# SID-MANUAL

The rough Outline of the SID-algorithm



## Prerequisites

* MATLAB 2017 or newer
* Point Spread Function struct: see lfrecon\_vsc

# Getting started

You have generated a Light Field Microscope (LFM) Video of neuronal tissue labeled with fluorescent Calcium Indicator. If the Video has not yet been converted into a series of *tif*-images, with filenames ending in the frame number, with leading zeros, of the specific image in the video, do this before you start. Transfer the *tif*-files in your desired Input folder and start generating the Input -struct. Fill in the address of the Input- folder as ***Input.LFM\_folder***.

Download *LFdisplay* from <http://graphics.stanford.edu/software/LFDisplay/> and determine x\_offset, y\_offset and dx (SID only considers the case where dx=dy).   
Set ***Input.rectify*** true, otherwise if the images have already been rectified set it to false. In the former case also fill in the parameters as ***Input.x\_offset***, ***Input.y\_offset*** and ***Input.dx***.

Set the field ***Input. psf\_filename\_ballistic*** to the address of the file containing the point-spread-function generated according to lfrecon\_vsc.

Next specify the frames that you wish to include into the SID-optimization procedure. Take into account that SID can only detect neurons, that are active in some of those frames and inactive in others. The number of frames taken into account are prohibitive for the SID algorithm and they should not exceed 3000. Before you specify the frames you should also take a look at the raw movie. A good way to do this is looking at the average signal (mean over all pixels for each frame). If there are any motion artefacts or some other patterns in the video (example: someone opened a door during the imaging), then it is best not to include those frames. If you already chose specific frames you can include them in ***Input.frames.list*** as a vector of their frame indices. Otherwise you can chose a starting frame ***Input.frames.start***, an incremental step size between frames you want to include ***Input.frames.step***, and a final frame ***Input.frames.end***.

After specifying the frames you wish to include, there is also the option of not just including those specific frames, but instead for each frame you wish to include, loading frames from that frame up to the next frame specified by ***Input.frames***, computing the average frame of these and including that image instead. You can choose to do this by setting ***Input.frames.mean*** true. It will increase the time it takes to load the video significantly but will also decrease noise contributions in your sample video.

Choose an Output-folder accessible to your MATLAB and set ***Input.output\_folder*** to the folder’s address.

Set ***Input.axial*** to the ratio between the physical length of a voxel in the axial direction vs. the physical length in the lateral direction.

Set ***Input.neur\_rad*** to the expected radius (in pixel) of a neuron. In cases with more scattering choose a bigger value. Normally 1/3 bigger is sufficient.

Set ***Input.native\_focal\_plane*** to the z-index of the native focal plane of your point spread function. This z-slice is typically the one with the highest reconstruction artefacts, or the one with the highest artefacts around it.   
  
If your workstation offers GPU-support, set ***Input.gpu\_ids*** to the indices of the gpu-Devices you wish to offer to the algorithm, otherwise set it to [].

Background estimation   
Next specify if you wish to do a background subtraction. If you wish to do so set ***Input.bg\_sub*** to true and set the number of iterations for the estimation of the background to ***Input.bg\_iter*** to 2 (More iterations are usually not necessary). The algorithm will in a first step calculate the standard deviation image of the residual video, which will be included into the estimation of neuronal centers, and also the algorithm will include the spatial background component into the SID-optimization of the spatial filters of specific neurons as the last component of ***SID\_output.forward\_model\_ini*** resulting in a continued optimization of the background, when taking local neuronal patterns into account and also significantly improve the performance of the non-negative-matrix factorization (NNMF). It is recommended to set ***Input.bg\_sub*** to true (See note at the end of the “Get Started” chapter”).

## De-trending/Bleaching

If effects of bleaching (exp. Decaying signal) are visible in your video, you can de-trend, which is performed by multiplying each pixel trace by a common estimate of the bleaching signal.   
Since the offset of the exp. Decay is not the same for all pixels, this method is not perfect of course.  
Set ***Input.detrend*** to true if you wish to perform this operation. It is recommended to do so. Also set ***Input.delta*** (TODO).

Microlens-pattern detection  
In order to prevent overfitting during the NNMF the algorithm calculates an estimation of the part of the images containing the round shapes located behind microlenses, since this is the only part containing the desired information. There are two parameters to help the algorithm give a good estimation, and this is essential. You can choose standard values for ***Input.crop\_params***, which is 2x1 double array of values between zero and ones. Otherwise the algorithm will prompt you to choose them according to the current estimation he will provide in a figure output during the run of the algorithm. The first parameter gives a cut-off value for the estimation of which part of the image even contains active microlenses, the second parameter gives a cut-off for an estimate of the microlens pattern.

