# Project Report: *Exploring the Diderot programming language and its applications to the visualization of neural models* ELEN E6001

Daniel Clark
djc2184

December 26, 2013

## Overview

This objective of this report is to explain the functionality of a domain-specific programming language called Diderot and discuss how it can be utilized in the realm of computational neuroscience. The report will give a thorough explanation of the idea behind Diderot, some examples of its applications, and its potential use for implementing high-level brain operators on continuous neural fields. A particular high-level operator, optical flow, will be discussed as well as some approaches for calculating it, as initial experiments into Diderot involved coding some of these approaches. Further detail will be given into the experiments performed with the software, a summary of what was learned from it, and its advantages and shortcomings with regard to the implementation of neural information processing algorithms.

An aspect of image visualization which Diderot excels at, line integral convolution or LIC, will be explained in detail - as this method was used to visualize the phase portraits of various neuron spiking models. This report will break down the idea behind LIC, its mathematical formulation, computational methods, and its applications. It will also discuss the basics of each neuron model, the behavior of the phase plane, and bifurcation theory.

Finally, a summary of experiments performed with visualizing the phase portraits via LIC in Diderot will be given. A breakdown of the Diderot code, MATLAB code, and miscellaneous importing/exporting utilities developed will be delivered. In addition, the appendix has a README file showing how the code pieces come together and how the files are organized so further experiments can be done.

## Contents

# The Diderot Language

This section will discuss the idea behind Diderot, how the language is designed and implemented, and some example programs and results. As this report focuses on the utility of Diderot in the context of computational neuroscience, the following subsections are good reference for discussion of the language throughout the report.

## Background

Diderot is primarily designed for the visualization and analysis of biomedical image data with the theory that scientists and researchers should be able to implement sophisticated image processing algorithms on a high level, that is, without worrying about the numerical methods and workarounds that come with typical software implementation. In addition, due to the ever-increasing sophistication of biomedical signal acquisition technologies, which are capturing more data at higher resolutions than ever, there is a tremendous need to process these images in parallel for optimized performance. As such, the language's core model is that of a portable and parallel framework, which allows for running on multiple threads. This emphasis on parallelism realizes Diderot as an optimal medium for running a large number of mostly independent computations for image analysis and visualization [1].

In order to allow for a high-level interaction with the image data, Diderot is optimized for operating on *continuous tensor fields* [2]. A continuous tensor field is a $d$-dimensional field of tensor values (i.e. scalar, vector, or matrix values at each point in the field) which, ideally, has infinitely small displacement between its samples. Utilizing this entity as the primary type of object for numerical manipulation in code allows for Diderot to be used in across all sorts of applications, as this continuous field can be used to describe any number of complex things (e.g. blood flow, neural circuits).

Since Diderot's primary domain is that of real-world image data (such as an MRI or CT scan), the assumption is made that the discrete inputs for processing are sampled from some continuous real-world space. Diderot approaches defining this data in a continuous realm via the fundamentals of signal processing, where, the underlying continuous signal can be reconstructed via convolving discrete samples with a continuous reconstruction kernel, $h(x)$. For higher-dimensional signals, the reconstruction kernel can be expressed as a product of separable one-dimensional kernels, i.e. $H(x, y, z) = h(x)h(y)h(z)$. The primary role of these kernels in Diderot is to provide for a realistic interpolation of the sampled data and to permit continuous operations (i.e. partial derivatives) on the interpolated signals.

With this framework, researchers need the flexibility to specify the differentiability of data as well as to utilize fast and accurate reconstruction kernels to ensure "real-world" data fidelity from discrete samples. Diderot ensures this. Part of the motivation the authors had for creating Diderot came from the fact that in order to guarantee fast runtimes, the computational cost must be optimized for the specific algorithm being implemented. This is held in contrast to canonical image processing tools like MATLAB or Python, which distribute computations uniformly over over the entire image data [2]. Diderot has an advantage over these array processing utilities in its ability to optimize access and computing time for irregular and sparse data as well. While this sacrifices a certain degree of generality (Diderot is indeed a domain-specific language or DSL), it offers the simplicity and robustness necessary for manipulating continuous tensor image data.

## Language design and implementation

The continuous tensor field is the fundamental element which makes Diderot a novel software package. As such, the tensor is implemented as a basic data type in the language. In addition to tensors, booleans, integers, and strings make up the core values for use. The keyword $\texttt{tensor}[\sigma]$ to declare a tensor element, with $\sigma$ specifying the order of the tensor - 0-order corresponds to a scalar element, 1-order to a vector element, 2-order a matrix. For example

$$\begin{aligned} \texttt{tensor}[\,] &\quad - \text{ creates a scalar} \\ \texttt{tensor}[3] &\quad - \text{ creates a } 3{\times}1 \text{ vector} \\ \texttt{tensor}[2,2] &\quad - \text{ creates a } 2{\times}2 \text{ matrix.} \end{aligned} \tag{1}$$

Synonms are implemented for commonly used tensors, such as `vec3` for 3D vectors or `real` for scalars. Three abstract data types are implemented to assist with the language's high-level format:

$$\begin{aligned} \texttt{image}(d)[\sigma] \quad &- \text{ (input) image data of } d \text{ dimension with } \sigma\text{-shaped tensor values} \\ \texttt{kernel}\#k \quad &- \text{ convolution kernel that is } k \text{ times differentiable} \\ \texttt{field}\#k(d)[\sigma] \quad &- \text{ continuous tensor field that is } k \text{ times differentiable,} \\ &\quad \text{of dimension } d \text{ and having } \sigma\text{-order tensor values} \end{aligned} \tag{2}$$

The `image` type stores the input to the Diderot program, such that Diderot is aware of how many dimensions the image is, and what type of tensor data it holds. `image` can be used to store a standard 2D frame of scalar values, or something more sophisticated, like a 3D image of 3×1 velocity vectors, or a time-evolving 3D image (4D data) of 2×2 Hessian matrices.

The `kernel` type instantiates an underlying "continuous" kernel used to perform interpolation of input data, as well as to implicitly calculate higher-order operations on continuous fields (i.e. gradient). There are a number of built-in kernels for use, including `tent` for non-differntiable, linear interpolated data, `ctmr` for once-differentiable Catmull-Rom spline interpolation, and `bspln3` for the twice-differentiable uniform cubic B-spline basis function. The use of these various kernels are up to the discretion of the researcher to utilize the appropriate basis for the formation of the continuous data domain. The amount of differentiability that a kernel allows for is important, as it limits the amount of gradient operations that can be performed on the resulting field. If a field $F_1$ is defined to be $k = 1$ times differentiable, and a gradient $\nabla$ operator is performed on it, then the resulting field $F_2$ will be $k = 0$ times differentiable, and using the $\nabla$ operator on $F_2$ will yield a compile-time error.

By convolving a `kernel` with an `image` (directly in code via the $\circledast$ character), one creates a continuous `field` of their input data. This field object is compatible with high-level operators like $\nabla$ (for fields of scalar values) and $\nabla\otimes$ (for fields of vector and matrix values). Then when one wants to access the modified field values, they can *probe* the field at a location `pos`, via something like: `fieldVal = F(pos)` and store the values of interest. Additionally, Diderot also supports adding and subtracting, as well as scaling of entire fields at once.

As Diderot is optimized for parallel programming, its code structure reflects an independent, multithread approach. A Diderot program is broken into three primary sections: global definitions, strand updates, and initialization.

The global definitions section is where the user declares immutable variables to use as constants throughout the program. This is also the section where an input image is typically loaded in and its continuous field is created. This can be seen in the "Global defs" section in the sample code shown on the next page. Along with various constants, the user can specify the granularity of how many samples they'd like to extract from the continuous field (regardless of the resolution of the input).

The strand updates section is analogous to a kernel function in CUDA or OpenCL as this part of the code does all of the central math of the program. The strand in code on the following page is called DEMO and it iterates via indexing parameters `xi` and `yi`. In this case, these parameters are used to probe the field at different positions. An `output` variable is declared, `sum`, which serves as the data that is written to the disk when the program is ran. The `update` method within the strand updates `sum` until a condition is met where the `stabilize` method is called. Once `stabilize` is invoked, the output strand finishes running on that particular index of `xi` and `yi` and the program outputs the results to a text file. Strands can also be terminated via the `die` keyword, in which case, no output is written to disk. Notice the gradient operator on field `V`. We create `V` as a continuous scalar (0-order tensor) field. However, when we probe the gradient of `V`, the output is a 2-dimensional vector (as there are two space dimensions of the input image from which to compute the derivative with respect to). Also in this process, the differentiability of $\nabla$`V` goes down by one count (from 1 to 0), which would prevent any more gradient operations from operating on that field (as it is no longer continuously differentiable).

The initialization section is pretty straightforward as it declares what strand is to be run and how the strand will be iterated through. The code below shows the `initially` method calling our strand name with parameters `xi` and `yi`, which will iterate over the range 0..`imgSizeX-1` and 0..`imgSizeY-1`, respectfully, with `xi` as the inner loop and `yi` as the outer. It should be noted here that the loop order follows a bulk-synchronous parallelism model [3], where, program execution happens in *super-steps*. The super-step here is the update method being called once over every strand dictated by the loop range so that every parallel strand is updated *synchronously*; `update` is then invoked again and again, over all strands, until the `stabilize` or `die` methods are called. This model allows for optimal performance and portability to parallel processing implementations. In the case of a serial execution, the super-step acts as an inner-loop (first `strand DEMO` runs per `xi, yi` once) and the update method as the outer loop (now update each strand one-by-one another time...etc).

```
// ----- Global defs -----
int imgSizeX = 300;     // how many x pts user wants from field
int imgSizeY = 200;     // how many y pts user wants from field
int stepNum = 25;       // step limit before writing to output
image(2)[] img = load("../data/einstein.nrrd"); // import image
field#1(2)[] F = img     ctmr; // convolve img with kernel = field

// ----- Strand section -----
strand DEMO (int xi, int yi) {
    real xx = lerp(0.0, 3.0, -0.5, real(xi), real(imgSizeX)-0.5);
    real yy = lerp(0.0, 2.0, -0.5, real(yi), real(imgSizeY)-0.5);
    vec2 pos0 = [xx,yy];
    output real sum = F(pos0);
    int step = 0;
    // --- Update thread with math under condtions ---
    update {
        // Do some fancy math
        vec2 grad =    F (pos);     // take gradient of field and probe it
        sum = grad[0]+grad[1];
        step += 1;
        if (step == stepNum) {
          stabilize;            // write "sum" to output
        }
    }
}

// ----- Initialization section -----
initially [ DEMO(xi, yi) | yi in 0..(imgSizeY-1), xi in 0..(imgSizeX-1) ];
```

