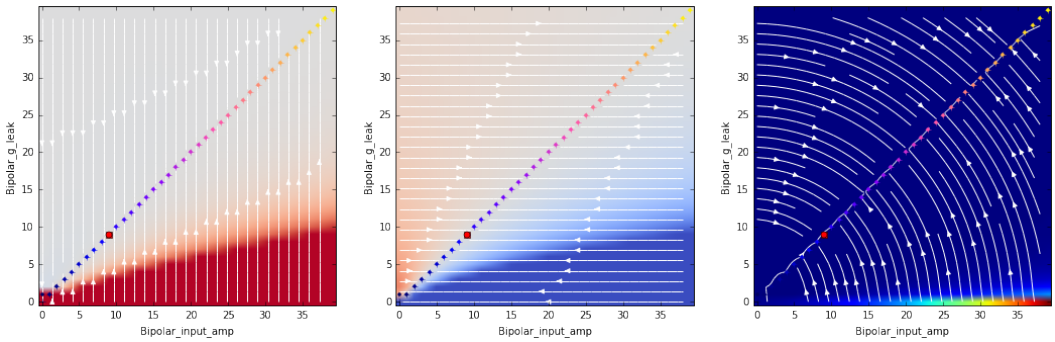

Figure 10: The interaction of two parameters in the retina model. We visualized a grid search for the true parameters of a ground-truth model in terms of the gradients with respect to each of the parameters and the error function with an overlay of the flow field resulting from the two gradient functions. Small dots show the minima along one parameter axis.

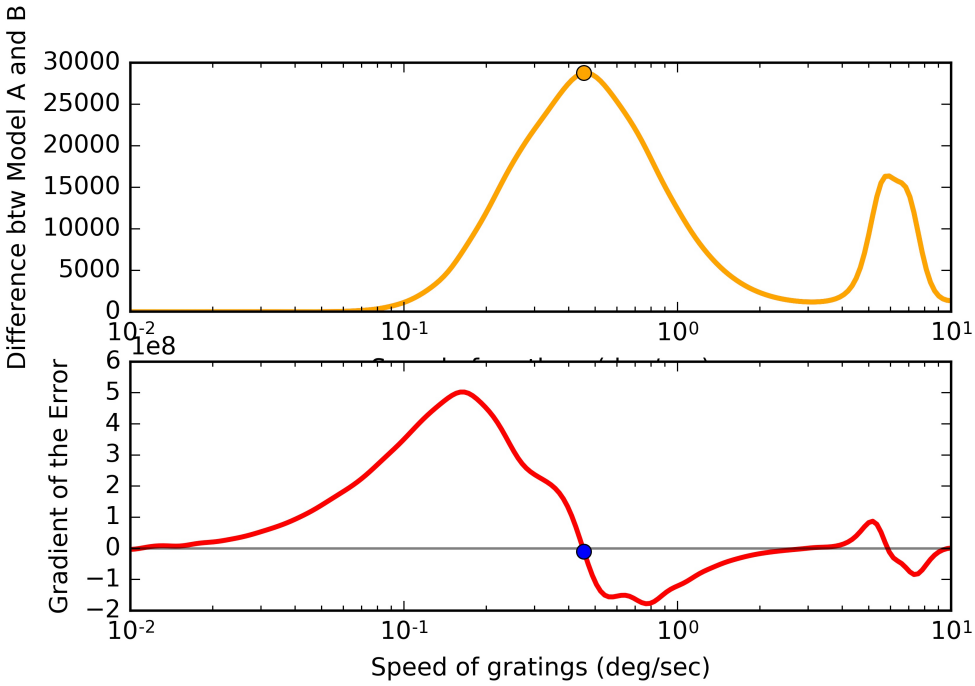

Figure 11: The difference between two models, slightly differing in their parameters, is low for very fast or very slow moving gratings, but in between there exists an optimum which is marked with a point. The same point is marked in the gradient of the error with respect to the grating speed parameter. There is also a second, local maximum.