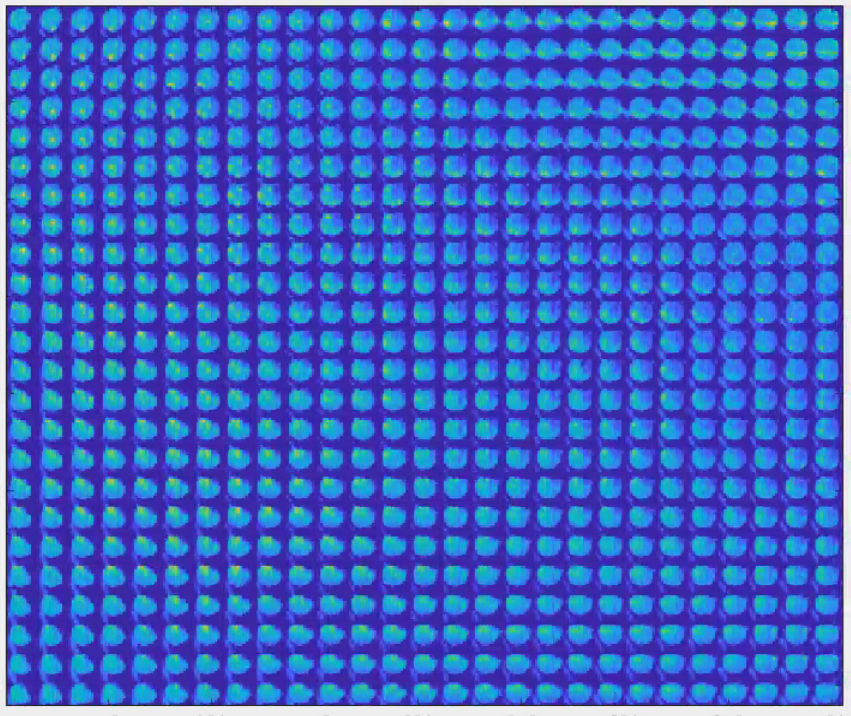


Figure 1: microlens pattern

If you wish to only analyze a certain rectangle inside your movie, specify a binary image set to true inside that rectangle you desire and false outside. You include this binary array as ***Input.crop\_mask*** and you have to set ***Input.do\_crop*** true. If ***Input.do\_crop*** is true, but there is no crop\_mask specified, the algorithm will choose the estimation of the part of the images that is active described in the previous paragraph.

## Low-rank-NNMF

An important part of the initial estimation of neuronal positions is the output of a low-rank-NNMF. (In the case of very large images sizes this can be a problem and you can choose to turn off this part of the algorithm by setting the rank (***Input.nnmf\_opts.rank***) of the NNMF to zero, but you also have to set ***Input.optimize\_kernel*** to false and not set a value for ***Input.recon\_opts.ker\_shape***.)  
  
First set the rank (***Input.nnmf\_opts.rank***) to an appropriate value. Usually about 30 components is enough, but if the reconstruction is very costly you should turn it down. Ten components still gives good results. If you increase the number of components, you increase the chance the neighboring neurons fall in different components thereby increasing the chance to separate them from each other. Increasing the rank therefore usually decreases the density of neurons in the corresponding reconstructed spatial filters of the NNMF. Set the number of iterations the algorithm should perform (***Input.nnmf\_opts.max\_iter***) to 600. This is usually sufficient!

Initialization methods for the Low-rank-NNMF  
SID offers to initialization methods. First option is to set ***Input.nnmf\_opts.ini\_method*** to ‘rand’, which initializes the temporal components T with smoothed random traces and the spatial component S with the non-negative-least-squares solution. The second option is to set ***Input.nnmf\_opts.ini\_method*** to ‘pca’ which initializes the temporal and spatial components with the absolute value of the first n (rank = n) principal components of the sample video. It is recommended to use the PCA initializer since it offers the possibility to compare different runs of the NNMF with different parameters closer and leads to an inherent background subtraction by encouraging the first component to become background.

Regularizers  
The NNMF-algorithm inside SID allows a variety of Regularizers to increase the separation performance, as well as clear up the images to prevent artefacts in the subsequent reconstruction of the spatial components of the NNMF. The NNMF-algorithm tries to solve the problem

)

where is the vector of the Lagrange multipliers and is a vector of possible regularizers.  
Each component of corresponds to a specific field of ***Input.nnmf\_opts***:

|  |  |  |  |
| --- | --- | --- | --- |
|  | meaning | Recommended value | Recommended |
| ***Input.nnmf\_opts.lamb\_ spat*** | -reg of S |  | Yes |
| ***Input.nnmf\_opts.lamb\_ temp*** | -reg of T |  | No |
| ***Input.nnmf\_opts.lamb\_corr*** | -norm of cov-matrix |  | No |
| ***Input.nnmf\_opts.lamb\_orth\_L1*** | -norm of gram matrix of S |  | Yes |
| ***Input.nnmf\_opts.lamb\_orth\_L2*** | -norm of gram matrix of S |  | Yes |
| ***Input.nnmf\_opts.lamb\_spat\_TV*** | -norm of Total Variation in space |  | No |
| ***Input.nnmf\_opts.lamb\_temp\_TV*** | -norm of Total Variation in T |  | No |

You can combine all these regularizers, but usually one is sufficient and using multiple regularizers can lead to undesired effects. It is recommended to only use ***Input.nnmf\_opts.lamb\_orth\_L1*** or ***Input.nnmf\_opts.lamb\_ spat*** with a value of 1.

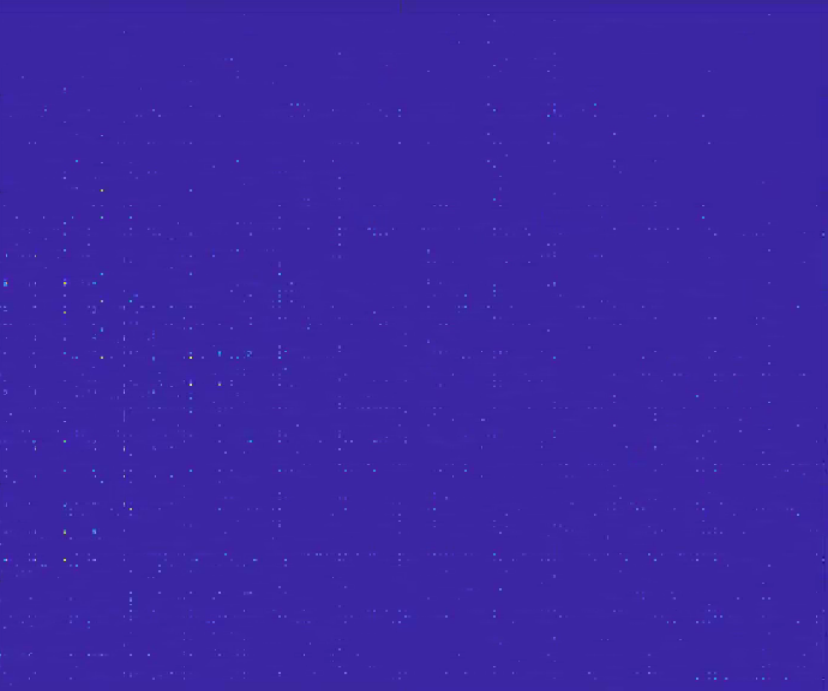
Choosing a good Lagrange multiplier/Cross-validation  
The Question how to choose the Lagrange multipliers can be addressed by manually evaluating the outputs of the *fast\_nmf* – function (SID’s low rank NNMF func.). This involves starting with an initial guess and then run the NNMF algorithm and visually evaluating the patterns in the spatial filters. If you see nicely separated single neuron LFM-patterns (case 1), then the Lagrange multiplier was a good choice, otherwise if the pattern looks very noisy, or contains incomplete single neuron LFM-patterns (case 2), the Lagrange multiplier was too large, finally if the components are rather full (as opposed to sparse) the Lagrange multiplier was probably too small (case 3). Keep in mind that one of the components (normally the first) will always be full, since it is used as the background component.  