It is important to get a sense of how Diderot manages the "continuous" fields under the hood. The fields are abstract data types, so the entire "continuous" field is not necessarily known or calculated when a field operation is performed (i.e. convolution, gradient). In fact, the underlying values are only computed when the field is *probed* at a particular location. This is best understood by looking at an example equation which shows the underlying calculations for computing the gradient of a field at point $\boldsymbol{x} = [x, y]^\top$.

$$\begin{aligned}
\nabla F(x) &= (V \circledast \nabla h)(\boldsymbol{x}) \\
&= \left( V \circledast \begin{bmatrix} \frac{\partial}{\partial x} h \\ \frac{\partial}{\partial y} h \end{bmatrix} \right) \\
&= \begin{bmatrix} \sum_{i=1-s}^{s} \sum_{j=1-s}^{s} V[\boldsymbol{n} + \langle i, j \rangle]\ h'(\boldsymbol{f}_x - i) h(\boldsymbol{f}_y - j) \\ \sum_{i=1-s}^{s} \sum_{j=1-s}^{s} V[\boldsymbol{n} + \langle i, j \rangle]\ h(\boldsymbol{f}_x - i) h'(\boldsymbol{f}_y - j) \end{bmatrix},
\end{aligned} \tag{3}$$

where $V$ is a discrete (scalar-valued) input, $h$ is a (separable) continuous kernel, $\boldsymbol{x}$ is the field position index, $\boldsymbol{M}^{-1}$ is the space mapping matrix, $\boldsymbol{n} = \lfloor \boldsymbol{M}^{-1}\boldsymbol{x} \rfloor$ is the discrete-mapped point, $\boldsymbol{f} = \boldsymbol{M}^{-1}\boldsymbol{x} - \boldsymbol{n}$ is the continuous point shifted by the discrete-mapped point. It can be seen here that the field gradient is computed only around a single point in the field space $\boldsymbol{x}$. The equation above implements differentiation with respect to each axis via a convolution of separable kernel $H$ over its support $s$. Its components $h(x)h(y)$ form the total spatial kernel, and the derivative with respect to each direction is found by differentiating that component of $H$ and performing a convolution over the image. It is important to note that there is a mapping between the *image space* and the *world space*, defined by matrix $\boldsymbol{M}$, see Fig. 1.

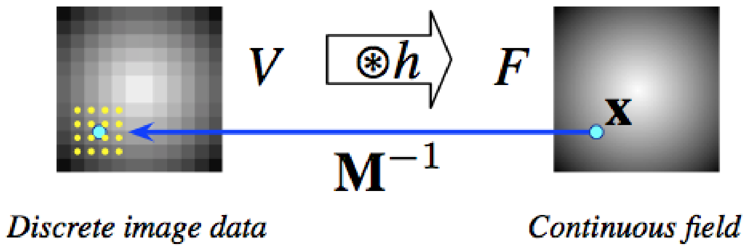


Figure 1: Mapping from continuous “world space” to discrete “image space”

This mapping is usually defined in metadata of the input file, where the displacement between points of the digital data corresponds to a *world space* displacement of a certain amount. This is important, as the continuous field in Diderot is defined in terms of world space - so when convolution is performed on the discrete image, the real-world index $\boldsymbol{x}$ must be mapped to *image space*, calculated over the support $s$ using locally interpolated image values, then re-mapped *back* to world space to give the correct value for the continuous field at that point. It should be noted that the kernel support $s$ cannot be modified or updated from within a Diderot program at this stage of development. This sounds like a lot of inherent computation to be performed at any one step, but since Diderot's strand section is essentially an independent kernel, the field calculations can be performed (speedily) in parallel across the entire image.

## Examples

The software package comes with a handful of data files and example code to perform various types of image analysis and visualization. The authors make clear in [2] to distinguish these two terms: *image analysis* is the process of extracting quantitative descriptions of the image makeup in order to characterize specific properties about the underlying real-world object. *Image visualization* is using tools from computer graphics to qualitatively depict patterns taken from the image.

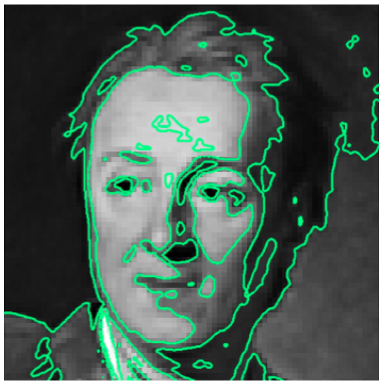
An example of image analysis that Diderot implements is calculating the isocontours of a grayscale image. The program appraoches this via a particle method, where the contour features are extracted via a two-dimensional grid of initial “particles” that traverse over the image via a gradient descent optimization method (Newton-Raphson). The particles produce an output or `die` out depending on their ability to `stabilize` in the program. Because this particle method is inherently parallel, it fits easily into Diderot's bulk-syncrhonous framework. This results of this can be seen in Fig. 2a

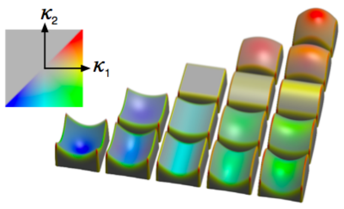
An image visualization method Diderot elegantly demonstrates is visualizing surface curvatures. This specific example maps a colormap onto the various object surfaces present in the image based on the magnitude of their curvatures. To demonstrate the readibility of Diderot code in relation to the equations implemented, consider the following calculations for curvatures $k_x$, $k_y$:

$$
\begin{array}{rl}
\textit{Equation}: & \textit{Code}: \\
H = \nabla F \otimes \nabla F & \texttt{tensor[3,3] H = } \nabla \otimes \nabla\texttt{F(pos);} \\
\boldsymbol{n} = ||\nabla F|| & \texttt{vec3 norm = normalize(}\nabla\texttt{F);} \\
P = I - \boldsymbol{n} \otimes \boldsymbol{n}^{\top} & \texttt{tensor[3,3] P = identity[3] - norm } \otimes \texttt{ norm;} \\
G = -\dfrac{PHP}{|\nabla F|} & \texttt{tensor[3,3] G = -(P}\bullet\texttt{H}\bullet\texttt{P)/|}\nabla\texttt{F|;} \\
d = \sqrt{2|G|^2 - \mathrm{tr}(G)^2} & \texttt{real d = sqrt(2.0*|G|\^{}2-trace(G)\^{}2);} \\
k_x = \dfrac{\mathrm{tr}(G) + d}{2} & \texttt{real kx = (trace(G) + d)/2.0;} \\
k_y = \dfrac{\mathrm{tr}(G) - d}{2} & \texttt{real ky = (trace(G) - d)/2.0;}
\end{array}
\tag{4}
$$

The equations behind curvature calculations and the program that calculates this on real images look almost identical, with a line-for-line match. This concept is used to produce the colored surfaces shown in Fig. 2b. This sample code served as a nice starting template for the experiments that were carried out in later sections.

It should be noted that any Diderot program, including the examples, can be run serially or in parallel. The backend of the compiler can be told to compile code for: sequential C code, parallel C code, and OpenCL. Plans for a CUDA backend are in the works, however this functionality has yet to be seen. The expectation is that, especially for very large datasets, that parallel hardware support (such as a GPU cluster) will accelerate runtime results dramatically.


(a) Isocontour analysis



(b) Curvature color-mapping visualization

Figure 2: Diderot analysis and visualization examples

## High-level brain operators

The primary utility of Diderot is that it abstracts away the tedious computations needed to perform math on continuous tensor fields. It allows for a manipulation of lots of data from a high-level, including the ability to use continuous derivatives as entire entities in programmable equations. With regard to computational neuroscience, it is advantageous to take the same approach with regard to neural circuits. If we can implement biologically-plausible algorithms on the neuron level, interconnect these entities as neural circuits, and achieve some benefit with regard to neural information processing (i.e. encoding of relative motion in a visual stimulus), then there should be a way to abstract away all of the low-level engineering and define a high-level "brain operator." In other words, is it possible to define an operator, call it $\beta$, that performs sophisticated encoding on a neural circuit in the way that a gradient operator, like $\nabla$ used in Diderot, performs a transformation of data on a continuous tensor field? What kinds of operators are possible and how can neural circuitry be represented in the form of a continuous tensor field? The following sections will address these questions and analyze our initial experiments with Diderot.

## Continuous neural fields

The amount of connections and synapses between neurons in any particular area of the brain is gigantic. Therefore, there is a need to represent the dynamics of significant layers of neurons in a convenient way, on a macro level. One way to do this is to use the already maturely-developed area of continuous dynamical systems to approximate the behavior of neural networks. This implies that continuous mathematical equations are able to model neural activity as quantities in a continuous field, with functional relationships to sources and sinks *of that field* [4]. This allows for an explicit form for the activity at any particular point in the field as a weighted (non-linear) function of neighboring points. The basic idea comes from the assumption that the average activity of a neuron (i.e. average firing rate) in the field at point $x$ at time $t$ can be represented by

$$Z_i(x,t) = f_i\Big(u_i(x,t)\Big), \tag{5}$$

where $Z_i(x,t)$ is the activity of neuron at point $x$, at time $t$, in layer $i$ of some neural network. $u_i(x,t)$ is the average membrane potential and $f_i$ is a non-linear activation function assumed to be monotonically non-decreasing [5]. In addition, this model assumes a time-varying, two-dimensional weighting function $w_{ij}(x,y;t)$ (shown in Fig. 3) to quantify the conduction time, synaptic delay, and inter-neuron connection strength; in other words, the amount of influence activity from neuron at point $y$ in layer $j$ has on neuron at point $x$ in layer $i$, $t$ time units after firing initiates from neuron $y$. A majority of the literature models this weighting function to be of a lateral-inhibition type - where any neuron has strong, excitatory connections local to its proximity, and dominantly inhibitory influence on neurons farther away - and this model has been shown to yield realistic, biologically-inspired topographic maps of stimulus encoders in the visual system [6] [7]. In addition to this weighting function, the field model must also incorporate any intensity change due to an external stimulus, $s_i$ input as well as the steady-state resting membrane potential, $r_i$, such that

$$s_i(x,t) = \bar{s}_i + \Delta s_i(x,t) \tag{6}$$

$$r_i = \bar{s}_i - h_i, \tag{7}$$

where $\bar{s}_i$ is the average stimulus intensity in layer $i$, $\Delta s_i(x,t)$ is the change in stimulus at point $x, t$ and $h_i$ is the difference between the contribution of potential due to the stimulus $\bar{s}_i$ and the resting potential $r_i$ in layer $i$. $h_i$ can also be thought of as where $u_i$ will converge (given no stimulus fluctuations) after a certain duration of time, $\tau_i$. These values can be generalized at the layer-level as the same type of neurons are arranged on their respective layer, as seen in Fig. 3 [5].

