# Chapter 1

# ch2_software

# Chapter 2

# Pipeline overview

## 2.1 phases

### 2.1.1 preprocessing

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

#### 2.1.1.1 time series analysis

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

#### 2.1.1.2 spatial signal deconvolution (unmixing)

### 2.1.2 signal localization and extraction

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

### 2.1.3 active signal analysis

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

#### 2.1.3.1 independent cellular activity measures

Quality Complexity Spatial Temporal

#### 2.1.3.2 network analysis

## 2.2 Computing cost:

boundary analysis for iOS and config

## 2.3 Storage cost:

compression and imposed boundaries # Image Processing

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

## 2.4 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.

### 2.4.1 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 multithreaded 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.

### 2.4.2 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.

## 2.5 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.

### 2.5.1 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.

### 2.5.2 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.

### 2.5.3 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.

### 2.5.4 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.

### 2.5.5 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.

### 2.5.6 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.

### 2.5.7 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.

#### 2.5.7.1 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

#### 2.5.7.2 Image Processing

- Motion Correction
- Image Enhancement

#### 2.5.7.3 Motion Correction Two approaches to find displacement

#### 2.5.7.4 Spatially Homogeneous phase correlation

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

#### 2.5.7.5 Image Enhancement

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

#### 2.5.7.6 Feature Extraction

1. Feature images (temporally independent)

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

#### 2.5.7.7 Acceleration and Optimization Procedures for Online Video Processing

#### 2.5.7.8 Incremental Update of Statistics

##### 2.5.7.8.1 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
```

#### 2.5.7.8.2 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, dKur
        dm2 = dm2/max(1,n-1);
        dm3 = dm3*sqrt(max(1,n))/(m2^1.5);
        dm4 = dm4*n/(m2^2);
    end
```

### 2.5.7.9 Simple Processing on GPU

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

#### 2.5.7.9.1 Alternative Libraries

- NVIDIA Performance Primitives
- OpenCV
- VLFeat

- OpenGL
- OpenCL
- OpenVX
- CLosedDoesNotExist (…?)
- Shader Languages
- GLSL
- HLSL
- WebGL
- Halide
- – FFmpeg
- GStreamer

### 2.5.8 Choice of Interface

#### 2.5.8.1 Procedural Framework: Pipes, Streams, & Graphs

##### 2.5.8.1.1 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”

##### 2.5.8.1.2 Scheduling

##### 2.5.8.1.3 Adaptive

#### 2.5.8.2 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?

#### 2.5.8.3 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)

##### 2.5.8.3.1 Motion Compensation & Interpolation

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

## 2.6 Continuous Online Video Processing

## 2.7 Introduction

The current generation of sCMOS cameras can capture full-frame resolution video at either 30 fps or 100 fps, depending on the data interface between camera and computer (USB3.0 or CameraLink). At 16-bits per pixel and 2048x2048 pixels, the maximum data rate for the USB3.0 camera is 240 MB/s. Imaging sessions typically last 30-minutes or less. However, pixels are typically binned down 2x2, and frame rate often reduced; processing speed and storage constraints are the primary motivation for doing so. The effect of doubling resolution on processing time when using the graphics card is nearly negligible, however. By identifying ROIs online and extracting the traces of neural activity allows us to discard acquired images and instead store the traces only, or feed them into an encoder for online analysis.

Graphics Processing Units were traditionally developed for the consumer gaming market. They are optimized for the process which involves translating a continuous stream of information into a two-dimensional image format for transfer to a computer monitor. In the context of gaming, the stream of information

received by a GPU describes the state of objects in a dynamic virtual environment, and is typically produced by a video game engine. These processors are highly optimized for this task. However, they are equally efficient at performing the same type of procedure in reverse – reducing a stream of images to structured streams of information about dynamic objects in the image – and thus are popular for video processing and computer vision applications.

Any GPU architecture will consist of a hierarchy of parallel processing elements. NVIDIA's CUDA architecture refers to the lowest level processing element as "CUDA Cores" and the highest level as "Symmetric Multiprocessors." Typically data is distributed across cores and multiprocessors by specifying a layout in C-code using different terminology, "threads" and "blocks." Blocks are then said to be organized in a "grid." Adapting traditional image processing or computer vision algorithms to run quickly on a GPU involves finding a way to distribute threads efficiently, ideally minimizing communication between blocks.

MATLAB makes processing data using the GPU seemingly trivial by overloading a large number of built in functions. Performance varies, however, and often the fastest way to implement a routine is by writing a kernel-type subfunction – written as if it operates on single (scalar) elements only – that can be called on all pixels at once, or all pixel-subscripts, which the function can then use to retrieve the pixel value at the given subscript. The kernel-type function is compiled into a CUDA kernel the first time it's called, then repeated calls call the kernel directly, having minimal overhead. Calls go through the arrayfun() function.

Data transfers between system memory and graphics memory is often the major bottle-neck. Therefore, this operation is best performed only once. However, once data is on the GPU, many complex operations can be performed to extract information from the image, all while staying under the processing-time limit imposed by the frame-rate of the camera sending the images.