Figure 2: Case 1



Figure 3: Case 2

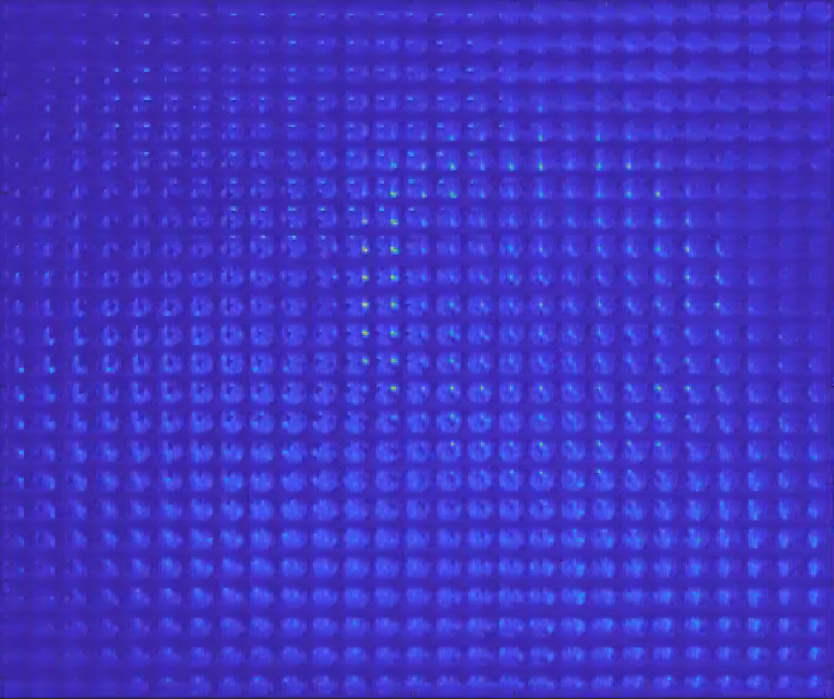


Figure 4: Case 3

Otherwise the fast\_nmf algorithm includes the option to do cross validation for a selection of values of one of the Lagrange multipliers. What cross validation does is to partitions the data in a number of ***Input.nnmf\_opts.xval.num\_part*** parts and run the NNMF algorithm for a certain Lagrange multiplier for each of the parts on the video minus that part, then generates an estimate of T on that part and computes the 2-norm of the residual. These values are then averaged over all the parts and used as an estimate on how good that parameter (Lagrange multiplier) works when you try to train the NNMF to pick up spatial components that generalize well. Of course, this is only a very indirect measure and since we cannot run the NNMF algorithm so many times on the whole video, since we are prohibited by time, the cross validation algorithm only takes into account the 100x100 sub-video with the greatest variance.  
  
It still turns out to be very useful and it is recommended to activate this function when dealing with new data. To do so, you first must set the field (sub-struct) ***xval*** inside ***Input.nnmf\_opts***. Set ***Input.nnmf\_opts.xval.num\_part*** (the number of partitions) something that leaves the size in time of all of the partitions at least bigger than the usual decay time of a Ca-transient. In the case of our data something between 5 and 10 was usually good. Then set ***Input.nnmf\_opts.xval.std\_image = SID\_output.std\_image***. This is necessary to provide the algorithm with the dimensions of the video.   
Finally set ***Input.nnmf\_opts.xval.multiplier*** to the Lagrange multiplier that you want to find the value for (eg. ‘lamb\_orth\_L1’) and set ***Input.nnmf\_opts.xval.param*** to a vector of the parameters (Lagrange multiplier values) that you want to check. If you don’t set the last two fields, the standard multiplier is ‘lamb\_orth\_L1’ and the standard Parameter values are ***opts.xval.param = lambda\*exp(-2\*[0:4])*** .

## The Reconstruction (Two strategies)

The next step in the SID pipeline is the Reconstruction of the three-dimensional information encoded in the spatial components of the NNMF. The basic reconstruction is a non-negative-least-squares (nnls) solver, but since the LFM volume does not have constant resolution, but the point spread function is based on a homogenous discretization of the volume, we encounter reconstruction artefacts near the native focal plane, where the resolution reaches a minimum of one microlens diameter. This can be remedied by various regularization techniques.

The first technique is, as before, modifying the objective function. In which case the SID-reconstruction algorithm includes the following regularizers:

|  |  |
| --- | --- |
| Name of field: | meaning |
| ***Input.recon\_opts.lamb\_L1*** | -regularization |
| ***Input.recon \_opts.lamb\_L2*** | -regularization |
| ***Input.recon \_opts.lamb\_TV\_L2*** | -norm the total variation |

***Input.recon \_opts.lamb\_TV\_L2*** is a 1x3 vector and allows different regularization for each spatial direction. The Total Variation regularization helps a lot in cases with big neurons (radius in pixels>10), since the Artefacts we usually encounter are of high spatial frequency located around the native focal plane.   
In fact if we would run the algorithm without regularization long enough we would encounter a volume consisting only of high frequency artefacts. We therefore normally do a regularization by early stopping.  
The -regularization in some cases increases the artefacts, but helps in others, when there are undesired low spat. Frequency structures. Combined with the next technique it can produce very nice results. You do not need to worry too much about choosing ***Input.recon\_opts.lamb\_L1*** too high, since the algorithm corrects it down to a proper value if it was chosen too high (so high that the zero volume is a local minimum).  
  