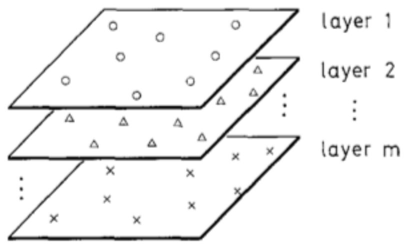


**Fig. 1.** Neural field

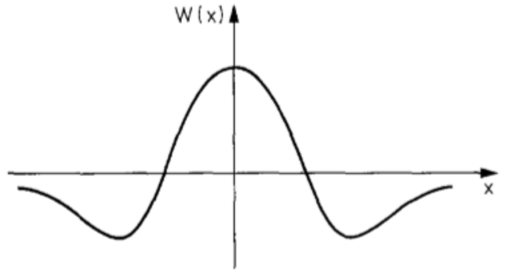


**Fig. 2.** Weighting function $w(x)$ of a lateral-inhibition type neural field

Figure 3: Continuous neural field layers and weighting function

The above assumptions can be combined to form the field equation [5]

$$\tau_i \frac{\partial u_i(x,t)}{\partial t} = h_i + \Delta s_i(x,t) - u_i(x,t) + \sum_{j=1}^{m} \int_{\Omega_j} \int_{v} w_{ij}(x,y;t-v) Z_j(y,v)\ dv\ dy, \tag{8}$$

where the last term is a temporal convolution of the time-varying weighting function with the neural activity at $y, v$, which is then summed over the entire neighborhood $\Omega_j$ of each layer, and summed over all the layers in the network. This term is combined with any input stimulus deviation $\Delta s_i(x,t)$ and distance to resting potential $h_i$; subtracting the present membrane potential $u_i(x,t)$ finally satisfies the differential equation (scaled by decay time constant $\tau_i$).

If a neural circuit is modeled as a network of these continuous fields, we should be able to define operators which act on the dynamic expressed in Eq. (8) in such a way that quantitatively expresses how the actual brain encodes and interprets elements of an input stimulus. This is a powerful sentiment, but one which is hard to start exploring without a specific objective in mind. The next section will outline an element of visual stimulus recognition to put our objective into context.

## Optical flow

Optical flow is generally thought of as the quantification of relative motion of a vision sequence. It is an example of a phenomena that is indirectly detectable from the stimulus - in other words, motion "flow" itself is not actually present in the frame-by-frame light intensities that make up a visual field. The perception of optical flow is realized through some sort of operation on the stimulus. This section will first present a basic optical flow algorithm to get a general idea of the kind of computations needed in software, then a biologically-inspired approach is explored that holds a possibility for how the brain encodes visual motion. In general, the following techniques approach the problem in three stages: pre-filter input data via smoothing process, extract relevant features, and weight and integrate features to produce a two-dimensional field of flow. The methods for features extraction are the novel part of the algorithms which will be focused on.

The canonical approach for detecting optical flow is by computing velocity from spatiotemporal derivatives of image intesity - this is known as the *differential approach* [8], where the image intensity at a point $\boldsymbol{x}$ at time $t$ is defined via velocity $\boldsymbol{v}$ as

$$I(\boldsymbol{x},t) = I(\boldsymbol{x}-\boldsymbol{v}t,0), \tag{9}$$

where $\boldsymbol{x} = [x,y]^\top$ and $\boldsymbol{v} = [u,v]^\top$ represent two-dimensional quantities for position and velocity, respectively, in each coordinate axes. The reasonable assumption is made that intensities are conserved, in other words, the image intensity along its direction of spatiotemporal motion has no change from one time instant to the next. Formally,

$$\frac{dI(\boldsymbol{x},t)}{dt} = 0 \ \Rightarrow \nabla_{\boldsymbol{x}} I(\boldsymbol{x},t)\cdot\boldsymbol{v} + \frac{\partial I(\boldsymbol{x},t)}{\partial t} = 0. \tag{10}$$

Eq. (10) is known as the gradient constraint equation, where

$$\nabla_{\boldsymbol{x}} I(\boldsymbol{x},t) = \begin{bmatrix} \frac{\partial I(\boldsymbol{x},t)}{\partial x} \\ \frac{\partial I(\boldsymbol{x},t)}{\partial y} \end{bmatrix}. \tag{11}$$

Horn and Schunck approached detecting optical flow utilizing this method and posing it as an optimization problem [9]. They solved for their algorithm form by minimizing Eq. (10) with an additional smoothness term to ensure a certain level of continuity of flow, where

$$\int_\Omega (\nabla I \cdot \boldsymbol{v} + \frac{\partial I}{\partial t})^2 + \lambda^2(||\nabla u||_2^2 + ||\nabla v||_2^2) d\boldsymbol{x} \tag{12}$$

is performed over an image domain of $\Omega$, and $\lambda$ is a tuneable parameter, which controls the influence of the smoothness constraint on the output. An iterative solution was arrived at, which enables numerical algorithms to compute for

$$u^{k+1} = \bar{u}^k - \frac{I_x\Big(I_x\bar{u}^k + I_y\bar{v}^k + I_t\Big)}{\alpha^2 + I_x^2 + I_y^2} \tag{13}$$

$$v^{k+1} = \bar{v}^k - \frac{I_y\Big(I_x\bar{u}^k + I_y\bar{v}^k + I_t\Big)}{\alpha^2 + I_x^2 + I_y^2}, \tag{14}$$

where the partial derivatives of $I(\boldsymbol{x},t)$ with respect to $x,\ y,\ t$ are represented as $I_x,\ I_y,\ I_t$, respectively, and $\bar{u}^k,\ \bar{v}^k$ are the locally-averaged values for the optical flow at iteration $k$ (computed via spatial filtering - e.g. Gaussian kernel smoothing). It is natural to think of the spatiotemporal gradient of our intensity function $\frac{dI(\boldsymbol{x},t)}{dt}$ as a single entity over the entire image with $3 \times N_x \times N_y \times N_t$ discrete 4-dimensional grid, where $N$ represents the number of points at which the gradient is analyzed. In general, the gradient entity is actually continuous in all dimensions, and can be best represented as a *continuous tensor field* - a convenient formulation for a Diderot implementation.

A more biologically-inspired model represents flow velocity gradients (the $\boldsymbol{v}$ in Horn-Schunk) as likelihood values corresponding to membrane potential of neurons [10]. Specifically, the model assumes a neural network composed of three cascading stages, inspired by the structure found in the visual cortex, with feed-forward, feedback, and lateral connectivity between neural units. The first stage of the cascade approximates an activation function of the spatially-integrated input signals - akin to the linear-nonlinear (LN) neuron encoding model. The literature expresses this in the form of an ordinary differential equation, where

$$\dot{x}^{(1)} = -x^{(1)} + f_{sample}\Big([x^{FF}]^\alpha * \Lambda_{space}\Big) * \Lambda_{vel}. \tag{15}$$

The change in likelihood is a function of the previous likelihood value and a sampled (nonlinear) function of the spatial convolution of a kernel $\Lambda_{space}$ with a quadratically-transformed (by $\alpha$) input signal, which is smoothed in the temporal domain (to omit discontinuous sharp velocity changes) by $\Lambda_{vel}$. An isotropic kernel, like a Gaussian, is a good model for the smoothing kernels in this model as it has been shown that the early stages of the visual system have these types of receptive fields [11]. The second stage in this model incorporates feedback connectivity to enhance and characterize the likelihood values encoded on the first. This is analogous to the connections between V1 and area MT in the higher stages of the cortex. Mathematically, this is represented as the ODE

$$\dot{x}^{(2)} = -x^{(2)} + x^{(1)}(1 + \beta x^{FB}). \tag{16}$$

This equation demonstrates the positive effect of feedback connectivity, but notice, only in the case where activity is present on that unit. If $x^{(1)}$ is very small, the feedback influence will be negligible. Finally, the third stage incorporates lateral connectivity, specifically, lateral inhibition - which provides for a divisive normalization of likelihood values. Not only does this mathematically bound the quantities calculated, it replicates a normalization phenomena seen in vivo through visual devices such as gain control [12]. The third and final stage for velocity gradient encoding is shown by the ODE

$$\dot{x}^{(3)} = -\gamma x^{(3)} + x^{(2)} * \Lambda^{+} - x^{(3)}\big(x^{(2)} * \Lambda^{-}\big), \tag{17}$$

where the convolution with kernels $\Lambda^{+}$, $\Lambda^{-}$ yield the traditional center-surround receptive fields of the simple cells in lower layers of V1. This sophisticated structure serves as the feature extraction stage described in the beginning of the optical flow section. The weighting and integration of these velocity gradient features is implemented via a similar three-stage cascade with more complicated kernel functions (i.e. asymmetric difference of Gaussians) performing integration of the gradients in a more abstract, higher space - see Fig. 4. For the sake of brevity, the equations will not be elaborated any more here, but can be found in detail in the literature [10]. We will focus on just the feature extraction (V1) stage of the bio-inspired model as the higher stages (MT, MST) that perform the integration and flow vectors are of a similar structure.