## 2.8 Method and Approach

The entire procedure for processing images and extracting cell signals can be performed in substantially less time than most commonly available tools using the approach described in Aim 1, particularly the methods for restricting the spatial extent of pixel-association operations, and distributing operations across parallel processing cores using a Single Program Multiple Data (SPMD) archetype. However, the total time still exceeds that of the acquisition session. Inefficiency arises from the overhead involved with distributing data and passing information between separate parallel processes. Graphics cards, however execute in what's called Single Instruction Multiple Data (SIMD) fashion, to distribute computation across the thousands of processing cores.

The processing components are implemented using the MATLAB System-Object

framework, which allows for slightly faster performance through internal optimizations having to do with memory allocation. Most system objects, each representing one step in the serial processing and signal-extraction procedure, also have companion functions that implement the computation-heavy components of each algorithm using a pre-compiled CUDA kernel.

### 2.8.1 Benchmarking & General Performance

Built-in MATLAB functions that execute on the GPU can be profiled with benchmarking functions like *gputimeit()*, or with the *tic/toc* functions. When execution isn't fast enough, they need to be replaced with custom functions. The custom functions typically achieve the speed up necessary by enabling the operation to carried out on several frames at once. This reduces the over-head costs inposed for each function call by spreading it over several frames. This solution is not ideal, as it increases the latency of solutions, however does not preclude implementation in real-time system if the procedures are adapted to run on a real-time hybrid system-on-module like NVIDIA's Tegra X1, which should involve minimal effort once a standard set of successful procedures is realized. The current implementation tests the processing time of each stage of the process to ensure that the sum is less than the acquisition time for each frame dictated by the inverse of the frame-rate (30-50 milliseconds).

### 2.8.2 Buffered Operations

Combining frames for each operation can result in near linear speedup. For example, for the phase-correlation step required for motion correction, the FFT and IFFT are called on 16 image-frames at once, and the time take to accomplish is approximately the same as if the operation were called on 1 frame. This essentially leads to a 16x speedup, though the latency is also increased slightly. The best size to use is difficult to pre-determine, and typically must be measured for varying size 'chunks' using the benchmarking functions indicated above. The system objects manage the details necessary to allow buffered chunks of video to be passed to each stage without introducing artifacts at the temporal edges between chunks.

### 2.8.3 Image Pre-Processing & Motion Correction

Pre-processing is implemented as with the offline procedure, with a few changes. Images are aligned in chunks, and they are aligned sequentially to two templates. One template is the most recent stable frame from the preceding chunk. The other is a recursively temporal-low-pass filtered image that mitigates slow drifts. Aligning to the first template is usually more stable as the brightness of cells in

the recent image will be more similar to those in the current chunk than will be the brightness of cells in the slow-moving average.

The displacement of each frame is found to sub-pixel precision, then used with a custom bicubic resampling kernel that replaces any pixels at the edges with images from the moving average.

### 2.8.4 Sequential Statistics

A number of statistics for each pixel are updated online and can be used for normalization and segmentation procedures later in the process. These include the minimum and maximum pixel intensity, and the first four central moments, which are easily converted to the mean, variance, skewness, and kurtosis. The formulas for making these calculations are given below, and are performed in a highly efficient manner as data are kept local to each processing core, and repeat computations are minimized.

```
n = n + 1;
f = F(rowIdx,colIdx,k);
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);
```

Furthermore, the value used to update each central moment at each point in time can be used as a measure of change in the distribution of each pixel caused by the current pixel intensity, as explained next.

### 2.8.5 Non-Stationarity & Differential Moments

Stationary refers to the property of a signal such that its statistics do not vary over time, i.e. its distribution is stable. Neural signals tend to specifically *not* have this property, in contrast to other measurable components such as those contributed by physiologic noise (heart-rate, respirations, etc.). Thus, by analyzing the evolution of statistical measures calculated for each pixel as frames are added in sequence gives a highly sensitive indicator of neural activity. This

is done using a routine analogous to that for updating central moments given above, except the values returned are not only the updated moment, but also the updating component – essentially the partial derivative with respect to time. This is illustrated below, including the normalization functions which convert the partial-moment values to their variance, skewness, and kurtosis analogues:

```
% 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);
```

These functions run on images representing the image intensity, and also on images taken from sequential differences indicating the temporal derivative of image intensity. The combination of outputs from these operations indicate both when image intensities are significantly high relative to past distribution, and also when intensities are changing significantly faster than learned from their past distribution.

### 2.8.6 Surface Classification: Peaks, Edges, Curvature

Edge-finding methods are employed for establishing boundaries between cells, and first and second-order gradients are used to compute local measures of curvature from an eigenvalue decomposition of the local Hessian matrix. I won't go into detail, as the utility of these procedure in the most recent implementation has been lost, but nevertheless, the operation is optimized and ready to be plugged back in when further development calls for better accuracy informing cell-segmentation, or when a faster or more accurate motion-correction algorithm is called for.