The second technique builds on the fact that the artefacts are of high spatial frequency and enforces sparsity on the problem by modifying the forward-projection-function (Convolution of the Volume with LFM-point spread function (psf) to generate the sensor image). This is done by choosing a kernel, that should resemble the shape of a neuron in your normal reconstruction, and convolving the Volume with that kernel before performing the convolution with the psf. The SID-reconstruction algorithm offers the following options:

|  |  |  |
| --- | --- | --- |
|  | meaning | Options |
| ***Input.recon\_opts.ker\_shape*** | Shape of the kernel: | ‘gaussian’ – Gaussian kernel ‘lorentz’ – Lorentzian kernel ‚ball‘ – binary in shape of ball ‘user’ – kernel predefined by user |
| ***Input.recon \_opts.ker\_param*** | Additional parameters for kernel | Depending on ker\_shape: ‘gaussian’- 1x2 vector First component is standard  deviation in lateral direction, second is  standard deviation in axial direction.  ‘lorentz’ – same  ‘ball’ – same (radius instead of standard deviation)  ‘user’ – ker\_param is the kernel |

Choosing an appropriate kernel and -regularization leads to very good results.

There are two good strategies for a precise estimation of the neuronal centers in the next step of the pipeline:  
  
Strategy 1  
Perform an LFM reconstruction with the regular forward-projection function and activate the -regularization and Total Variation regularization. Set Input ***Input.optimize\_kernel*** false, ***Input.recon\_opts.lamb\_L1*** to 0.1, ***Input.recon\_ opts.lamb\_TV\_L2*** to [0.1 0.1 4] and set ***Input.filter true***. This last part means, that a bandpass filter will be applied to the messy basic reconstructed volume, which should get rid of artefacts and only leave neuronal shapes in the volume.

Strategy 2  
Perform an LFM reconstruction with a modified forward-projection function and activate the -regularization. The proper kernel will be estimated through the expected radius of the neurons (***Input.neur\_rad***) and by applying Strategy one to a single NNMF-component and subsequent segmentation (see next part). Set Input ***Input.optimize\_kernel*** true, ***Input.recon\_opts.lamb\_L1*** to 1, and set ***Input.filter*** true. The bandpass-filter in the end is not strictly necessary but helps in some cases. The expected neuronal radius in the reconstructions will of course be modified before the band-pass filter, since they are sharper defined when using this strategy, as a result the bandpass-filter is very fast.  
This strategy is only advised if your NNMF generated quite sparse spatial components, like in case 2 of the low-rank nnmf chapter, but not recommended otherwise, since the algorithm tends to overfit in those cases.

The Segmentation

Given that everything worked up to this point as it should, you don’t have to change any of standard values for this section. Should over segmentation occur. You can check this by looking in the output folder during runtime and checking the ‘\_segmm\_segmentation\_’ files. If you see too many red dots at places where you cannot see neuronal shapes, you should modify the variable ***Input.segmentation.threshold***, its standard value is 0.01, which was experimentally determined, and its maximal value is 1.

There are two more fields associated with the segmentation sub-struct ***Input.segmentation:***

* ***Input.segmentation.top\_cutoff***: The segmentation ignores all neurons with a smaller z-coordinate than this threshold.
* ***Input.segmentation.bottom\_cutoff***: The segmentation ignores all neurons with a larger z-coordinate than this threshold.

## LFM-library generation

The generation of the library of LFM patterns of each of the putative neurons found by the segmentation may be performed by one of two algorithms.   
  
The first algorithm called *generate\_LFM\_library \_CPU*, produces the LFM pattern corresponding to a binary ball of radius ***SID\_output.neur\_rad*** at the location of the neuron, using a sparse forward projection algorithm.

The second algorithm called *generate\_LFM\_library\_CPU*, is a little bit more involved. It needs the information in which of the low rank NNMF components the putative neurons were found and produces for each neuron the LFM-forward-projection of the Volume containing the average of the reconstructed NNMF components where the neuron was found in a cube of length ***Input.neur\_rad***.   
This results in a better initial guess of the LFM\_library, but needs GPU support to do this in an acceptable amount of time.

The first algorithm will be used automatically if no GPU support is selected. If you wish to use the first algorithm but want to use GPU support in the rest of SID-algorithm, then set ***Input. use\_std\_GLL*** true.

## Template generation

The Template generation is straight forward. The algorithm generates for each z-slice a circle, based on the LFM-point-spread-function and a threshold ***Input.template\_threshold***. The default value is 0.01. If you wish to increase the template sizes you have to decrease this value, its range is [0,1].

## SID-Alternative-convex-search

The core of the SID-algorithm, and last step before the final extraction of the neuronal timeseries, only requires three parameters to be set. First you can choose additional regularizers by setting their Lagrange multipliers:

|  |  |
| --- | --- |
| Name of field: | meaning |
| ***Input. SID\_optimization\_args.lamb\_L1*** | -regularization |
| ***Input. SID\_optimization\_args.lamb\_L2*** | -regularization |
| ***Input. SID\_optimization\_args.lamb\_orth\_L1*** | -norm of the Gramm matrix |

Usually a value of 1e-4 is a good choice. Should the number of neurons significantly decrease during the optimization procedure, it may be too large and should be entirely turned off or it can be determined by visual inspection of the components of *SID\_output.forward\_model\_iterated (F)* after the first iteration.   
Hint: Turn the value to zero, find a component of (F), that clearly contains a neuronal signature, use a successive approximation procedure to find the value, where the noise patterns outside of the clearly defined neuronal signature vanish, then half this value and test it on multiple SID iterations.

