Documentation notes: Sparse Emulation of Unknown Dictionary Elements (SEUDO)

Jeff Gauthier and Adam Charles*

Last Updated: January 13, 2021

Contents

*Thanks to all manuscript co-authors for help with improving this documentation

1 Copyright and Disclaimer

2 Overview

This software lets you perform several related analyses on calcium imaging data:

- look through transients and classify them manually
- automatically classify transients
- visualize the contamination rate in many cells (FaLCon plots)
- filter out contamination (SEUDO)

All features are implemented in MATLAB.

To explore how the functions and GUIs work, we recommend running sections of the `demo.m` file, which uses an example dataset, before working with your own data. The rest of this document provides an overview of how to load data and use each type of functionality. There are more usage examples in `examples.m`, and the full details are described in the help for each function.

As a general note, this code is in active development and we welcome your feedback! If you have bug reports, feature requests, or would like more thorough documentation for particular functionality, please email seudoalgorithm at gmail dot com.

Installation and compatibility

To install SEUDO, add the directory "seudoCode" to the Matlab path. The demo dataset can be downloaded from the two links provided in the `demo.m` file. Full functionality requires Matlab version 2016b or later. Software was tested in versions 2015b, 2016b, 2017b on Mac OS 10.12, and version 2018b on Linux.

3 Required packages

SEUDO is written in MATLAB, with nearly all functions being self-contained. The one exception is Templates for First-Order Conic Solvers (TFOCS) package [1]. This flexible package enables robust solving a wide range of convex optimization programs and is scalable to larger problem sizes. SEUDO uses TFOCS to solve the partial, non-negative LASSO

$$\{\widehat{a}, \widehat{c}\} = \arg\min_{a,c\geq 0} \|y - \Phi a - Wc\| + \lambda\|c\|_1,$$

where y is the calcium data, Φ are the spatial profiles, W is the structured noise representation dictionary, a is the activity of the cell profiles, and c is the sparse representation of the structured noise. TFOCS can be downloaded from http://cvxr.com/tfocs/ and has extensive documentation at http://cvxr.com/tfocs/doc/.

4 Loading the data

All functions are implemented in MATLAB and rely on an object class called `seudo`. A given seudo object is defined by two things:

1. a motion-corrected dF/F movie
2. a set of source profiles (a.k.a. cell shapes, spatial footprints, pixel weights, active components, ROIs, etc) generated by a cell-finding algorithm, such as CNMF

The simplest way to provide the movie is a 3D matrix of size $[Y \times X \times N_{frames}]$, though other options are available (see `examples.m`). The profiles should be a 3D matrix of size $[Y \times X \times N_{sources}]$. The following command loads them into a seudo object named `se`:

```
se = seudo(M,P);
```

where M is the movie matrix and P is the profile matrix. All subsequent examples will assume the seudo object is named "se", though of course you can choose any name, or even create an array of seudo objects to load several datasets at once:

```
se(1) = seudo(M1,P1);
se(2) = seudo(M2,P2);
...
```

Most cell finding algorithms also provide a set of time courses for each source. These can be provided with an optional parameter `timeCourses`, like this:

```
se = seudo(M,P,'timeCourses',T);
```

where T is a matrix of size $[N_{frames} \times N_{sources}]$. If time courses are not provided, least squares time courses will be computed automatically. Either way, these are considered the "default" time courses, and they live in the field `se.tcDefault.tc`. The overall goal of this software is to evaluate and improve how well these time courses match the original movie.

For most functionality, discrete transients need to be identified in each time course. This is done automatically when the data is first loaded, though you can subsequently supply custom transient definitions like this:

```
se.computeTransientInfo('default','transientFrames',TF)
```

where `TF` is a logical a matrix of size $[N_{frames} \times N_{sources}]$ indicating which frames have transients for each source. This function computes summary information about each transient (such as the transient profile, the starting and ending frame, etc) and stores it in the seudo object to be used later.

Most of the functions described below involve making changes to the seudo object, such as manually classifying transients, or computing new time courses with SEUDO. These changes can be saved to disk and reloaded for use in subsequent MATLAB sessions. The files are specified at loading time like this:

```
se = seudo(M,P,'tcDefault','path/to/tcDefault.mat','tcSeudo','/path/to/tcSeudo.mat');
```

Below we'll cover how to save data to these mat files. An advantage of loading information from disk is that you won't need to load the original movie again (unless you want to run a function that requires access to the movie, such as SEUDO).

