Pipeline overview

phases

preprocessing

Image filtering operations Normalization Motion estimation and compensation Layer classification foreground/background Cell/neuropil/vessel

time series analysis

Prediction and factorization of temporal components/contaminants Model: Unidirectional trend (sources...) Oscillatory ("seasonal") Unpredictable

spatial signal deconvolution (unmixing)

signal localization and extraction

Segmentation of cells Overlap and "disconnected" regions Initial vs incremental cell identification

active signal analysis

Signal statistics Shape and stability of probabilistic distribution Pixel level vs cell level

independent cellular activity measures

Quality Complexity Spatial Temporal

network analysis

Computing cost:

boundary analysis for iOS and config

Storage cost:

compression and imposed boundaries # Image Processing

This section borrows from AIM-1 and AIM-2 of the prospectus.

Procedures for Calcium Imaging

The general goal of processing image data from functional fluorescence imaging experiments is to restructure raw image data in a way that maps pixels in each image frame to distinct individual cells or subcellular components, called 'Regions-Of-Interest' (ROI). Pixel-intensity values from mapped pixels are typically then reduced by combination to single dimensional 'trace' time-series. These traces indicate the fluorescence intensity of an individual neuron over time, and the collection approximates the distinct activity of each and every neuron in the microscope's field of view. However, this task is made difficult by motion of the brain throughout the experiment, and also by the apparent overlap of cells in the image plane captured from the camera's 2-dimensional perspective. These issues can be partially mitigated with a few image pre-processing steps – alignment of images to correct for motion being the most critical. These options are described in the Methods & Approaches section below. Most software packages geared specifically toward functional imaging implement either of two basic classes of pixel->cell mapping algorithms. One approach is to use image-segmentation routines for computer vision, which seeks to combine adjacent pixels into distinct spatially segregated regions representing objects in the image.

The other common approach is to perform an eigenvalue decomposition on the covariance matrix from a stack of image frames (also called spectral decomposition, or Principal Component Analysis, PCA), resulting in an assembly of basis vectors defining the weighting coefficients for each pixel. Multiplying the basis-vectors (i.e. "components") with all frames produces a one-dimensional trace for each component. The linear combination is similar to the weighted image-segmentation method in that it assigns fractional coefficients to pixels. However the procedure for computing the covariance matrix employed by PCA operates on as many pixels as are in the image, multiplying each with every other pixel – a problem with np2 complexity, where p is the number of pixels in the image. I mention these issues inherent to PCA not because this project will attempt to address them, but because this project was initiated following tremendous difficulty attempting to use PCA-based cell sorting methods with large datasets.

Computer Software Environments for Image Processing

The widespread usage of MATLAB in neuroscience communities lends potential for greater usability and easier adaptation to software developed in this environment. While software development environments with a focus on "ease-of-use" have traditionally presumed crippling sacrifices to computational performance, this assumption is getting to be less accurate.

Standard programs include ImageJ, the built-in routines in MATLAB's Image Processing Toolbox, Mosaic from Inscopix, which is merely a compiled version of MATLAB routines which uses the MATLAB engine, Sci-Kits Image for Python,

and a remarkable diversity of other applications. MATLAB is a commercial software development platform which is geared toward fast production and prototyping of data processing routines in a high-level programming language. It implements several core libraries (LINPACK, BLAS, etc.) that make multi-threaded operations on matrix type data highly efficient. While MATLAB has traditionally been a considered the standard across neuroscience research labs, it was also well recognized that its performance was lacking for routines that aren't "vectorized", when compared to applications developed using lower-level languages like FORTRAN, C, and C++. Nevertheless, it remained in common use, and recent releases have added features that can drastically mitigate its performance issues, particularly through the development of a "Just-In-Time" compiler that automatically optimizes the deployment of computation accelerator resources for standard MATLAB functions. This feature enables code that performs repeated operations using for-loops or while-loops nearly as fast as equivalent code written in C. Additionally, code can be compiled into executable format using the Matlab Compiler toolbox, or used to generate equivalent C or C++ code using Matlab Coder.

Computational Resources for Processing Large Data Sets

Routines for extracting the activity in each cell from a collection of raw imaging data rely on an ability to simultaneous access many pixels separated over space and time (and consequently separated on disk). For long recording sessions, however, the size of the collection of stored image data grows dramatically. This substantial increase in the size of data easily exceeds the capacity of system memory in the typical workstation computer available to researchers. Thus, performing the necessary processing routines using standard programs is often unfeasible.