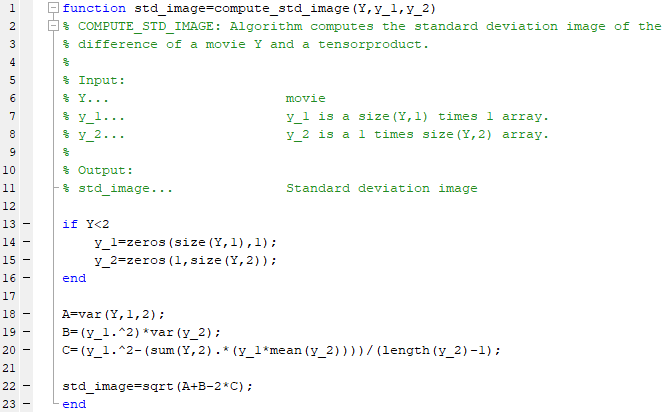
(TODO pictures)  
  
Next there is the total number of iterations the SID algorithm should perform. If you turned off the L1-regularizer of the spatial SID update, it may be advised not to exceed four iterations, since otherwise a form of overfitting will occur. It is also a matter of the constraints of computational costs, and usually there cannot be gained much beyond 5 iterations. You can set this value, by setting ***Input.num\_iter*** to the appropriate value.

Finally, if you wish to increase the size of your templates with each run of the algorithm, you can set ***Input.update\_template*** true. This will help to merge components and thereby getting rid of fragments that would be otherwise picked up as neurons.

Note: All solvers include the additional option to use the standard deviation instead of the 2-norm of the residual in the objective function. You can do so by adding ***opts.use\_std*** in the corresponding opts struct. Keep in mind to turn off the background if there is no inherent background in the movie.

# SID-sub-routines

This chapter contains detailed descriptions of the various SID-sub-routines. As well as hints for troubleshooting those sub-routines.



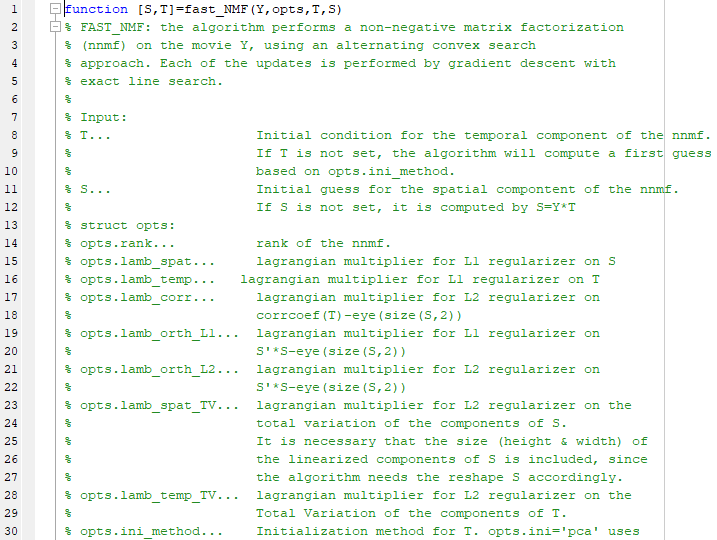
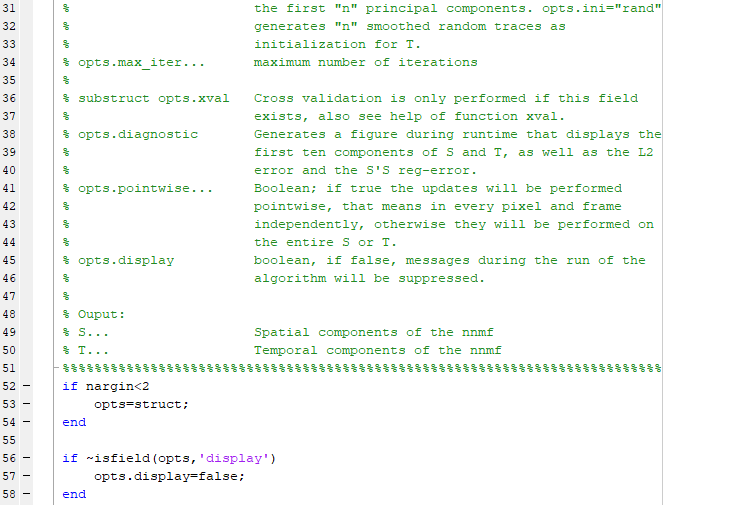
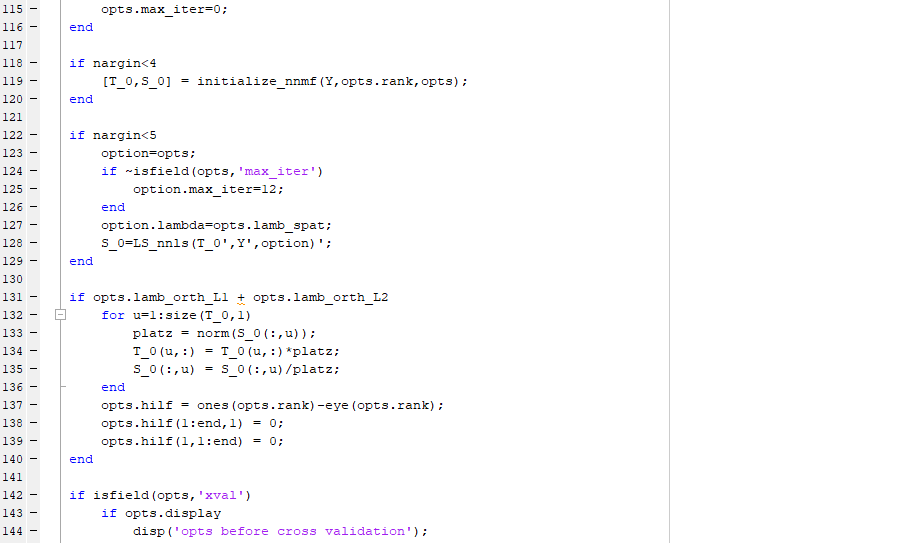
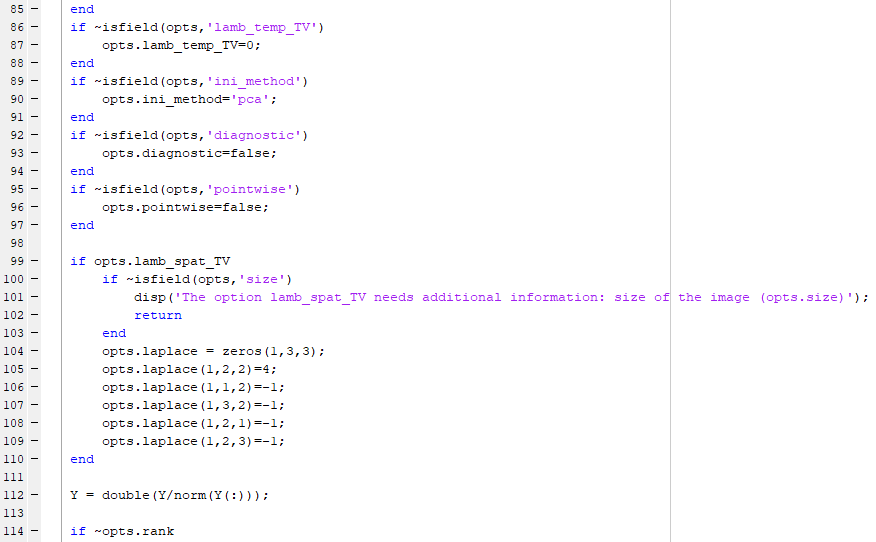
To compute the standard deviation image of the difference of a movie and a tensor product, without computing the difference, therefore saving memory, I expanded the expression for the computation of the variance and reformulated each summand in terms of movie and factors of the tensor product.