One technical note about seudo objects: they inherit from the `handle` object class. This means that any manual annotations entered in a GUI are immediately stored in the seudo variable. It also means that seudo objects cannot be duplicated (for example, the command `se2 = se` will just create a new name for the seudo object, but the object itself wont be duplicated; any changes made to `se2` will also be applied to `se`). Instead, the way to perform multiple analyses on the same dataset is to create duplicate subfields of the seudo object, like this:

```
se.tcDefault(2) = se.tcDefault(1);
```

When you run an analysis, you can specify which set of time courses to use like this:

```
se.autoClassifyTransients({'default',2},...)
```

You can think of the seudo object as storing several kinds of information in a hierarchy. The diagram in Figure 1 depicts this hierarchy graphically by representing data dependencies. For example, transient definitions depend on what the time courses are, so transient definitions fall below time courses in the hierarchy. Similarly, transient definitions are required to have a transient classification, so they are placed above transient classification. The figure also shows how to access these data programatically.

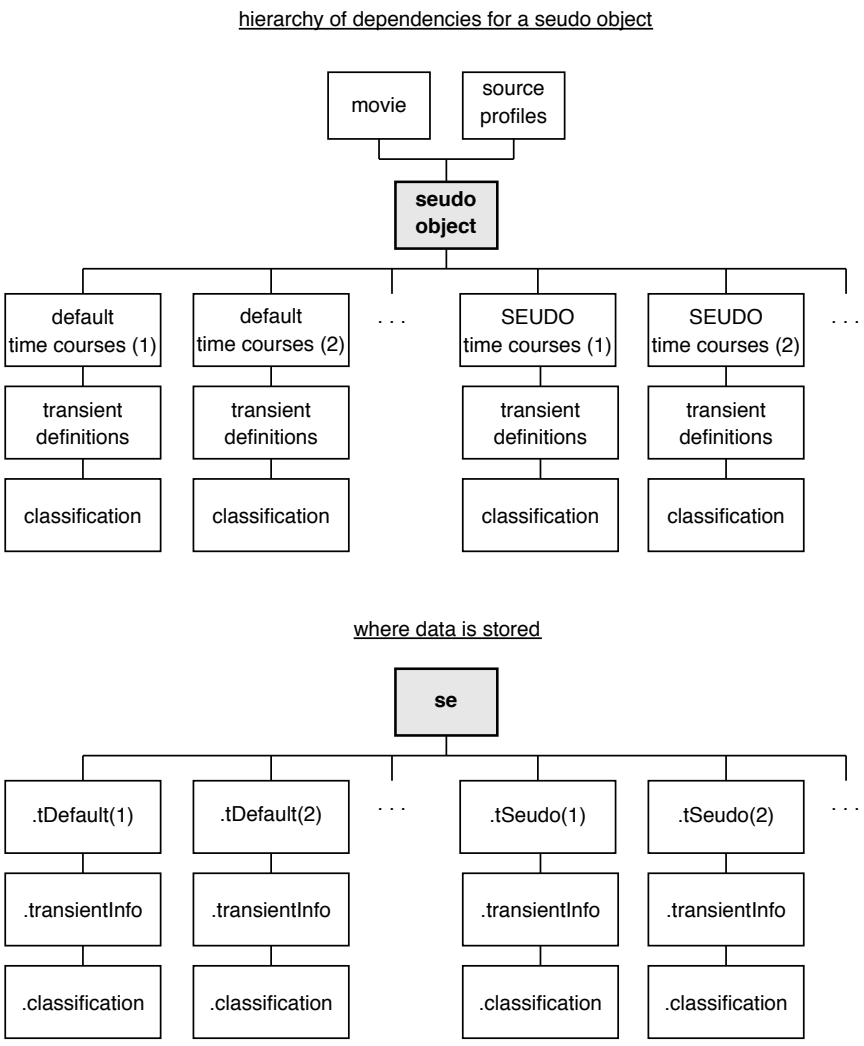


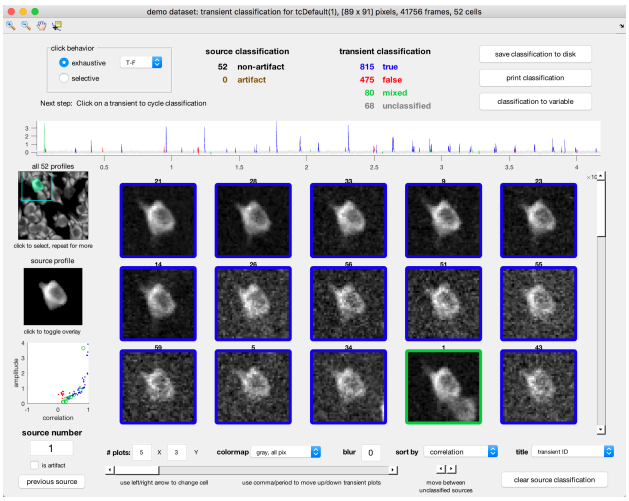
Figure 1: SEUDO object dependencies depicted graphically.