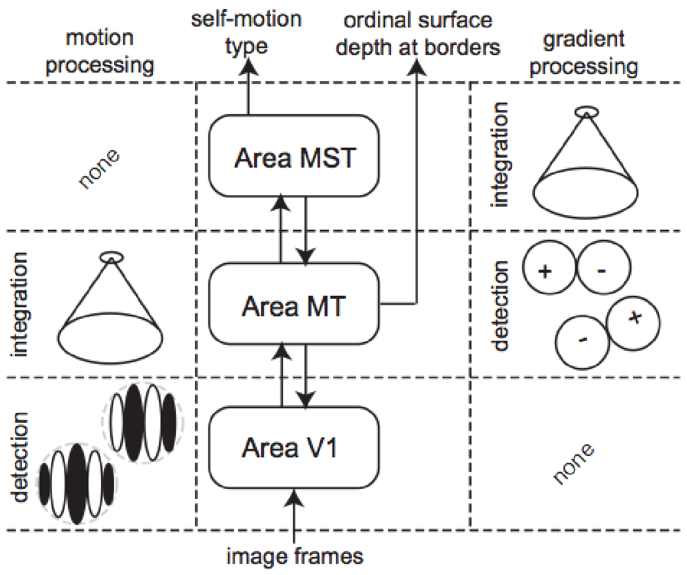


Figure 4: Process pathways in the bio-inspired optical flow model [10]

In this model the likelihood values $x$ exist, not only for each spatial location (i.e. topographic map of neurons), but also for every motion direction and motion speed that we care to resolve. From a higher level, this can be represented as a single entity (or neural circuit) of $N_x \times N_y \times N_d \times N_v$ elements, where $N$ corresponds to the number of $x$, $y$ spatial elements, $d$ directional resolution, and $v$ velocity resolution. Though of a different form than the Horn-Schunk intensity-based gradient entity, it also can be generalized and conveniently represented as a continuous tensor field.

## Initial Diderot experimentation and limitations

With the above sections in consideration, we can begin to approach using Diderot for implementing continuous tensor fields to represent neural circuits. The neural field $u_i(x,t)$ given in Eq. (8) is analogous to the feature-extraction stage of optical flow in that it utilizes a continuous field, $N_x \times N_t \times N_j$, where $N$ represents the number of the physical neural units at locations $x$, at $t$ times, on neural layers $j$. It performs a spatiotemporal smoothing via the double intgeral with the weighting function, sums across all layers, and accounts for additional stimulus and steady-state recovery time in determining activity. With this in mind, the first approach with Diderot was to create a basic feature-extractor from a time-varying input stimulus. Out of the neural field model, the Horn-Schunk model, and the bio-inspired (Neumann) model, the simplest calculations correspond to the Horn-Schunk model, so this was tried first.

In Horn-Schunk, the feature extraction is performed by calculating a localized-average of the spatiotemporal velocity vectors,

$$\bar{\boldsymbol{v}} = mean_{local}\Big(\frac{dI(\boldsymbol{x})}{dt}\Big) = \begin{bmatrix}\bar{u}\\ \bar{v}\end{bmatrix}. \tag{18}$$

In the simplest possible case, we can take two binary images and calculate the flow vectors. For this, we used the images from Fig. 5

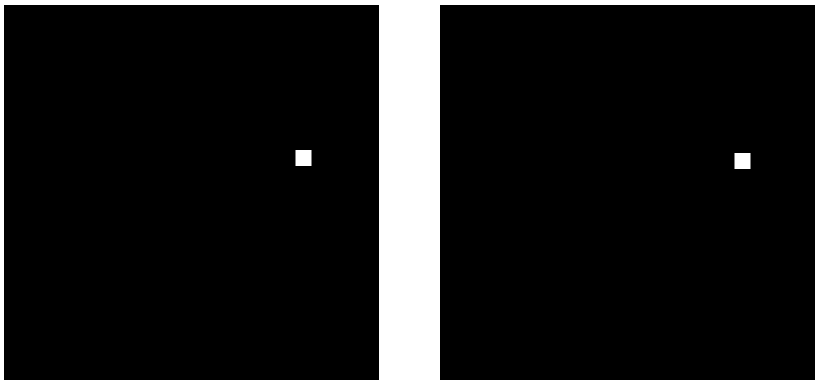

(a) First image (b) Second image

Figure 5: Diderot analysis and visualization examples

This image sequence had to be converted to a Diderot-friendly input. Diderot deals exclusively with the *.nrrd* format for input data. The nrrd file contains the metadata for mapping world space to image space coordinates as well as other information about the input [13]. Since this file format is not standard or widely-used, a file converter script *img_2_nrrd.py* was created in python which takes input image files (*.png, .jpg*, etc) and creates the metadata and nrrd file format from them. The world space coordinates for the nrrd file was calculated assuming that the $x$ and $y$ ranges went from 0 to 1 in world space, so the $dx$ and $dy$ were simply the inverse of the number of pixels in that direction. For example, a $640 \times 480$ image would have world space coordinates of $(0.0015625, 0)$ for $x$ and $(0, 0.0020833)$ for $y$.

The *img_2_nrrd.py* file consolidated the image sequence together in a $N_x \times N_y \times N_t$ nrrd data structure for input to Diderot. Fig. 6 shows the two images overlayed.

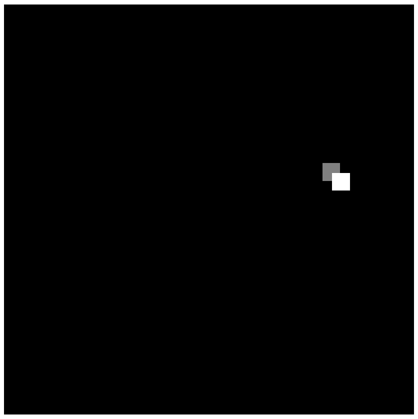


Figure 6: First and second image overlay

To verify that Diderot can compute the gradient with respect to each axis correctly, we first convolved our input with a `ctmr` kernel to form the continuous tensor field $F$. Since Diderot represents this as a continuous entity, the spatial and temporal granularity can be determined by the user, despite the resolution of the input. Instead of taking the gradient of a two-image sequence, we sampled 10 slices from the underlying continuous field, and calculated the gradient using the del operator from Diderot on each slice (with respect to each dimension $x,\ y,\ t$). The output of the result was ported to MATLAB. Fig. 7 shows the 7th time slice gradients calculated by Diderot.

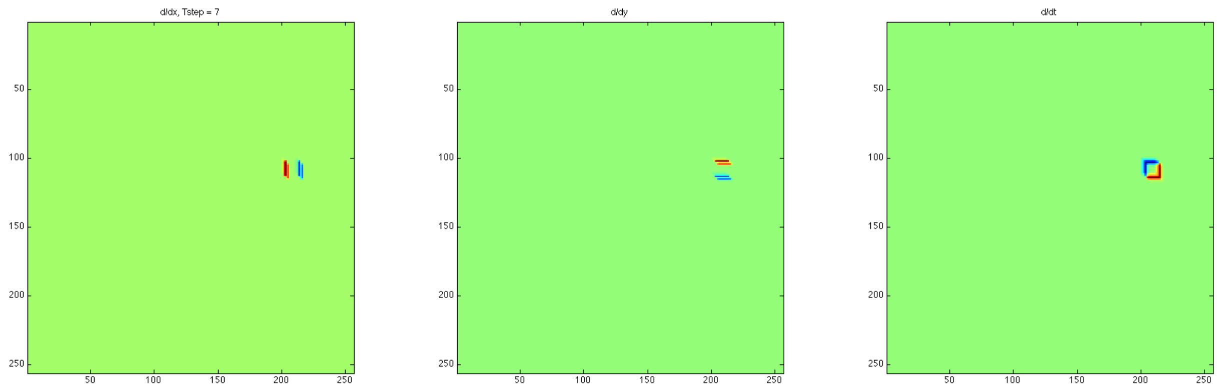


Figure 7: Display of $x,\ y,\ t$ gradients calculated from Diderot

This verified that Diderot can accurately transform the discrete input into a continuous field, and produce reliable gradient information. The next step was to perform iterative local averaging on the continuous tensor field to implement the Horn-Schunk algorithm of Eq. (13) - in other words calculate $\bar{u}^k,\ \bar{v}^k$. However, it quickly became apparent that this is a problem at this stage of development for Diderot. To implmement the aglorithm, the local average of the gradients needed to be calculated. The gradient operator in Diderot was able to generate another field from which we could probe at specific locations to extract the gradient tensor, however, there was no way to conveniently access neighboring points on the field and store an average of them to use for the next iteration - which is the computational core of the Horn-Schunk method.

The problem with the current state of Diderot is that it does not support global mutable memory [1] [14]. This is a great limitation in performing the feature extraction and integration stages for the optical flow methods discussed above, as well as the canonical neural field model from Eq. (8). All of these methods require *inter-strand communication* to enable the neural network to perform proper encoding; the three-stage cascade with feedforward and feedback connections of the bio-inspired model and the neural field weighting function all require the ability to communicate and update a common variable. Since Diderot does not have this feature enabled at this time, we must explore other ways to use the software to its strong points; primarily, as a visualization tool - which can be quite powerful for developing an intuition for certain phenomena discussed in computational neuroscience.

## Visualizing the phase plane

When considering other potential uses for Diderot, it was important to keep in mind the novelties of the software: the flexibility and convenience in dealing with continuous tensor fields at a high level, and the portable parallel framework optimized for image analysis and visualization. With this in mind, we explored various topics in computational neuroscience that could benefit from this model. A great way to understand the dynamics of various conductance-based neuron models is by visualizing its response in the *phase plane*. This section will discuss several basic neuron models and how they operate from the point of view of their phase portraits. A novel visualization technique called line integral convolution (LIC) will also be introduced. Finally, the results of using Diderot to visualize phase portraits using LIC will be discussed along with the code developed and relevant figures.

### Neuron dynamics and bifurcation

Three neuron models were explored: Hodgkin-Huxley, Morris-Lecar, and Fitzhugh-Nagumo. The Hodgkin-Huxley model is implemented via evaluating KCL across the neuron membrane with the assumption that the cross-membrane currents are attributed to four (activation) voltage-gated potassium ion (K$^+$) channels, three (activation) and one (inactivation) voltage-gated sodium ion (Na$^+$) channels and an ohmic leak current attributed to chloride ions (Cl$^-$) [15]. The dynamics of the gates are represented as probabilities via gating variables $n$, $m$, $h$ for K$^+$ activation, Na$^+$ activation, and Na$^+$ inactivation, respectively. The result is the following set of the nonlinear differential equations which form the Hodgkin-Huxley neuron model:

$$\begin{aligned} C\dot{V} &= I - \overbrace{\bar{g}_K n^4 (V - E_K)}^{I_K} - \overbrace{\bar{g}_{Na} m^3 h (V - E_{Na})}^{I_{Na}} - \overbrace{g_L (V - E_L)}^{I_L} \\ \dot{n} &= \alpha_n(V)(1-n) - \beta_n(V)n \\ \dot{m} &= \alpha_m(V)(1-m) - \beta_m(V)m \\ \dot{h} &= \alpha_h(V)(1-h) - \beta_h(V)h. \end{aligned} \tag{19}$$

The phase plane for neuron models is generally defined to be a plot over one of the gating variables vs. voltage response $V$ - though it can be generalized to be three or four-dimensional when more than one gating variable is plotted. In the Hodgkin-Huxley case, the gating variable $n$ can be plotted against $V$ to observe how the neuron is expected to behave when injected with an input current $I$. This plot can give an idea of where the dynamical system has stability or instability. An example of this is shown in Fig. **??**.