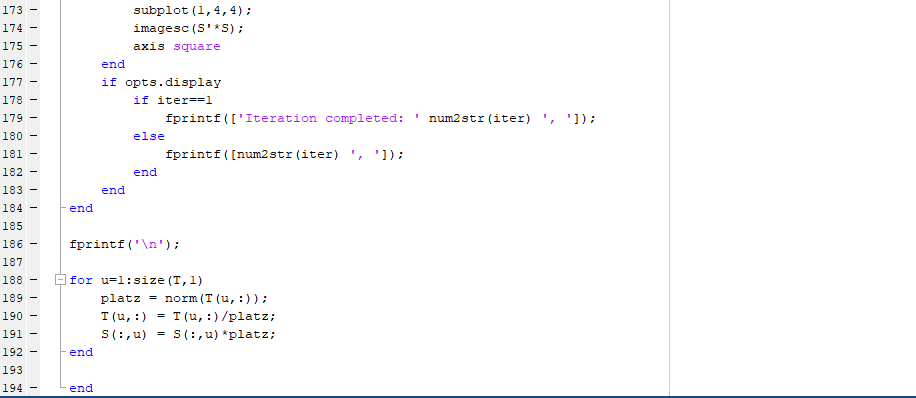
# 

This algorithm perform a form a blockwise gradient descent on the objective function

Here s corresponds to *bg\_spatial*, and t corresponds to *bg\_temporal*. This can be seen when we calculate the gradient along s and t:

and set them to zero. Between update we normalize the previously updated component. This simplifies the code and leads to better performance in the general case of NNMF.

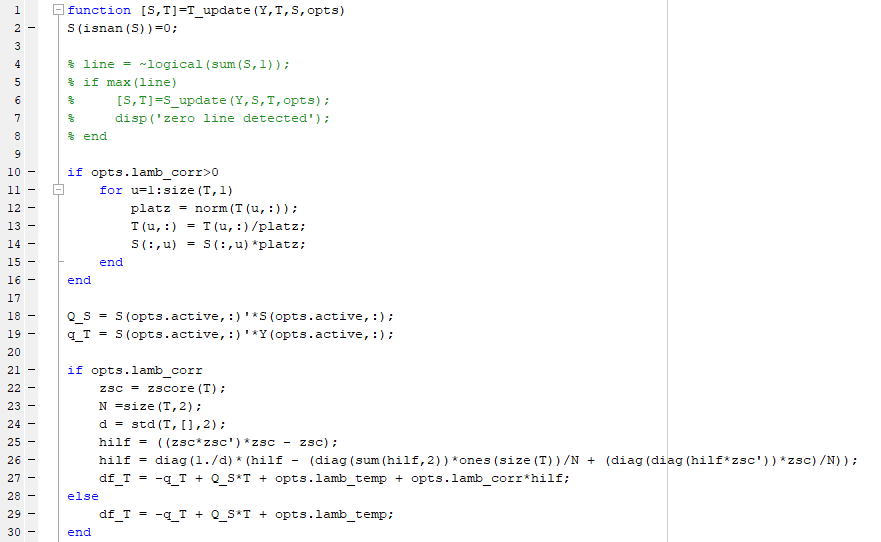


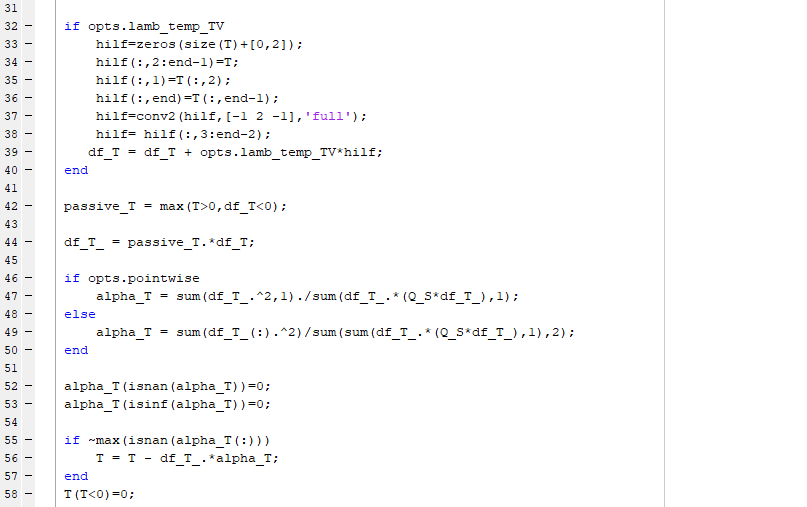
This algorithm performs updates on the variable S and T, overall resulting in an implementation of blockwise coordinate descent with exact line search and projected gradient descent.   
  