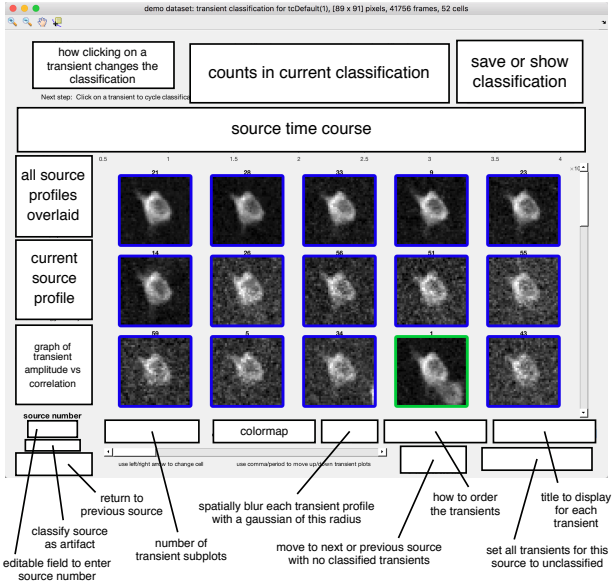
5 Manual classification

The fastest way to get a sense of how much contamination is in your data is to launch the transient classification GUI with this command:

```
se.classifyTransients
```

The interface lets you browse through the dataset by showing transient profiles for each source, and interactive elements let you quickly navigate to other sources.[1]



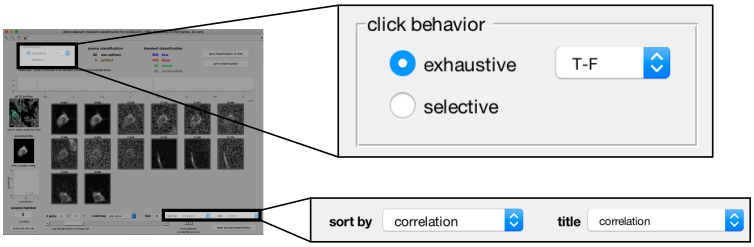


There are several ways to get to another source: clicking the scroll bar, using the arrow keys (unless focus is lost), entering a new value into the "source number" field, or clicking on the source you want in the image of all source profiles. If multiple profiles overlap, repeatedly click on the

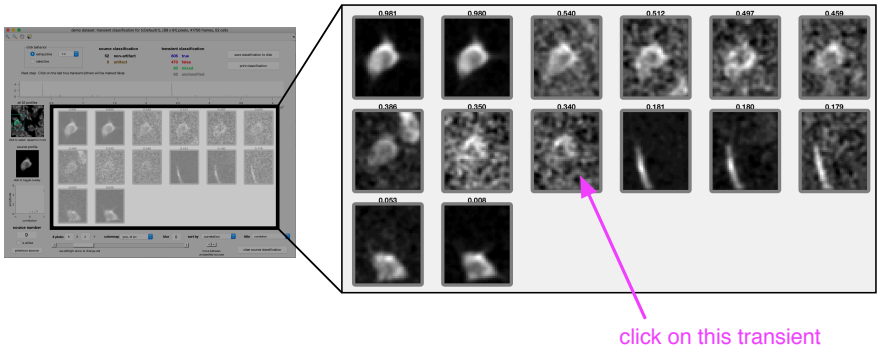
[1] Across different computing platforms, GUI appearance may change, and in particular some text labels might be obscured. Resizing the GUI to make it larger typically allows all labels to display correctly.

overlapping region to cycle through them. If you want to switch back and forth between two sources, you can click the "previous source" button to jump back.

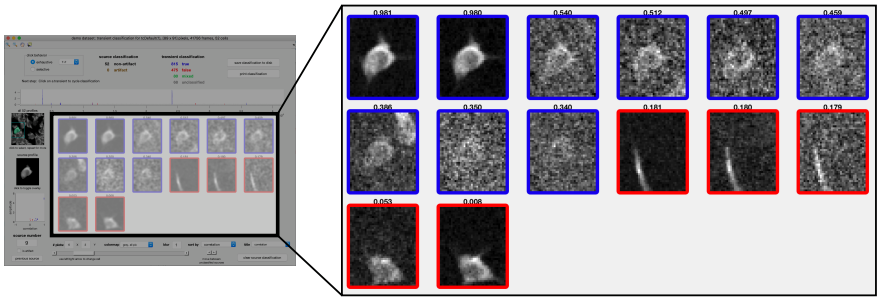
The GUI also allows you to manually classify transients as true, false, or a mixture of true and false. And you can flag entire sources as being artifactual. Manual classification can be time-consuming, so we have developed a trick to very rapidly classify all transients for one cell. Because true transients tend to have higher correlation with the source profile than false transients do, sorting transients by their correlation usually sorts them in a true to false gradient. To do this, set the "sort by" popup menu to "correlation". You can also display the actual correlation value for each transient by setting the "title" popup to "correlation".