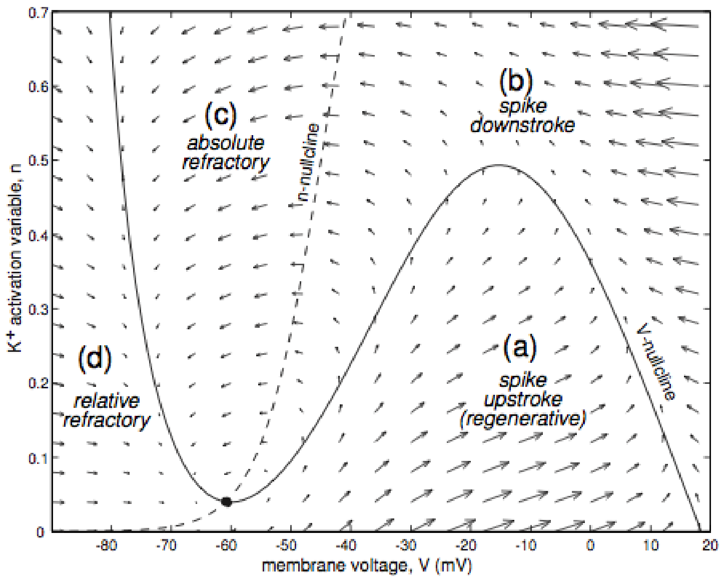


Figure 8: Phase portrait of a reduced conductance-based model, $I_{Na,p} + I_K$, with $V$ and $n$ nullclines [16]

The plot shows the upstroke and downstroke paths of a voltage spike via vector field gradients. These arrows are telling how we can expect the state of the neuron (in $V - n$ space) to change in time. The nullclines are paths in the plane where the respective states stay constant (i.e. $\dot{V},\ \dot{n} = 0$) according to the equations. Each nullcline divides the plane into two regions, where on either side of each nullcline, the variable is changing in opposite directions. For example, beneath the $V$-nullcline (solid line) the voltage is increasing in time, but above it is decreasing. This allows a labeling of the neuron state in different regions of the plane, as shown in Fig. 8. The intersection of these nullclines is an *equilibrium point* and reflects where the output response of the neuron has no change with respect to either state - i.e. a resting state of the neuron model.

Given certain conditions, namely the parameters used in the model and the input current magnitude, the trajectory of the neuron state can either come to rest at the equilibrium point, or it can enter into a periodic, oscillatory path known as a *limit cycle* (i.e. neuron fires). This qualitative change in the phase portrait is known as *bifurcation* and is essential in understanding how a neuron fires from a given input stimulus.

Neurons are grouped into two basic types of excitability, type I and type II neurons. These types characterize how an input stimulus causes the dynamics of the underlying neuron model to change from non-firing to firing. With regard to bifurcation in the phase plane, type I models realize a limit cycle via saddle-node bifurcation of equilibrium points - where an unstable node and stable node move closer together until they coalesce into a single point and annihilate each other - resulting in a periodic stability in the form of the limit cycle, whereas type II models achieve bifurcation via a single equilibrium path changing stability due to Hopf bifurcation[17]. The Hodgkin-Huxley neuron is of type II, but there are other neuron models, like the Morris-Lecar model, which can show type I and type II bifurcations, depending on its parameters [18]. Since both type I and type II neurons have been observed *in vivo*, it becomes useful to compare and visualize several neuron models in the phase plane. The Morris-Lecar model is a reduced excitation model consisting of two non-inactivating conductances, $n$ (K$^+$ recover variable), and $m$ (Ca$^{++}$ conductance channel variable), where $m$ is typically modeled to be at its steady-state $m_\infty(V)$. The reduced model can thus be described by the nonlinear differential equations:

$$
\begin{aligned}
C\dot{V} &= I - g_L(V - E_L) - g_{Ca}m_\infty(V)(V - E_{Ca}) - g_K n(V - E_K) \\
\dot{n} &= (n_\infty(V) - n)/\tau_n(V) \\
m_\infty(V) &= \frac{1}{2}\Big(1 + \tanh(\frac{V - V_1}{V_2})\Big) \\
n_\infty(V) &= \frac{1}{2}\Big(1 + \tanh(\frac{V - V_3}{V_4})\Big) \\
\tau_n(V) &= \phi \cosh\Big(\frac{V - V_3}{2V_4}\Big),
\end{aligned}
\tag{20}
$$

where $V_1,\ V_2,\ V_3,\ V_4$ are tuning parameters. Finally, the Fitzhugh-Nagumo model can also be analyzed. This model is a reduced Hodgkin-Huxley-type model which guarantees a cubic $V$-nullcline with $w$ acting as the recovery activation variable. Fitzhugh-Nagumo has tuning parameters $a,\ b,\ c$ and is defined by the differential equations [16]:

$$
\begin{aligned}
\dot{V} &= V(a - V)(V - 1) - w + I \\
\dot{w} &= bV - cw.
\end{aligned}
\tag{21}
$$

Accurately depicting the bifurcation of neuron states is vital to replicating how neurons encode information *in vivo*. It is helpful comparing the different neuron models in the phase plane, as it is easier to analyze then something like spike trains from each model. However, the gradients display of Fig. 8 can be an eyesore, making it hard to compare between various models (e.g. Hodgkin-Huxley vs. Morris-Lecar). Additionally, the vector field usually has to be distorted to illustrate the model's dynamics. It would be helpful to visualize the phase plane via another medium, which is discussed in the following section.

## Line integral convolution

The vector field from the phase plane of Fig. 8 is convenient for visualizing the general flow of the states, but it would be nicer to have an image displaying the vector field as a continuous surface. Line integral convolution is a visualization technique that is specifically designed to transform a vector field into a solid image that better shows how the vector field looks continuously. The technique requires two inputs, the vector field grid $V$ and a texture image $F$. The local behavior of the vector field is approximated by a local stream line that starts at the center of each pixel in the field and moves outward in "forward" and "backward" directions. The position vector is incremented by 0.5 in both $x$ and $y$, where an entire pixel corresponds to a $1 \times 1$ unit, to ensure the stream line starts exactly in the center of the field vector pixel. The coordinates of the stream line are then calculated as

$$P_0 = (x + 0.5, y + 0.5) \tag{22}$$

$$P_f^k = P_f^{k-1} + \frac{V\big(\lfloor P_f^{k-1} \rfloor\big)}{||V\big(\lfloor P_f^{k-1} \rfloor\big)||} \Delta s_f^{k-1} \tag{23}$$

$$P_b^k = P_b^{k-1} - \frac{V\big(\lfloor P_b^{k-1} \rfloor\big)}{||V\big(\lfloor P_b^{k-1} \rfloor\big)||} \Delta s_b^{k-1}, \tag{24}$$

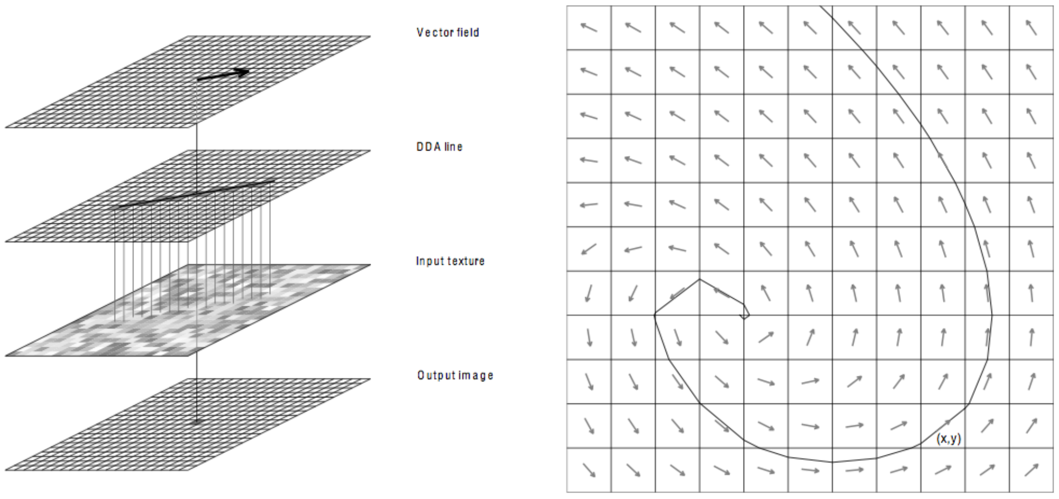
where $P_f^k,\ P_b^k$ correspond to the stream line position indices of forward and backward directions at the $k$th iteration, $V\big(\lfloor P^{k-1} \rfloor\big)$ is the gradient field value at the floored index of $V$, $\Delta s_f^{k-1},\ \Delta s_b^{k-1}$ are the (always positive) step sizes in the forward and backward directions, respectively [19]. In general, a convolution kernel function $k(w)$ is computed and used as a weight in the LIC over the image, where

$$h^k = \int_{s^k}^{s^k + \Delta s^k} k(w)dw. \tag{25}$$

The result of Eq. (25) is then used to weigh the underlying image texture at each stream line index $i$; this result is then summed over the length of the streamline $l_f,\ l_b$ and normalized by the sum of the weights, where

$$LIC(x,y) = \frac{\sum_{k=0}^{l_f} F\big(\lfloor P_f^k \rfloor\big) h_f^k + \sum_{k=0}^{l_b} F\big(\lfloor P_b^k \rfloor\big) h_b^k}{\sum_{k=0}^{l_f} h_f^k + \sum_{k=0}^{l_b} h_b^k}. \tag{26}$$

The algorithm is well understood by studying Fig. 9



(a) Integration process (b) Stream line formation across vector field

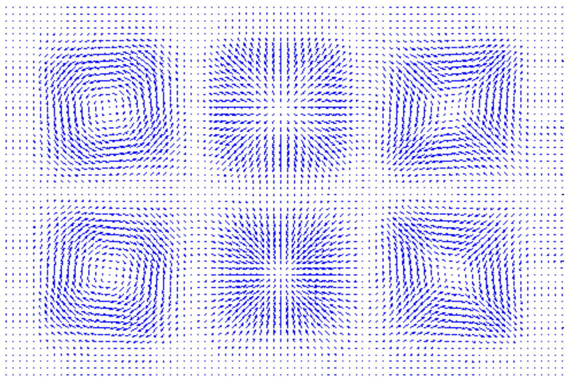
Figure 9: LIC algorithm visualization [19]

In most cases, the underlying image texture is a white noise image; this way no local areas of the vector field yield heavier LIC lines than any others. Additionally, a box filter for $k(w)$ can be used over the underlying texture image, so the inclusion of $h^k$ in the summation terms can be taken out and used as a single scaling value (akin to step size) across the entire stream line to be summed over.