Up until line 130, the code just sets the default values and in the case where initialization is required sets initial values according to the parameters.

Between the lines 131 and 140, necessary modifications for the orthogonality regularizers are performed. Including normalization of S and generation of the variable ***opts.hilf***. This variable will be needed when computing the gradients of either of the orthogonality regularizers.

Between the lines 142 and 152, cross validation is performed, if required.

The rest of the code consists of the repeated updates performed by *S\_update* and *T\_update*, and in the case you activate diagnostic, it contains the computation and plotting of the curve of the objective function and the gram matrix of S.





This algorithm performs a gradient descent with exact line search update for the variables in T.

Initially possible nan values are replaced by zeros in line 2. Between line 10 and 16 the matrix T gets column wise normalized, while the rows of S get multiplied by the normalization, to ensure consistency.

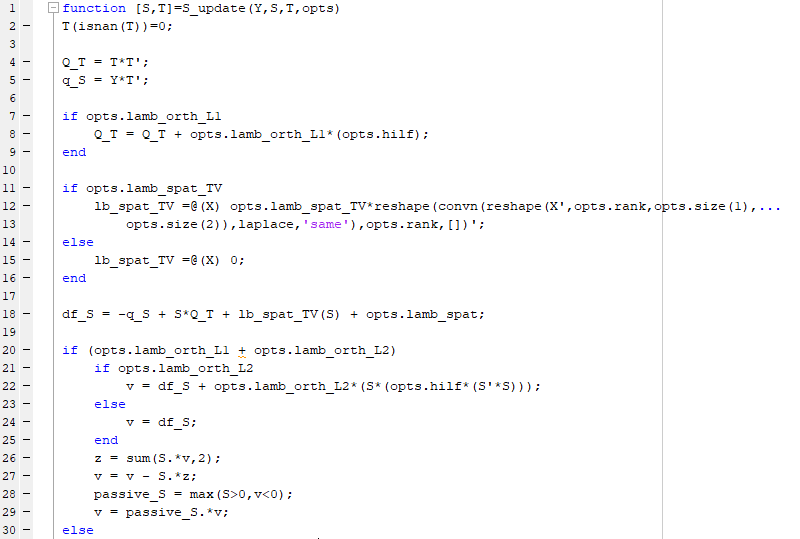
In line 18 and 19, two essential components of the gradient with regards to T are computed, namely those who summed up are the gradient of the error in the 2-norm squared between the movie Y and S\*T.

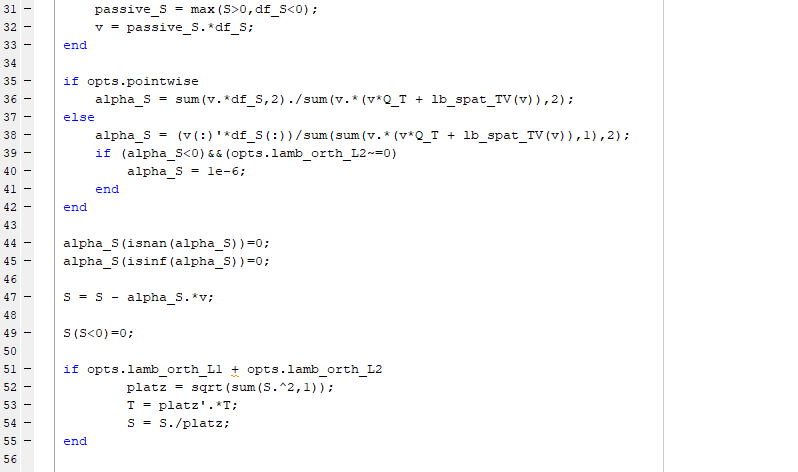
Between line 21 and 40 the necessary terms for the gradient of the correlation regularizer and of the Total Variation regularizer are computed and the components of the gradient combined.

Further in line 42 and 44 the surface projection on the non-negativity constraint of the gradient is performed.   
  
Between line 46 and 50 the algorithm computes to optimal learning rate (exact line search). This is not true in the case of the correlation regularizer!  
  
Line 52 and 53 remove nan values and infinities from the learning rate.

Line 55 to 57 performs the gradient descent step.

Line 58 projects back onto the surface of the non-negativity constraint.





This algorithm performs a gradient descent with exact line search update for the variables in S.

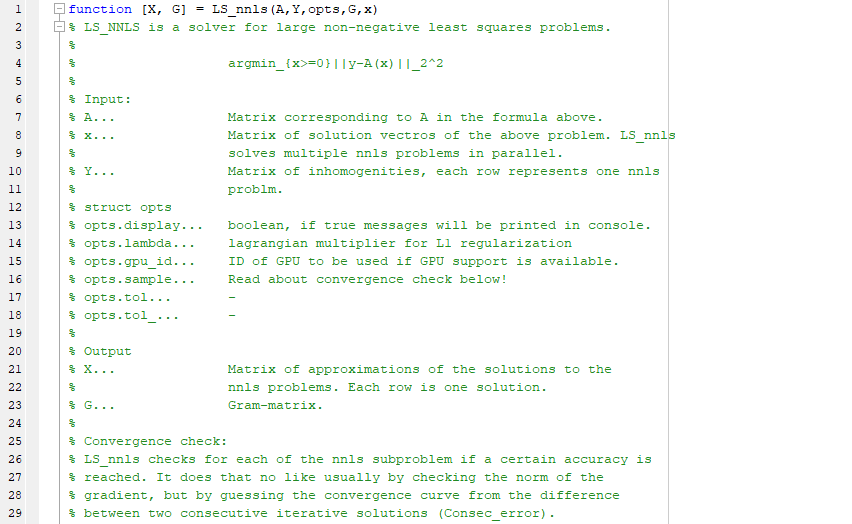
As in T\_update, in line 2 the algorithm replaces nan values in T by zeros and in line 4/5 the components of the gradient of the 2-norm of Y-S\*T are computed.