For the following source, it looks like the first 9 transients are true, and the last 5 are false.



To immediately apply this classification scheme, be sure that click behavior is set to "exhaustive", and the popup says "T-F". Then click on the 9th transient. Voila! All 14 transients are classified with a single click.



From here, you can change the classification of individual transients by clicking on them, or reset all transients from this source to unclassified with the button "clear source classification".

Here's an overview of how clicking behavior works. If all transients for a source are unclassified, the "click behavior" radio button determines what a click does. If set to "exhaustive", all transients from the first one to the clicked one are labeled true, and the rest are labeled false when the popup menu is "T-F", or unclassified when "T-U". If "selective" is chosen, then clicking on a transient just changes its classification. After at least one transient from this source is classified, clicking on a transient changes only its classification, no matter what "click behavior" is set to.

The classification GUI provides several options for visualizing transient profiles and comparing them to the source profiles. The most essential is establishing the number of plots to display, which can be optimized for your screen size and viewing distance using the "# plots" fields. Other changes to visualization can be activated using keyboard shortcuts:

C - cycle through colormaps (three options)

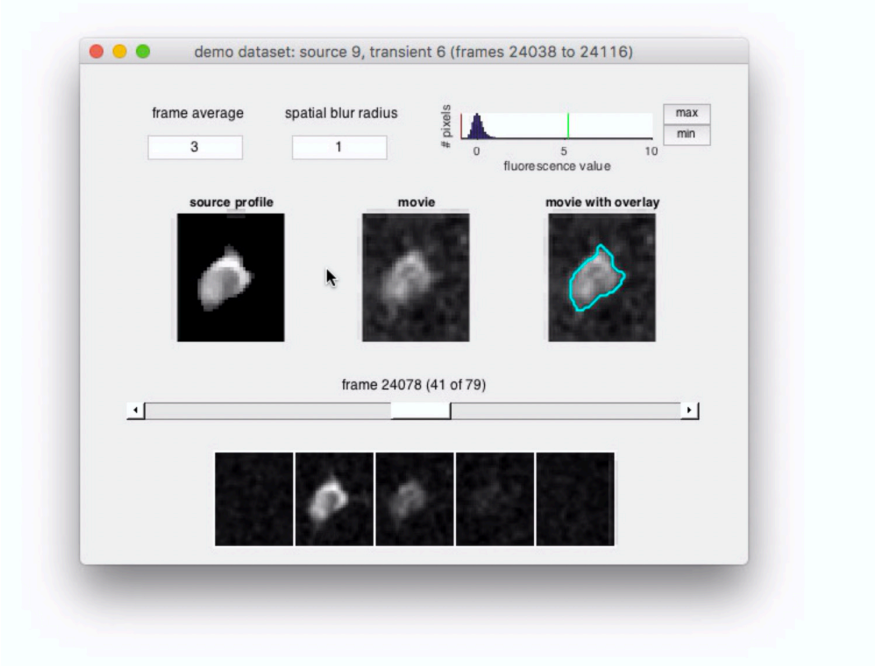
O or V - overlay source profile on each transient profile

1, 2, 3, ... 9 - apply Gaussian blurring to transient profiles to see structure obscured by noise. The radius in pixels is equal to the number typed.

0 (zero) or `(backtick) - turn off Gaussian blurring

If it seems like a source profile arose from an artifact, such as two cells being fused or a motion artifact, you can flag it by clicking the "artifact" checkbox (keyboard shortcut A). In subsequent analyses that use transient classification, sources flagged as artifacts are typically ignored.

You can also view the raw movie frames from the transient by right-clicking or control-clicking on a transient profile. This opens a mini GUI to inspect the transient. As in the main GUI, there are



options to turn on frame averaging and spatial blur, which are especially important when looking at individual frames. You can also adjust the min and max brightness levels. The histogram in the

upper right shows the pixel values in the movie being displayed, with red and green lines indicating which pixel values correspond to black and white, respectively. Clicking on the "max" or "min" button brings up a crosshairs, and clicking in the histogram will change the value.

If you want to classify all sources in a dataset, you might encounter some tricky cases you want to postpone working on until later. To return to them, use the the arrow buttons labeled "move between unclassified sources".

When you classify transients in the GUI, your entries are automatically stored in the seudo object variable in this field:

```
se.tcDefault.transientInfo.classification
```

Accessing them there can be cumbersome, so there are several ways to easily extract the classification. To display the current classification in the MATLAB command window, click "print classification" (but beware that this printout can take up a lot of space). To store the classification in a variable in the base workspace, click "put classification in variable". You can also save the classification in a .mat file to load in future sessions with "save classification to disk".