LIC is a great application for a Diderot program as not only are the calculations done on an independent pixel-by-pixel basis, but the flexibility of having a user-defined granularity to the input data allows for a very detailed LIC plot, despite a low-resolution vector field. Since the continuous tensor field object in Diderot can interpolate coarse input data, the midpoint method step can be relaxed to a simpler, forward/backward Euler integration. Additionally, the box filter can be used, so the kernel function $k(w)$ doesn't have to be calculated for each pixel. This simplifies the equation, algorithmically, to

$$\begin{aligned} P_0 &= (x,y) \\ P_f^k &= P_f^{k-1} + \Delta s V(P_f^{k-1}) \\ P_b^k &= P_b^{k-1} - \Delta s V(P_b^{k-1}) \\ \Rightarrow LIC(x,y) &= \frac{\sum_{k=0}^{2l} F(P_f^k) + F(P_b^k)}{2l+1}, \end{aligned} \tag{27}$$

where the step size $\Delta s$ and stream line length $l$ are the same for forward and backward. The Diderot implementation of LIC implements Eq. (27) with the exception that it adds an additional $|V(P_0)|$ bias term to the numerator of $LIC(x,y)$ for each pixel. Diderot's LIC demo yields the results from Fig. 10.


(a) Original vector field

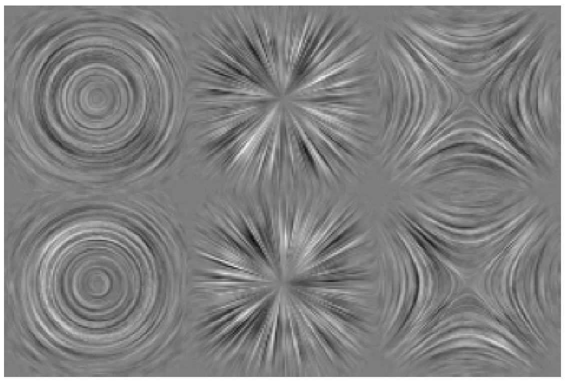

(b) LIC with white noise texture

Figure 10: LIC results from Diderot program *lic.diderot*

## Experiments visualizing phase portraits

The first step in the experimentation was to generate the vector gradient field for the neuron models in the phase plane. This involved choosing parameters for each neuron model that would demonstrate the system dynamics and were realistic. The parameters for the Hodgkin-Huxley model were chosen as:

$$\begin{aligned} E_L = 10.6mV, \quad E_{Na} = 120mV, \quad E_K = -12mV \\ g_L = 0.3mS, \quad g_{Na} = 120mS, \quad g_K = 36mS, \end{aligned} \tag{28}$$

For Morris-Lecar, the model parameters were chosen as:

$$\begin{aligned} E_L = -60mV, \quad E_{Ca} = 120mV, \quad E_K = -84mV \\ g_L = 2mS, \quad g_{Na} = 4mS, \quad g_K = 8mS \\ V_1 = -1.2, \quad V_2 = 18, \quad V_3 = 2, \quad V_4 = 30, \quad \phi = 0.2. \end{aligned} \tag{29}$$

For both Hodgkin-Huxley and Morris-Lecar $C$ was set to 1 $\mu$F/cm$^2$ and the voltage range went from -90 mV to 40mV. Finally, for Fitzhugh-Nagumo, the model parameters were chosen as:

$$a = 0.1 \quad b = 0.01 \quad c = 0.02, \tag{30}$$

where the voltage range was from -0.6 to 1.2 V. Utilizing the equations from Eq. (19), Eq. (20), and Eq. (21), we generated the phase portraits for the range of voltages stated, and the recovery variables from 0 to 1. The `quiver` function in MATLAB was used to plot and visualize the vector fields for each neuron model. Fig. 11 shows initial attempts at plotting the phase portraits. The nullclines of the plots were calculated by solving for $\dot{V}, \ \dot{n}, \ \dot{w} = 0$ in the differential equations Eq. (19), Eq. (20), and Eq. (21).

It is apparent that the visualization is not ideal. The gradient vectors are so distorted by the different units of change, as $dV$ has change in mV, which is several orders of magnitude higher than $dn$ - change in probability, which is bounded to be between 0 and 1. The $dV$ values can be force-normalized, but this can result in a distorted phase portrait if done incorrectly as bad $dV$ values due to improper normalization skew the gradient vectors' angles. It became necessary to verify any normalization of the quiver plot by overlaying actual limit cycle runs over the gradient vectors. The "multi-path" limit cycle runs start iterating over each model's differential equations with initial $V$, $n$ values at every point sampled from the phase portrait gradients grid. As the limit cycle paths are how the states actually traverse per unit time, the gradient vectors for the phase portrait should follow these paths exactly; this provided a way to verify our normalization of the gradient vectors such that they depict an accurate phase portrait.

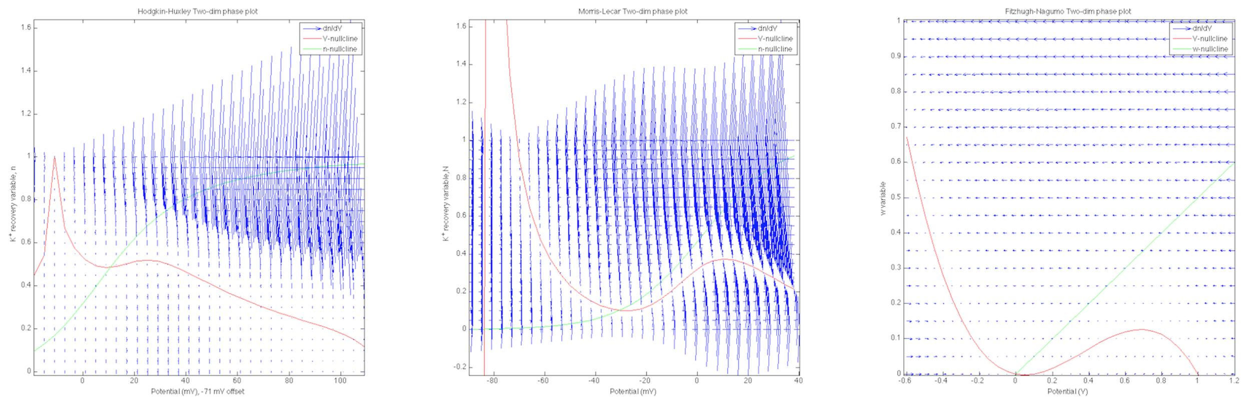


Figure 11: Un-normalized phase portraits for each neuron model

The normalization technique we used involved taking a range of the input potentials and dividing $dV$ by a factor $\eta$ times the range, or

$$dV_{norm} = dV/\eta\big(\max(V) - \min(V)\big). \tag{31}$$

Utilizing Eq. (31) and overlaying the multi-path limit cycle runs yielded an accurate representation for the phase portraits, as seen in Fig. 12. The current values for each plot were $I = 30\mu A/cm^2$, $I = 80\mu A/cm^2$, $I = 0\mu A/cm^2$ for the Hodgkin-Huxley, Morris-Lecar, and Fitzhugh-Nagumo models, respectively.

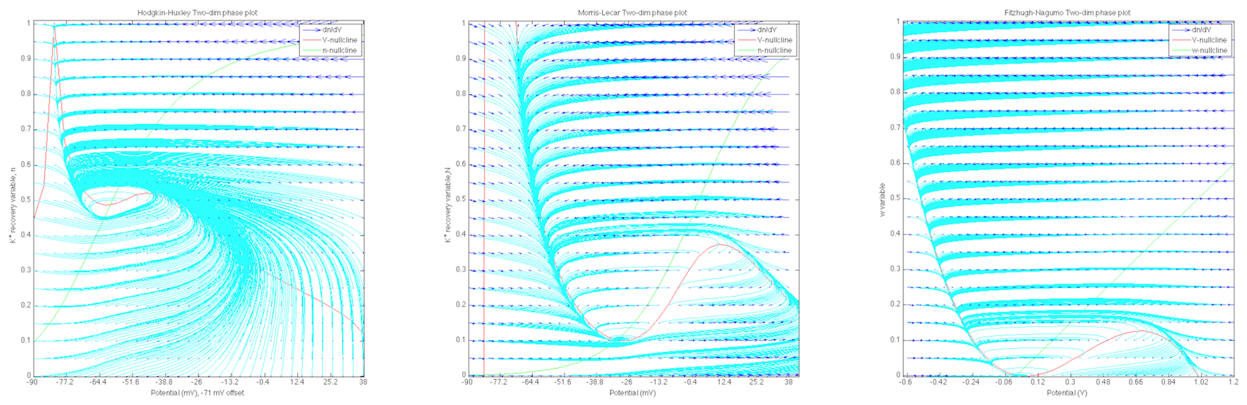
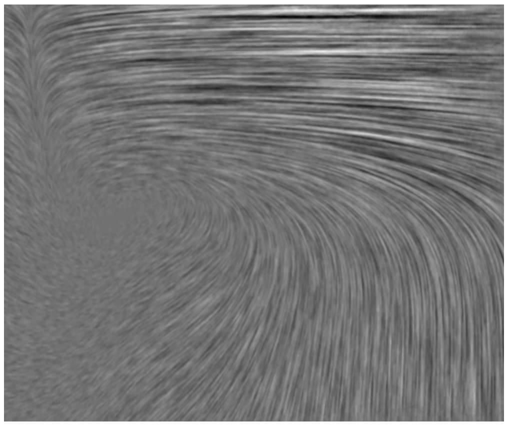
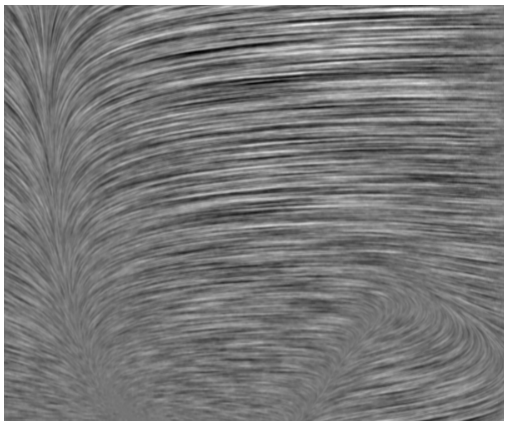


Figure 12: Normalized phase portraits for each neuron model

With the shape of the phase portraits verified in this manner, we then wanted to test Diderot's ability to import and perform LIC over the inputs. Each phase portrait gradient vector field was exported from MATLAB as a $2 \times 33 \times 21$ .h5 file. This file was then conerted to a .nrrd file via the *h5_to_nrrd.py* Python script. This supplied the world-space coordinates in much the same way as *img_2_nrrd.py*. These .nrrd files served as inputs to *licp.diderot* - LIC visualization code that specifically handled the phase portrait data files. The code used a `tent` kernel to convolve the gradient field into a continuous tensor field and we probed it at $600 \times 500$ points to perform the LIC. The step-size was best found to be $\Delta s = 0.0005$, with $l = 200$. A white-noise image was used as the underlying texture for convolution and was also imported to a continuous tensor field with a `tent` kernel. The results of the LIC can be seen in Fig. 13.