Another popular approach to this challenge is the migration of processing routines to a cluster-based system. In this way image data can be distributed across many interconnected computer nodes capable of performing all locally restricted image processing procedures in parallel, then passing data to other nodes in the cluster for tasks that rely on comparisons made across time. Access to clusters capable of performing in this way has historically been restricted to those working in large universities or other large organization, and the diversity of cluster types is sizeable, with clusters often having very particular configuration requirements for implementing data processing jobs efficiently. These issues would pose some difficulty to the use and shared development of software libraries for image processing routines, although the growth of "cloud computing" services such as Amazon's EC2 and the Google Compute Engine, and also collaborative computing facilities like the Massachusetts Green High-Performance Computing Center (http://www.mghpcc.org) mitigate many of these issues. Additionally, efforts to produce a standardized interface for accessing and distributing data, and for managing computing resources across diverse computing environments have

seen appreciable success. Apache's release of the open-source cluster computing framework, Hadoop, and a companion data-processing engine called Spark (http://spark.apache.org/), has encouraged a massive growth in collaborative development projects, a consequently increased the availability of robust shared libraries for data processing in a variety of applications. The Spark API can be accessed using the open-source programming Python, and also using other languages like Java, Scala, or R. One project specifically geared for image processing of neural imaging data is the Thunder library, a Spark package released by the Freeman lab and developed in collaboration with a number of other groups at Janelia farm and elsewhere.

Many applications will find the recent improvements in accessibility and standardization make cluster computing an attractive and worthwhile option for processing a very large set of reusable data. However, this strategy would impose harsh limitations for a neuroscientist with a project that is continuously generating new data, as the time required to transfer entire imaging data sets across the internet may be prohibitive. Unfortunately, storage on the cloud is not so unlimited that it can manage an accumulated collection of imaging data generated at anything near the rate that sCMOS cameras are capable of producing. This rate imbalance is a central motivating issue for Aim 2 this project, and is discussed in more detail below.

Methods and Approach

Image processing is performed offline using MATLAB software. The goal of this procedure is to reduce the raw image sequence to a collection of one-dimensional traces, where each trace indicates the fluorescence intensity of an individual neuron over time, and the collection approximates the distinct activity of each and every neuron in the microscope's field of view. We implement the process in 3 distinct stages as described below. The main novel contribution of this work is the efficient extension of segmented ROIs into the third dimension by clustering features of ROIs segmented separately in two dimensions. Online processing uses a similar approach, and the differences are explained in the next section.

Image Pre-processing: Contrast Enhancement and Motion Correction

Alignment of each frame in the image sequence with all other frames is essential to the methods we use in subsequent steps for identifying and tracking cells over time. Thus, the goal of the first stage is to correct for any misalignment caused by movement of the brain relative to the microscope and camera.

Many algorithms for estimating and correcting image displacement exist and are well described in the medical imaging literature. We elected to use phase-correlation to estimate the induced motion in each frame, as we found this method

to be highly stable, moderately accurate, and (most importantly) fast, especially when implemented in the frequency domain and using a decent graphics card.

Phase-correlation estimates the mean translational displacement between two frames, one being the template or "fixed" frame, and the other being the uncorrected or "moving" frame. In the spatial domain this is accomplished by computing the normalized cross-correlation, which implies a 2-dimensional convolution of large matrices. The equivalent operation in the frequency domain is a simple scalar dot-product of the discrete Fourier transforms of each image normalized by the square of the template, followed by the inverse Fourier transform. The intermediate result is the cross-correlation (or phase-correlation) matrix, which should have a peak in its center for correctly aligned images, or a peak near the center, the offset of which indicates the mean offset between the two images. This peak can be found with subpixel precision by interpolation to give a more accurate alignment, although at some moderate expense in computation time.

For the template image we used a moving average of previously aligned frames when processing frames sequentially, or alternatively a fixed mean of randomly sampled and sequentially aligned images from the entire set when processing files in parallel. The simplest way to perform this operation is to use the built-in MATLAB function normxcorr2, which makes optimization decisions based on image size and available hardware automatically. However, performance can be improved by tailoring the operation to your particular hardware and image size, i.e. using fft2 and ifft2 for large images and a good graphics card.

Region of Interest (ROI) identification & segmentation

The ROI detection process used an adaptive threshold on the z-score of pixel intensity to reduce each frame to binary 1's and 0's (logical true or false). These binary frames were then processed using morphological operations to find and label connected components within each frame. For example, beginning with a z-score threshold of 1.5, all pixels that were more than 1.5 standard deviations above their mean were reduced to 1 (true), and all others reduced to 0 (false). Pixels reduced to 1 were often pixels overlying a cell that was significantly brighter during that frame due to activation of GCaMP. This initial threshold was adjusted up or down based on the number of non-zero pixels detected with each threshold. This was done to prevent spurious motion-induced shifts of the image frame from producing ROIs along high contrast borders. All morphological operations were performed using built-in MATLAB functions from the Image Processing Toolbox, which have fast parallel versions if the operation is run on a graphics card (e.g. imclose, imopen, etc.). Furthermore, the connected-component labeling and region formation operations were run using built-in MATLAB functions bwconncomp, and regionprops. Connected components were stored in a custom class and termed "single-frame ROIs," and these were then passed to the 3rd stage of processing, which merges them into a "multi-frame

ROI" that represents the location and spatial distribution of each cell identified over the entire video.