If you want to clear the classification for an entire dataset, which will reset all sources and transients to unclassified, you cannot do it in the GUI, so don't worry about an accidental click erasing all your work! Instead, run this command:

```
se.clearClassification
```

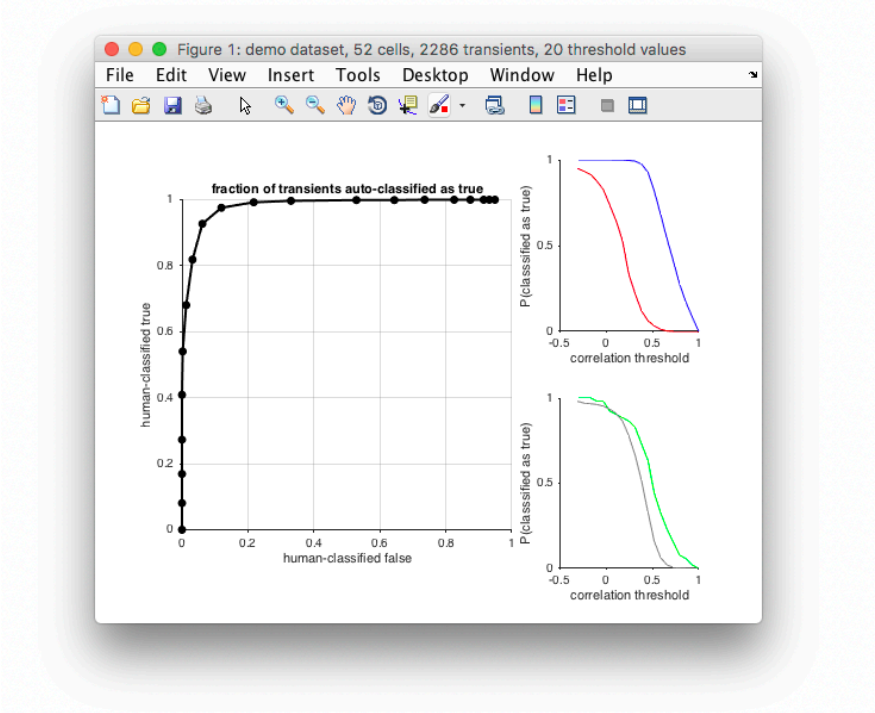
6 Automated classification

This command applies an automated classification to all transients from all sources:

```
se.optimizeAutoClassification(...)
```

Using automated classification can save a lot of time, but first you need to figure out the right parameters. Heres one way to do that efficiently: classify transients for several sources in the classification GUI, then tune parameters of automatic classification to see which ones match best.

There are many ways you could systematically vary parameter sets, as described in the function documentation. But the simplest way is to simply provide several different values for threshold. You can perform automatic classification and immediately see the results with the function `optimizeAutoClassification`. For example, the command `se.optimizeAutoClassification` will apply the default number of threshold values (20) evenly spaced between -1 and 1, and then generate a figure showing how transients were automatically classified.



You can also specify threshold values, like this:

```
se.optimizeAutoClassification('threshValues',[.4 .5 .6 .7])
```

After you pick the best one, you can classify all transients with this command:

```
se.autoClassifyTransients('default','overwrite',true, < optimal parameters >)
```

This will overwrite any manual classification that is currently stored and replace it with the automated classification.

In subsequent sections, it will be assumed that the current classification is "ground truth" provided by an expert human, and these values will be used to evaluate and calibrate other automated algorithms.

7 Visualizing automated classification (FaLCon plots)

To view the overall timecourse accuracy for many cells at a glance, we facilitate the plotting of FaLCon plots with the following command:

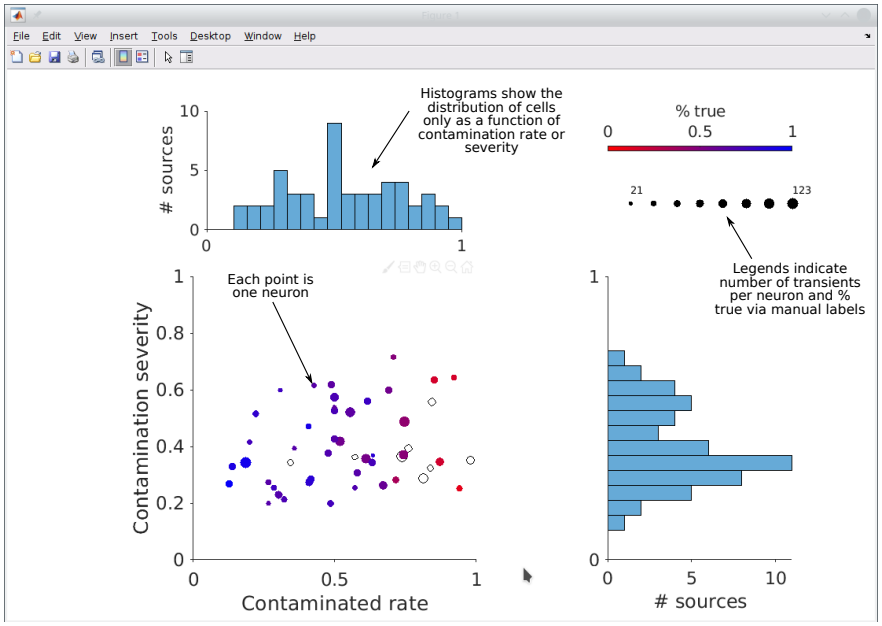
```
se.plotFalcon([],[],'corrThresh',0.5)
```