(a) Hodgkin-Huxley LIC


(b) Morris-Lecar LIC

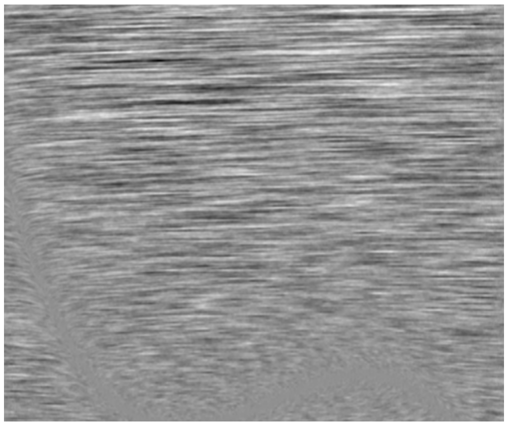

(c) Fitzhugh-Nagumo LIC

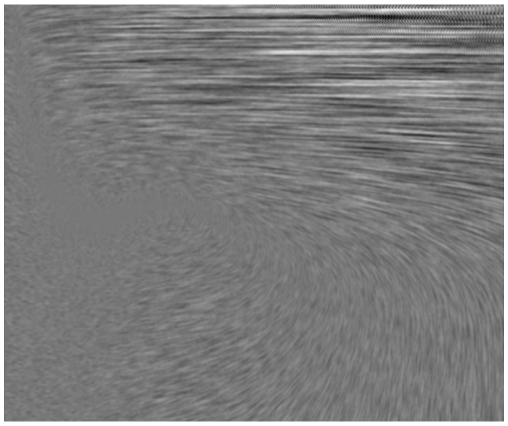
Figure 13: LIC results from Diderot program *licp.diderot*

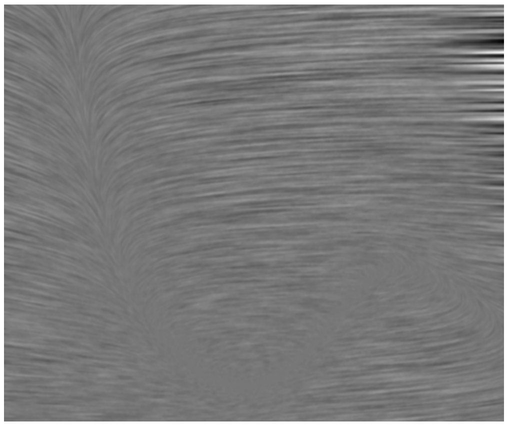
The LIC does a nice job of helping to visualize a "continuous" behavior of the models in the phase plane. The transition states are more obvious and the $V$ and $n$ nullclines appear inherently in the flow of the images.

The next step forward was to remove MATLAB's role in generating any of the data. We realized that interpolating the gradient vectors in to Diderot through a kernel was a technical distortion of the phase portrait, even if that kernel was linear. We set out to implement the neuron models directly in Diderot, calculate the phase portraits, perform the LIC of the calculations, and finally generate the correct output. The input data to Diderot would then be the range of $V$ and $n$ that we wanted to calculate the phase portraits over (calculated via *vnirange.m*). Interpolating the range into a continuous tensor would not be an issue, as it is merely an input to the model, not an output from nonlinear equations. Despite Diderot's limitations with not being able to define function handles, or share mutable memory, the models were successfully implemented and the LIC was calculated from the output of the models in the same way as with *licp.diderot* - all within one Diderot file.

The outputs of the figures were not quite on par with the *licp.diderot* results, however. Generating a smooth-looking LIC portrait requires many iterations per strand and a larger step size to sample from the underlying white noise texture. These parameters are limited for a few reasons. The amount of calculations that the code performs with the addition of the neuron model is far greater than in just calculating LIC from input vectors. As a result, the programs were more sensitive to bus errors and segmentation faults. This is due to limitations of the software package, hardware that is not optimized for the program, and boundary conditions that are not checked for in the Diderot code file itself. There was much experimentation done with the checking of boundary conditions (to ensure the field was being probed where expected and not running out of the program somewhere), however, a universal solution to the problem was not found. Regardless, the phase portrait/LIC generating code for each neuron model does yield reliable results, as shown in Fig. 14.

Note - these parameters are the exact same as the phase portraits of Fig. 13.


(a) Hodgkin-Huxley LIC


(b) Morris-Lecar LIC

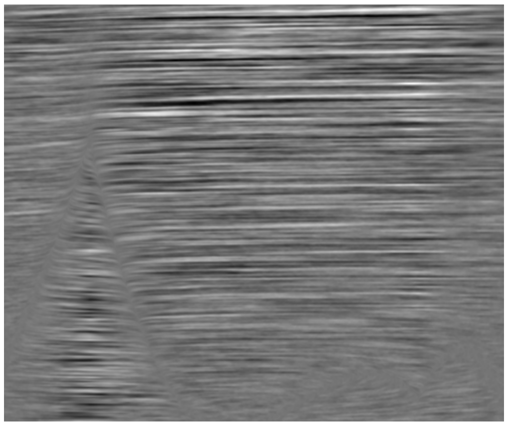

(c) Fitzhugh-Nagumo LIC

Figure 14: LIC results from Diderot programs *licHH.diderot*, *licML.diderot*, and *licFN.diderot*

With the code for a static current injection into the models developed, the next step was to evaluate the phase plane with a varying current. The code from Diderot programs *licHH.diderot*, *licML.diderot*, and *licFN.diderot* was easily modified to incorporate a bandlimited, time-varying current stimulus, which was imported to Diderot as a one-dimensional continuous field (the current stimulus was generated via *vnirange.m*). The programs *licHHInject.diderot*, *licMLInject.diderot*, and *licFNInject.diderot* were used to generate the time-varying LIC/phase portrait plots. The *licHHInject.diderot* file was unsuccessful in calculating the LIC for the entire input stimulus. The program would repeatedly report segmentation faults during runtime. It was speculated that the cause of this was the computational complexity of running the Hodgkin-Huxley model for many cycles in Diderot. However, this was resolved when running via OpenCL on the Huxley cluster.

As an additional visualization tool, the LIC plots were overlaid with the velocity vectors, nullclines, and limit cycles calculated in MATLAB, and the dynamics of the phase portrait was captured as a function of the input stimulus. Also, the neuron's output response was plotted to show the resulting spike train alongside the phase portrait. This graph was captured and saved as a video file so one can watch the process happen in real time. A screen shot of the Morris-Lecar model is shown in Fig. 15 and Hodgkin-Huxley in 16.

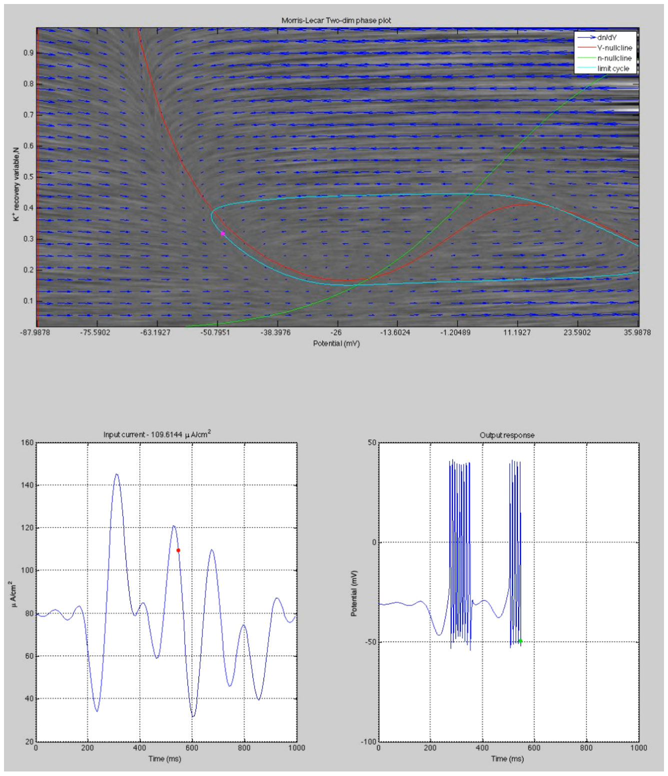


Figure 15: Screenshot of the LIC visualization alongside input and output of Morris-Lecar neuron model

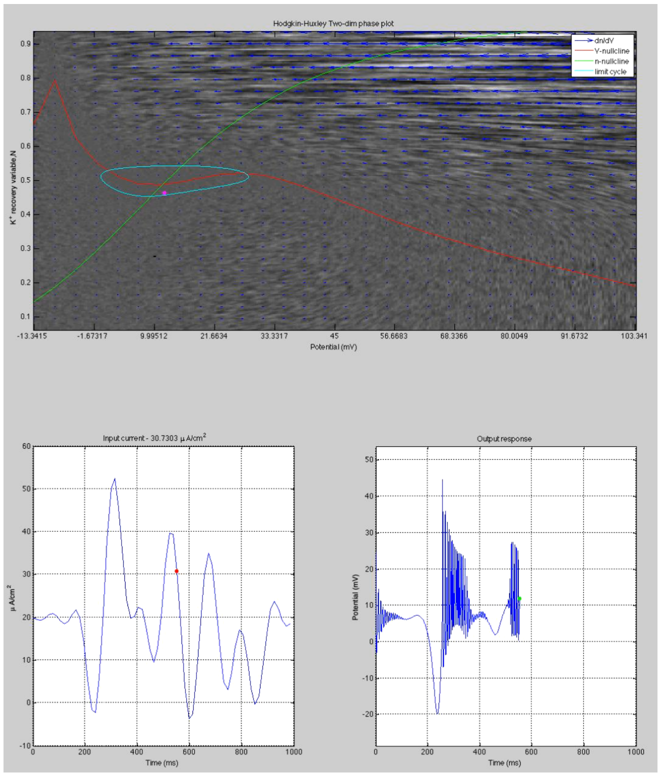


Figure 16: Screenshot of the LIC visualization alongside input and output of Hodgkin-Huxley neuron model

## Summary and further direction

Diderot is a great tool for interacting with data from a high-level domain. It provides the flexibility to manipulate this data at user-specified granularity, regardless of the input resolution. Image visualization and analysis algorithms that rely on mostly independent computations are well fit for use in Diderot. However, it is not ideal at this time to implement high-level brain operators or perform any sort of sophisticated feature extraction on input sequences. Despite this limitation, we were able to use Diderot as a visualization tool to bring about a convenient and intuitive way to understand the phase portrait of various neuron models.