Region of Interest (ROI) merging

The standard structure of region properties output by the MATLAB function regionprops (Area, BoundingBox, Centroid, etc.) are mimicked in a custom class called RegionOfInterest, where each field of the structure becomes a property of the custom class. We add additional properties for storing state information and data associated with each ROI, along with a number of methods for comparing, merging, manipulating, and visualizing the single-frame and multi-frame ROIs. The single-frame to multi-frame ROI merging procedure is essentially a clustering process that merges single-frame ROIs together using such criteria as the proximity of their centroids, as well as proximity of their bounding-box (upper-left and lower-right corners). Performing this operation quickly was highly dependent on pre-grouping ROIs based on centroid location in overlapping blocks of the image frame, as well as grouping by size. This enabled the clustering to be performed in parallel (across CPU cores) followed by a second iteration of clustering to deal with redundancy in overlapping regions.

Visualization

Once ROIs are established, all video data is reloaded and passed to a method in the *RegionOfInterest* class that extracts the 1-dimensional trace for each ROI representing the fluorescence intensity in that region over time. The ROIs and their traces can then be interactively visualized using another method in the *RegionOfInterest* class.

The *RegionOfInterest* class defines methods for rapid spatial comparison operations which can typically be viewed as an adjacency matrix using built-in image viewing commands. Visualization of the segmented cell overlay and 1D traces can be manipulated by assigning colors, removing ROIs, hiding ROIs, and more.

Predicting Activation State & Assessing Network Activity

The continuous signal intensity signals can be reduced to binary indicators of activity for each frame. This enables simplified and fast calculation of information theory measures, such as activation probability, joint and conditional probabilities, response entropy, mutual information, etc. The conversion from continuous to binary uses several abstractions of the signal applied to a Gaussian Mixture Model (GMM). The abstractions are calculated from the following:

1. Linear least-squares fits to moving windows with variable size to find slope of the line surrounding each point.

2. Skewness and Kurtosis of finite windows surrounding each data point.
3. Temporal difference of fluorescence intensity.

The gaussian mixture model employs all measures to infer periods of reliable distinct activation of neurons.

Parallel Processing

Many built-in MATLAB functions are implemented using efficient multi-threaded procedures, and these are used to the extent that they can be. However, for procedures that must operate on data in irregular formats (i.e. any format other than N-dimensional arrays of primitive data types), one also has the option of performing explicitly defined parallel operations by distributing data across multiple parallel processes, each with their own memory space. Below I give examples of how implementing in a multi-threaded fashion can substantially boost performance, and also an example of a situation where multi-threaded operations aren't possible without explicit calls for parallel distribution.

Standard elementwise operators like *plus* (+) and *times* (.*), as well as comparison operators like *equals* (==) and *less-than* (<) will be performed efficiently using as many processing cores as available when applied to large n-dimensional arrays of the same size. However, when operand sizes differ a simple call to the built-in operation will not work. For example, if we wish to subtract the average from each pixel over time from all frames in the series we can accomplish this with a call to MATLAB's *bsxfun* function, which stands for Binary-Singleton-eXpansion-FUNction, as shown below:

```
Fmeansub = bsxfun( @minus, F, mean(F,3) );
```

This operation passes a function handle as the first argument (denoted by the '@' symbol) indicating the operation to perform. It then passes the entire [IxJxK] array of image data as the second argument, and it's temporal mean with size [IxJx1] is calculated once and passed as the third. The function efficiently expands the mean argument as needed for fast distribution across parallel threads.

Managing Continuity

Data such as baseline frames and frames for alignment must be passed between parallel processors to maintain continuity between data divided temporally between processors. However, the efficient application of this approach was limited by the system memory and number of cores available, and was meant to be applied in a cluster environment.

Building the set of functions for offline processing enabled application to data already gathered, and also served as a framework that informed the necessary

procedures to be included in the online extension of this toolbox.