These plots

To view the overall timecourse accuracy for many cells at a glance, we facilitate the plotting of FaLCon plots with the following command:

```
se.plotFalcon([],[],'corrThresh',0.5)
```

which plots the falcon plot as



These plots allow for population assessment of both the rate of contamination (i.e., the fraction of transients automatically classified as 'false') and the severity of contamination (how much residual was unexplained by the cell shape for the transients labeled 'false').

FaLCon plots using different definitions of contamination rate and severity (i.e., based on the Ljung-Box quantile test) and using different threshold values via the arguments to `plotFalcon` (see in-code documentation).

8 Identifying parameters for SEUDO

To remove false transients, we have a developed a method called Sparse Estimation of Unknown Dictionary Objects, or SEUDO. Because SEUDO explicitly models contamination, it can provide more accurate time courses than is possible by defining transients and classifying them as true or false. Before applying SEUDO, it is important to find parameters that are adapted to your dataset and your particular application. For example, you might decide you need to remove all traces of contamination even if that requires sacrificing some true activity. Alternatively, you might want to keep all true transients, even if that means including a few false positives.

Parameters for SEUDO can be rapidly evaluated and optimized using several methods. The first step is to manually classify transients from several sources, since these will be used as a baseline for judging performance. The following sections will assume you have a set of expert-classified transients to work with. The more you provide, the better the results will generalize, but the longer it will take to compute results. We recommend looking through several sources to find a set of one or two dozen representative transients that can be confidently classified as true or false.

Before discussing the steps to test parameters, here is a brief description of each parameter and how to choose its value.

`lambdaBlob` - inverse of expected variance of contaminating activity (λ in the formal derivation). We will treat this as the key parameter to bias SEUDO towards erring on the side of removing contamination vs preserving true activity. Larger values of λ yield fewer blobs, and vice verse.

`sigma2` - expected noise variance (σ^2 in the formal derivation). This can be approximately estimated as the variance of a pixel over time in the ΔF/F movie. We recommend setting `sigma2` to a fixed value and primarily adjusting `lambdaBlob` to tune the algorithm.

`p` - probability of contamination in a given frame (p in the formal derivation). This typically has little to no effect on the results unless it shifts to being closer to 1 than 0.

`blobRadius` - radius of the Gaussian kernels. It can be approximated by inspecting movie frames to see the spatial scale of active sources. Sometimes even larger values work better, since this makes the blobs less similar to speckle noise in the movie.

`dsTime` - length of temporal downsampling, specified as number of frames. If greater than one, each movie frame in the analysis will be the average of `dsTime` original frames (from `dsTime` - 1 frames ago to the current frame). Greater downsampling improves SNR, though at the expense of time resolution.

`lambdaProf` - sparsity multiplication value for fitting the amplitude of source profiles, primarily serves to avoid small noise fluctuations from seeming like transients. Higher values create more sparsity. If set to 0 (the default), this sparsity constraint is not applied.

`padSpace` - SEUDO is performed on each source separately in a rectangular window around the source profile. `padSpace` indicates how many pixels beyond the minimum rectangle to pad the window on each side. For example, if the source profile fits into a 5×5 window, and `padSpace` is set to 3, SEUDO will operate on an 11×11 window centered on the source profile.

`minPixForInclusion` - When analyzing a given source, SEUDO fits each movie frame as a weighted sum of that source profile, small Gaussian kernels, plus any other source profiles that are located nearby. "Nearby" is defined by having at least `minPixForInclusion` nonzero pixels in the analysis window. We often get better results by setting this value to infinity so that no other source profiles are included.

`useCOM` - Instead of defining the analysis window as the minimum rectangle plus padding, you can also defined it as a square region centered on the source profile's center of mass. To do this, set `useCOM` to true (default is false), and specify the radius with `padSpace`.