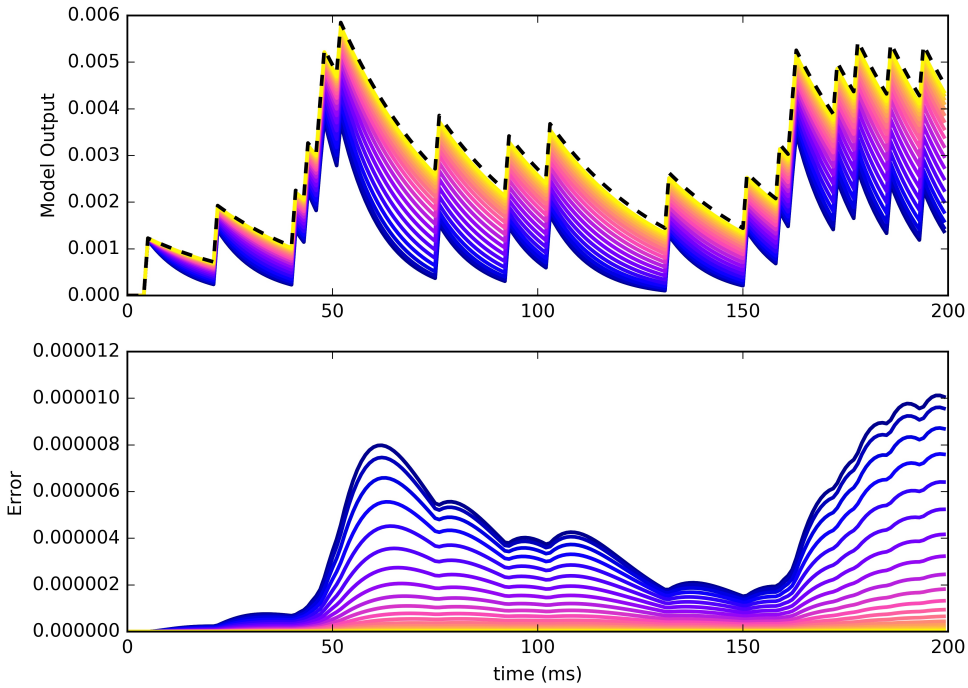

Figure 12: The trace of an exponential filter being fitted to a target: Top shows the different traces(with a very small adaptation between each iteration) from dark blue to yellow. The bottomplot shows the accumulated error for each trial.

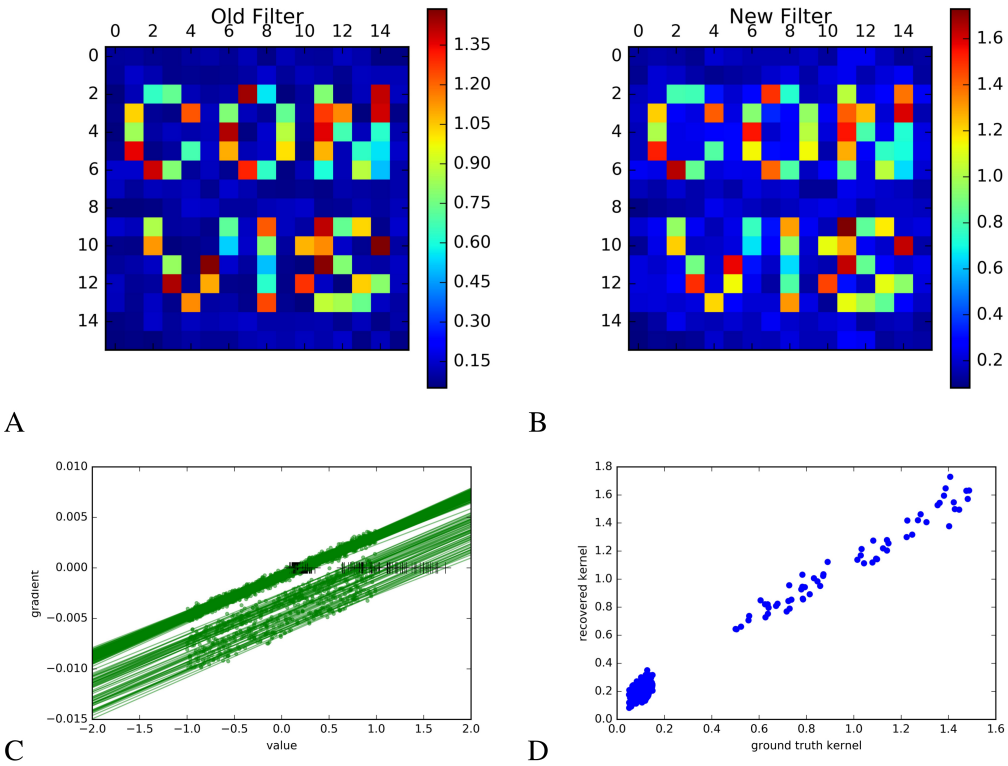

Figure 13: A receptive field recovered with gradient descent: Although the parameter space is large (16x16 pixel), the direction for each pixel can be estimated well, even in noise. (A) shows the original filter, a sparse text sample with multiplicative noise; (B) shows the recovered stimulus; (C) the gradient of each pixel, sampled over completely random kernels allows for a linear fit to find the zeros and thus the minimum of the error function (D) comparing old and new values the values were well recovered, up to a uniformly distributed noise source which was added on each trial. The code for this example can be examined in the additional material and in the example folder of the toolbox.

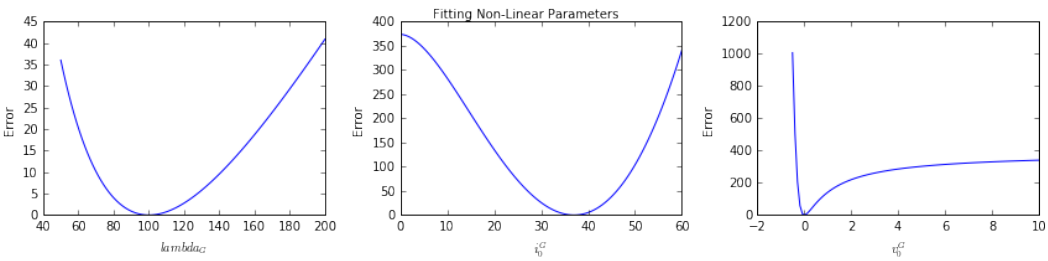

Figure 14: Nonlinear parameters do not show a squared error function. Still the shape is concave and the gradient points toward the minimum.