Computing Power and Connectivity

- Remote Clusters (AWS)
- Graphics Processing Units (NVIDIA GTX)
- Embedded Units (NVIDIA Tegra X2) 2. Well developed libraries
- ImageJ (so so)
- OpenCV (uses OpenCL)
- GStreamer (much better)
- OpenGL
- Shader

Image Processing

- Motion Correction
- Image Enhancement

Motion Correction Two approaches to find displacement

Spatially Homogeneous phase correlation

- aka normalized cross correlation - Feature Matching
- Detect features (i.e. corners) - Triangulate best

Image Enhancement

1. Contrast - Linear Scaling - Lookup Tables
2. Spatial and Temporal Filtering
3. Feature images - Gradients

Feature Extraction

1. Feature images (temporally independent)

- Gradients - Surface Curvature 2. Long Term Memory - Statistics
  - changes (single pixel)
- Mutual information changes (inter-pixel)

Acceleration and Optimization Procedures for Online Video Processing

Incremental Update of Statistics

central moments

```
function [m1,m2,m3,m4,fmin,fmax] = updateStatistics(x,m1,m2,m3,m4))
      n = n + 1;

      % GET PIXEL SAMPLE
      f = F(rowIdx,colIdx,k);

      % PRECOMPUTE & CACHE SOME VALUES FOR SPEED
      d = single(f) - m1;
      dk = d/n;
      dk2 = dk^2;
      s = d*dk*(n-1);

      % UPDATE CENTRAL MOMENTS
      m1 = m1 + dk;
      m4 = m4 + s*dk2*(n.^2-3*n+3) + 6*dk2*m2 - 4*dk*m3;
      m3 = m3 + s*dk*(n-2) - 3*dk*m2;
      m2 = m2 + s;

      % UPDATE MIN & MAX
      fmin = min(fmin, f);
      fmax = max(fmax, f);
end
```

Extract Features

```
function [dm1,dm2,dm3,dm4] = getStatisticUpdate(x,m1,m2,m3,m4)
      % COMPUTE DIFFERENTIAL UPDATE TO CENTRAL MOMENTS
      dm1 = dk;
      m1 = m1 + dm1;
      dm4 = s*dk2*(n^2-3*n+3) + 6*dk2*m2 - 4*dk*m3;
      dm3 = s*dk*(n-2) - 3*dk*m2;
      dm2 = s;
      m2 = m2 + dm2;
      % NORMALIZE BY VARIANCE & SAMPLE NUMBER -> CONVERSION TO dVar, dSkew, dKurt
      dm2 = dm2/max(1,n-1);
      dm3 = dm3*sqrt(max(1,n))/(m2^1.5);
      dm4 = dm4*n/(m2^2);
end
```

Simple Processing on GPU

```
[dm1,dm2,dm3,dm4] = arrayfun(@getStatisticUpdate(x,m1,m2,m3,m4)
[dm1,dm2,dm3,dm4] = arrayfun(@getStatisticUpdate(rowidx,colidx)
```

Alternative Libraries

- NVIDIA Performance Primitives
- OpenCV
- VLFeat
- OpenGL
- OpenCL
- OpenVX
- CLosedDoesNotExist (. . . ?)
- Shader Languages
- GLSL
- HLSL
- WebGL
- Halide
- – FFmpeg
- GStreamer

Choice of Interface

Procedural Framework: Pipes, Streams, & Graphs

Concurrency: Parallel = Performance?

Not always, no. While concurrent processing of independent tasks or sequentially arriving data elements will almost always increase performance, this is not always the case. At a lower instruction-level than we typically program, synchronous operations can often be optimized in ways that asynchronous operations cannot, typically through strategic register allocation, or by taking cache-hit performance). "Globally Asynchronous Locally Synchronous"

Scheduling

Adaptive

Choice of Operations

- What is the goal?
- Is it effective?
- Is the computation cost worth the result?
- Are there side-effects or artifacts?
- Can they be reliably controlled or accounted for?

Motion Correction

In our application, the goal of a motion correction operation is to artificially suppress translation of the brain tissue parallel to the image plane. *Phase-Correlation* (also referred to as *normalized cross-correlation*) has consistent performance across a range of image sources with varying spatial noise characteristics. However, a large non-uniform change from reference frames - such as occurs when cells with low baseline fluourescence are first activated - can cause drastic errors that must be recognized and corrected by a supervisory procedure. This can induce an undesirable, unpredicatable, and specifically inopportune latency Unfortunately in all the whole pipeline.

Unfortunately, “Globally Asynchronous Locally Synchronous”

as it’s In some experimental setups,

The phase correlation method of Motion Estimation - cost: 2-10 ms/frame - Frequently unstable (depending on video content)

Motion Compensation & Interpolation

- cost: 400-800 us/frame
- Requires infill with nearby or prior pixel values if frame size is to be maintained