To see the default values of all parameters, use this command:

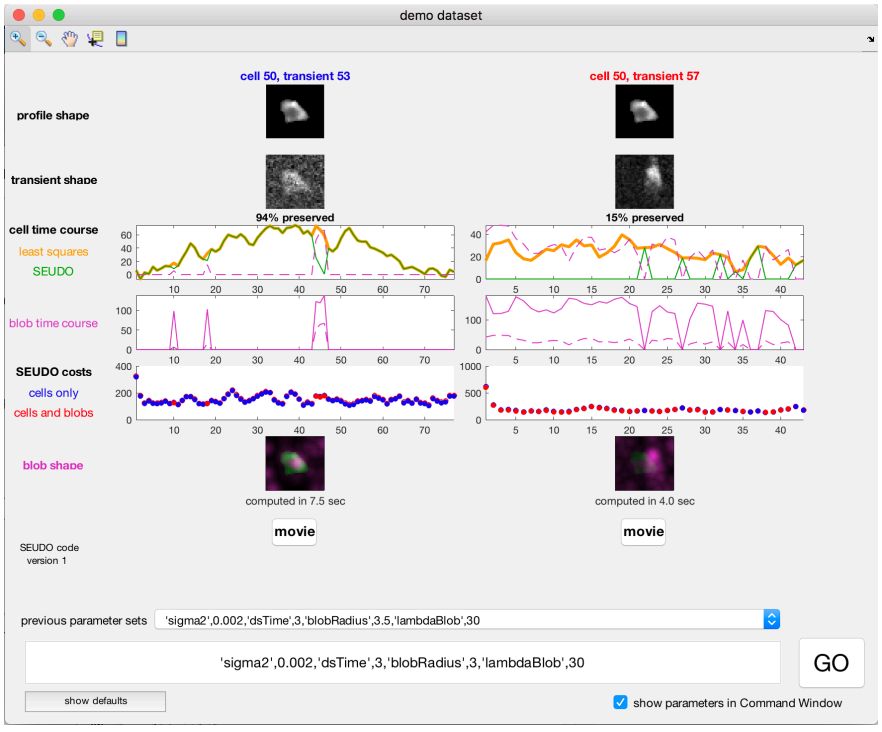
```
se.estimateTimeCoursesWithSEUDO('showDefaults',true)
```

You can try out different parameters for SEUDO with a simple GUI:

```
se.quickSEUDO(transSpec)
```

where `transSpec` specifies which transients to use in this format: $[N_{transients} \times 3]$ matrix, one row per transient, where the three columns are source ID, transient ID within that source, and the classification of the transient (`true` or `false`). The GUI allows you to type in a string of parameters and immediately visualize the results. It typically takes 5-10 seconds to compute SEUDO on one transient, so we recommend providing only 2-4 transients to get very rapid feedback.

The GUI shows the final time course estimated by SEUDO, as well as several intermediate results that can help you optimize parameters. We recommend altering parameters one at a time. The goal is to find a regime where the true transient is preserved and the false transient is removed.



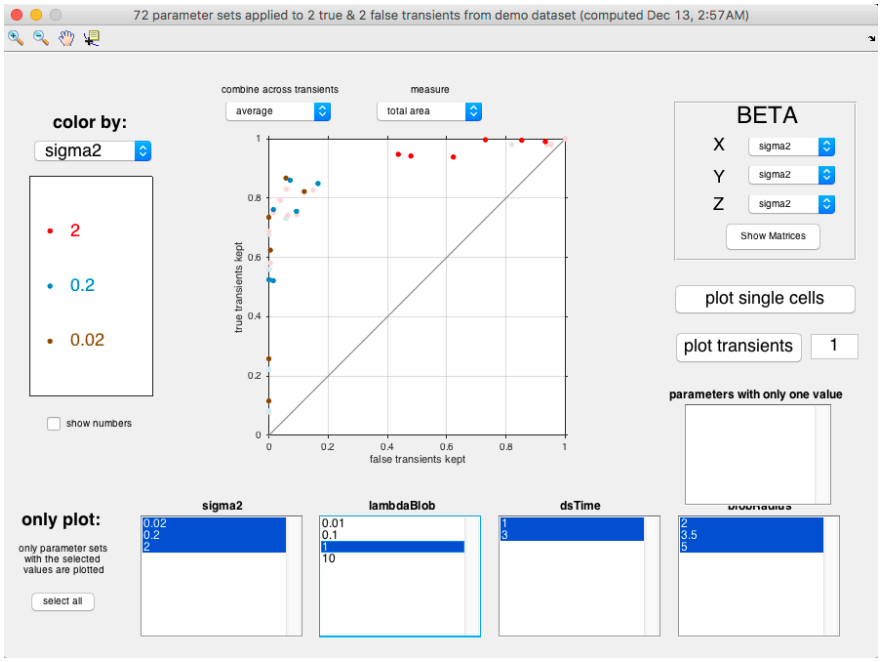
Clicking the "movie" button will launch the same viewer described above.