### 2.8.7 Online Cell Segmentation & Tracking

Cells are segmented by first running sequential statistics on the properties of identifiable regions on a pixel-wise basis. That is, as regions are identified in a method similar to that used offline in Aim 1, the region-properties are calculated (Centroid, Bounding-Box, etc.) and statistics for these properties are updated at each pixel covered by a proposed region. After sufficient evidence has gathered, Seeds are generated by finding the local peak of a seed-probability function that optimizes each pixel's proximity to a region centroid, and distance from any boundary. Regions are grown from these seed regions, and registered in a hierarchy that allows for co-labeling of cellular and sub-cellular components. Newly identified regions occur as new seeds, where as seeds overlapping with old regions are used to identify sub-regions, or to track regions over time.

### 2.8.8 Signal Extraction from Subcellular Compartments

I also have functions for the extraction of normalized Pointwise-Mutual-Information (nPMI), which can operate on a pixel-to-pixel basis or on a region-to-pixel basis. This operation accumulates mutually informative changes in all pixels in the maximal bounding-box (e.g. 64x64 pixels) surrounding each identified regions centroid. The weights given by this function can take on values between -1 and 1, and can be used to inform any reduction operations to follow. Additionally, spatial moments can indicate the subcellular distribution of activity across the identified region. In this context, the first spatial moment $M_{00}$ indicates the mean signal intensity.

### 2.8.9 User Interface for Parameter Tuning

Some system-objects also incorporate a user interface to aid in parameter selection for tuning. ## Compression: as a Tool, a Goal, as an Explanation

This section describes general compression algorithms as well as compression algorithms specifically tailored to application in video, accomplished by searching for both temporal and spatial redunancy.

---

## 2.9 Data Management

Essential to the long-term success of any operation is an effective strategy for the management I've all data coming in and out of the system. Especially important is the insured facility for continued access and translation of data as systems evolve or as data is passed to other infrastructure. Strategies that meet the need

of data generators and users often if it's somewhere on a spectrum of strictly defined data structures that minimize errors but are strict and difficult to evolve, versus those that require verbose documentation which hinders performance but are adaptable.

The new structures that scientists typically use are on the flexible and self-describing Spectrum and, these include two files for large image formats, fits files, cifs, Matlab structure definitions, which are an implementation of hdf5 file structures. And similarly standard JavaScript object notation in parentheses Json structures. At the other end exist Raw structures compression codex and other processing procedures, and cannot be accessed reliably without thorough knowledge of the source code for napi, or extensive packing and guessing. In the middle exist and number of serialization formats such as Google protobuf format, Thrift, Avril, Captain T, ND growing number of other formats that try to offer flexibility in evolution over time as well as variable levels of self-description and performance. These formats are often to find in simple files and are used by code generators 2 produce computational e efficient code for serialization and D serialization without losing cross platform compatibility or risking rapid deprecation as the needs for the format evolve with the needs of the users. Types can be added to provide new functionality without damaging backwards compatibility. Additionally this model is off in accompanied by a translator herbal component to JavaScript object notation which makes the format ultimately universal in the case of a desperate need for compatibility when performance no longer matters

Tags: data, directory, format, programming, serialization ## Data Scaling

This section describes the reality of how data scales as more and more sensors are added. Typically we humans think linearly, but the ramifications of increasing something like sensor size is often exponential. Furthermore, many operations we perform have costs that scale exponentially, and in my humble opinion are not even worth attempting if any such procedure must applied continuously on a data=stream. ## Distributed Dataflow and Streaming

### 2.9.1 Language: Is MATLAB the best tool for this job?

- Standard language in many engineering programs
- Proprietary
- Performance
- Compatibility
- Need for a "SandBox"
- Lacks modularity
-

### 2.9.2 Alternatives Languages

- Python
- C/C++
- Java/Scala
- Javascript/Node
- GO, Haskell

### 2.9.3 Databases

### 2.9.4 Big Data

- not exactly...
- disparate simple queries across

### 2.9.5 Computational Models

- Dataflow Processing
- Actors model
- Petri Nets
- Graph Processing - i.e. Tensorflow

---

# Chapter 3

# Information and Informativity

This section presents a background in information theory in the context of behavior sensors and video streams. I further elaborate on what types of information neuroscientists and healthcare providers would hope to extract from these data, focusing on a larger picture of connecting the process of extraction directly the ultimate application.

Whereas previous attempts to process data focus largely on putting it in an intermediate state that is workable with current limitations on computation, visualization, and developer interaction with the data-streams, we now have an opportunity to skip intermediate states and develop with applications in mind.

Transition into compression by showing that previous attempts were essentially compression algorithms developed specifically for the data and preconceived notions about results, but that we can also apply very well developed general compression algorithms, with slight modifications to extract features such as localized bitrate that quantify (in an unbiased way) information content in an selectable region of a video-frame as it changes over time.

---