It would be interesting to see how a three-dimensional LIC program could work to produce a phase portrait over the $V$, $n$, $m$ plane. This is applicable only to the Hodgkin-Huxley and Morris-Lecar neuron models and would yield a more sophisticated depiction of the underlying model dynamics. This is because the steady-state activation variable $m_\infty(V)$ would be replaced by the actual value of $m$ - yielding quantities that are truer to the original model formulations of Eq. (19) and Eq. (20). LIC is inherently capable and flexible enough to support three-dimensional data [19], the question is how to visualize the output of the data in a meaningful way - especially with regard to time-varying input current.

Implementing Diderot on a faster hardware platform that supports OpenCL is another area to further explore [20]. Diderot's strengths are based on the parallel programming architecture and currently support the OpenCL backend. The level of segmentation faults and bus errors actually went down or stopped altogether on some of the programs with accelerated hardware, particularly when we ran *licHHInject.diderot* using the OpenCL backend on the Huxley cluster. In general, the speed improvement using parallel processing is already evident when running a sophisticated program like *licMLInject.diderot*.

| | Sequential C | Parallel C | OpenCL | CUDA |
|---|---|---|---|---|
| 2.7 GHz Core i5 iMac | 30 mins 43.8 sec | 9 mins 20 sec | Memory alloc. error (runtime) | N/A |
| Huxley Cluster | 94 mins 39.6 sec | 25 mins 15 sec | Memory alloc. error (runtime) | N/A |

Table 1: Benchmark for *licMLInject.diderot*

Diderot's OpenCL backend does not yet support the amount of memory needed for *licMLInject.diderot* and the CUDA backend is still in development; however the potential speed gains in computation time from these frameworks is very large. For additional benchmarking, the *licp.diderot* file, which takes in the phase portrait of a neuron model and performs the LIC, was ran on the local machine and the cluster. The advantages in utilizing a backend with accelerated hardware is apparent as the OpenCL's runtime on Huxley was 9 times faster than on the serial-based run on the iMac; this would be expected to scale for more sophisticated Diderot programs.

| | Sequential C | Parallel C | OpenCL | CUDA |
|---|---|---|---|---|
| 2.7 GHz Core i5 iMac | 7.71 sec | 2.04 sec | 59.39 sec | N/A |
| Huxley Cluster | 9.59 sec | 2.71 sec | 0.85 sec | N/A |

Table 2: Benchmark for *licp.diderot*

Finally, working with Diderot's authors on implementing the abilities to have inter-strand communication, global mutable memory during thread processes, and a CUDA-supported backend [21] will make Diderot a much more functional medium for experimenting with high-level brain operators and continuous neural fields. These concepts are essential for implementing the latest models and algorithms for neural networks and neural encoding discussed in today's literature. With these functionalities, Diderot would be well equipped for further research in this realm.

## Acknowledgements

Special thanks to Yiyin Zhou and Nikul Ukani for directing my progress and giving feedback and suggestions on a daily basis. Thanks to Lev Givon for helping me with running experiments on the Huxley cluster. Thanks to Prof. Lazar for supervising the project. And thanks to the rest of the Bionet lab for various tips and consulting when I had questions.

## References

[1] Gordon Kindlmann. Diderot: A parallel dsl for image analysis and visualization. `http://youtu.be/m1cwdZL8mXk`, October 2013.

[2] Charisee Chiw, Gordan Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. Diderot: A parallel dsl for image analysis and visualization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*, pages 111–120, June 2012.

[3] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[4] J.S. Griffith. A field theory of neural nets: I: Derivation of field equations. *The bulletin of mathematical biophysics*, 25(1):111–120, 1963.

[5] Shun-ichi Amari. Dynamics of pattern formation in lateral-inhibition type neural fields. *Biological Cybernetics*, 27(2):77–87, 1977.

[6] James A. Bednar. Topographica: Building and analyzing map-level simulations from Python, C/C++, MATLAB, NEST, or NEURON components. *Frontiers in Neuroinformatics*, 3:8, 2009.

[7] Daniel Clark. Investigating topographic neural map development of the visual system. self, May 2013.

[8] J. L. Barron, D. J. Fleet, and S. S. Beauchemin. Performance of optical flow techniques. *INTERNATIONAL JOURNAL OF COMPUTER VISION*, 12:43–77, 1994.

[9] Berthold K. P. Horn and Brian G. Schunck. Determining optical flow. *ARTIFICAL INTELLIGENCE*, 17:185–203, 1981.

[10] Florian Raudies, Stefan Ringbauer, and Heiko Neumann. A bio-inspired, computational model suggests velocity gradients of optic flow locally encode ordinal depth at surface borders and globally they encode self-motion. *Neural Computation*, 25(9):2421–2449, 2013.

[11] Peter Dayan and L. F. Abbott. *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. The MIT Press, 2005.

[12] Odelia Schwartz and Eero P. Simoncelli. Natural signal statistics and sensory gain control. *Nature neuroscience*, 4(8):819–825, August 2001.

[13] Definition of nrrd file format. `http://teem.sourceforge.net/nrrd/format.html`, December 2013.

[14] John Reppy and Gordan Kindlmann. Re: Diderot mutable memory question. Email, October 2013.

[15] F. Gabbiani and S.J. Cox. *Mathematics for Neuroscientists*. Elsevier science & technology books. Elsevier Science, 2010.

[16] E.M. Izhikevich. *Dynamical Systems in Neuroscience*. Computational neuroscience. MIT Press, 2007.

[17] Germn Mato and Ins Samengo. Type i and type ii neuron models are selectively driven by differential stimulus features. *Neural Computation*, 20(10):2418–2440, 2008.

[18] Kunichika Tsumoto, Hiroyuki Kitajima, Tetsuya Yoshinaga, Kazuyuki Aihara, and Hiroshi Kawakami. Bifurcations in morris-lecar neuron model. *Neurocomputing*, 69(4-6):293–316, 2006.

[19] Brian Cabral and Leith Casey Leedom. Imaging vector fields using line integral convolution. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 263–270, New York, NY, USA, 1993. ACM.

[20] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[21] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2010. Version 3.2.

## Appendix: README code descriptions

```
Diderot research project README code/data descriptions:

matlab_code:
F_FN - Fitzhugh-Nagumo diff eq, function handle used in calculating limit
   cycles via lc.m
F_HH - Hodgkin-Huxley diff eq, function handle used in calculating limit
   cycles via lc.m
F_ML - Morris-Lecar diff eq, function handle used in calculating limit
   cycles via lc.m
h5write1 - function used for exporting h5 files
hstest - imports box1 and box4 pngs, plots them, imports gradient of the
   box pngs calculated from Diderot ('hs.txt') and imagesc's them. It also
    uses the basic Horn-Scuhnk optical flow method to find the flow
   vectors of box1 and box4
lc - limit cycle code that calls on a diff eq function handle to propagate
    the states
nrrdplot - plot data from input nrrd file (usually 2-dimensional or vector
    field input nrrds)
nrrdread - Mathworks-provided function for importing nrrd files to MATLAB
phasequiver - Function which takes in an input current, a neuron model and
    a rnage of V/n values and computes the phase portrait of quiver
   vectors and nullclines
phasevidfromdid - Function which computes limit cycles, quivers, and plots
    LIC of specified neuron model and creates a time-based video file of
   the input current, neuron response, and phase response
plotphasefromdid - used to verify that the phase response calculated from
   a Diderot output is realistic, yields a quiver plot of vector field of
   Diderot txt file
pp - plots phase portrait (PP)'s of HHN, ML, and FN models as well as runs
    the multi-path limit cycles to verify that the quiver plots are
   normalized properly
ppinject - plots and records video response of the PP of each of HHN, ML,
   and FN models with quiver and nullcline information
vnirange - creates vn"".h5 and It.h5 where "" corresponds to a neuron
   model and the h5 files serve as inputs to the h5_to_nrrd.py script to
   convert for input to the licInject Diderot files

diderot_code:
- licFN - LIC performed for the Fitzhugh-Nagumo model for constant input
   current
- licFNInject - Calculate PRC gradients of Fitzhugh-Nagumo model and
   perform LIC over a range of injected input current values
- licHH - LIC performed for the Hodgkin-Huxley model for constant input
   current
- licHHInject - Calculate PRC gradients of Hodgkin-Huxley model and
   perform LIC over a range of injected input current values
- licML - LIC performed for the Morris-Lecar model for constant input
   current
- licMLInject - Calculate PRC gradients of Morris-Lecar model and perform
   LIC over a range of injected input current values
```

```
- licp - Import phase response curve generated from MATLAB, converted to
   nrrd, and output its LIC (done through unu)
- ppFN - Calculate (phase portrait) PP gradients in Diderot for the
   Fitzhugh-Nagumo spiking model for a given constant current
- ppHH - Calculate PP gradients in Diderot for the Hodgkin-Huxley spiking
   model for a given constant current
- ppML - Calculate PP gradients in Diderot for the Morris-Lecar spiking
   model for a given constant current

python_code:
- h5_to_nrrd - converts h5 file to nrrd type
- img_2_nrrd - converts img files (jpg, png, etc) to nrrd type

data: input/output files generated from MATLAB, Diderot, and Python
   scripts

optic_flow: early code and data files for optic flow experiments
```