Once approximate parameters are chosen, a more thorough search of parameter space can be performed with the function `paramSearch`. This function takes two inputs: a handful of true and false transients to analyze, and a specification of the parameter space to be search. For each set of parameters, SEUDO will be applied to the transients. Depending on the number of transients and parameter sets, this search can take several hours, so we recommend starting with a small number of transients and relatively sparse sampling of parameter space.

After executing the parameter search, you can view the results in a GUI with this command:

```
seudoReviewParamSearch(psResults)
```

where `psResults` is the struct of results returned by `seudoReviewParamSearch`. The central plot in this GUI shows the performance of each parameter set as a single point. The axes are the fraction of transient area that is preserved for true (vertical) or false (horizontal) transients. An ideal parameter set would lie in the upper left corner, i.e. preserving 100% of true transients and 0% of false transients.



To identify how different parameters interact, you can color points based on a single parameter, or show only parameter sets that have a particular value for one parameter. Because this GUI is currently in beta, the full feature documentation is not yet complete.

9 Running SEUDO

The function `estimateTimeCoursesWithSEUDO` is used to perform SEUDO. When using this function, it is convenient to first specify parameters in a cell array:

```
seudoParams = {'dsTime',3,'sigma2',0.0020,'lambdaBlob',10,'blobRadius',3,...
                                          'padSpace',5,'saveBlobTimeCourse',1};
```

Because analysis of each source is independent, SEUDO can be parallelized by executing it on subsets of cells separately. For example, this command applies SEUDO only to sources 17 and 18:

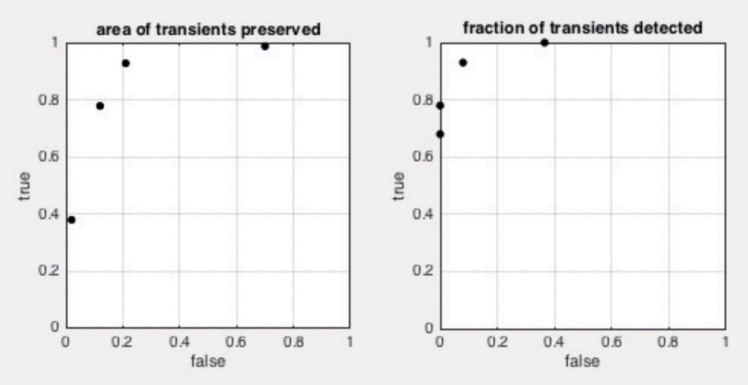
```
se.estimateTimeCoursesWithSEUDO(seudoParams{:},'whichCells',[17 18]);
```

The time courses computed by SEUDO can be accessed in `se.tcSeudo.tc`. If SEUDO is run multiple times, for example to try different parameter combinations, each run is stored in a new entry. These are accessed as `se.tcSeudo(1).tc, se.tcSeudo(2).tc`, etc.

One simple way to make SEUDO more efficient is to only apply it to frames with activity. In datasets where activity is sparse, this can significantly reduce processing time. The command `parallelSEUDO` runs SEUDO exclusively on frames with a transient, as defined in `se.tcDefault.transientInfo`. It also runs each cell in its own parallel process. The first argument to `parallelSEUDO` is the list of cells to analyze (providing an empty array defaults to all cells), and subsequent arguments are the standard SEUDO parameters.

```
se.parallelSEUDO( [] , seudoParams{:})
```

The code also provides a simple visualization of how well SEUDO performed. The command `computeSeudoROC` generates an ROC plot, using all parameters combinations of SEUDO that have been run. Note that this analysis requires the transients to be defined and classified, as described above.



A common workflow might be to use `quickSEUDO` to find a set of parameters with reasonable effectiveness, then try several values of lambda to sweep out an ROC curve.

```
% choose cells
whichCells = 10:15;

% choose lambda values
lambdaValues = [0.01 0.1 1 10 100];

% apply SEUDO using each value
```

```
for ll = 1:length(lambdaValues)
   seudoParams = {'dsTime',3,'sigma2',0.0020,'lambdaBlob',lambdaValues(ll),'blobRadius',3};
   se.parallelSEUDO(whichCells, seudoParams{:})
end

% visualize results
resultsROC = se.computeSeudoROC;
```

Together, these tools provide a straightforward path for optimizing SEUDO and exploring its effectiveness on each dataset.

References

[1] Stephen R Becker, Emmanuel J Candès, and Michael C Grant. Templates for convex cone problems with applications to sparse signal recovery. *Mathematical programming computation*, 3(3):165, 2011.