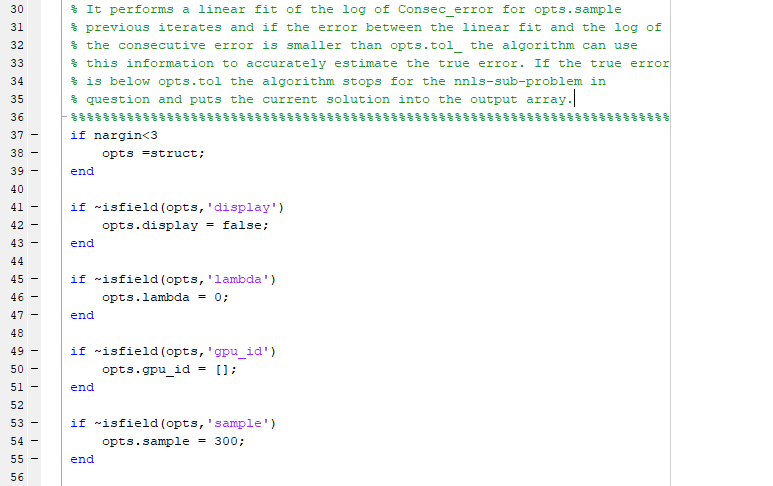
Between line 7 and 9 the matrix Q\_T is modified to include the contribution of the 1-norm orthogonality regularizer.  
  
Between line 10 and 16 a function handle is generated that performs the operation necessary to compute the contribution of the spatial Total Variation regularizer.

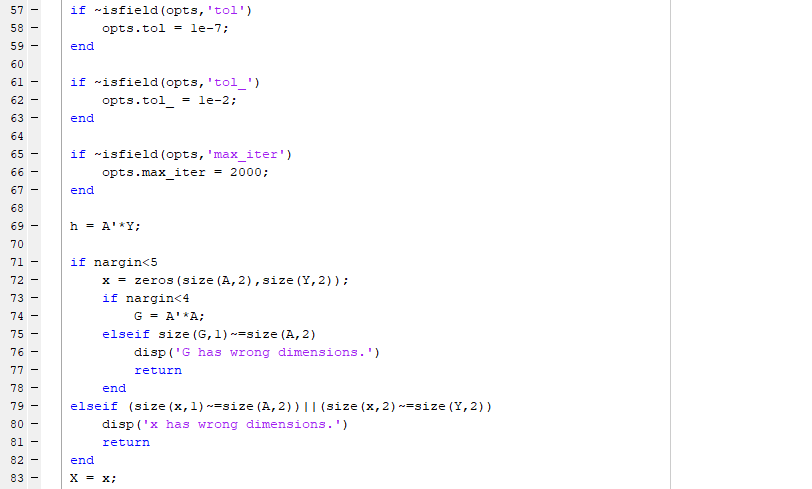
In Line 18 the gradient is assembled from its components.

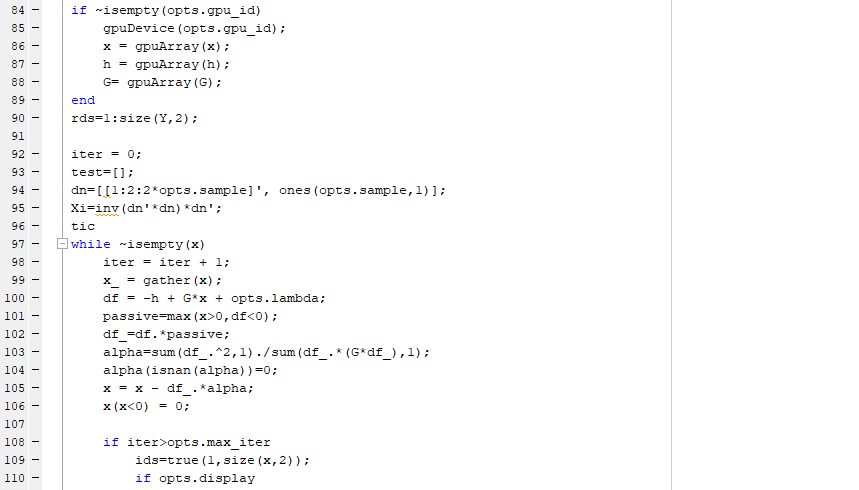
Between line 20 and 33 the direction v in which the update shall be performed is generated from the gradient via projecting along the normalization constraint we impose when we use an orthogonality regularizer, and projecting on the surface of the non-negativity constraint.   
Additionally, the final assembly of the gradientfor the 2-norm orthogonality regularizer is performed in line 22, but already not modifying the gradient, but the direction v.

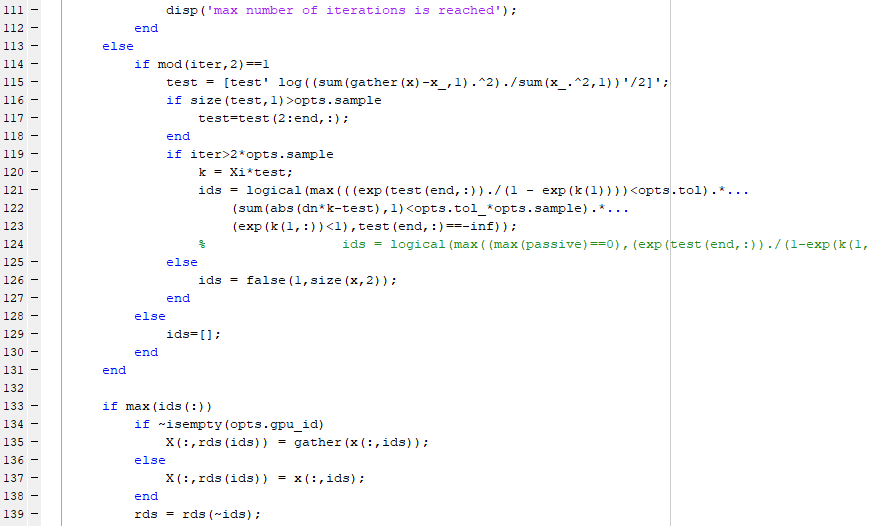
Between line 35 and 42 the exact line search is performed for the direction v, and in the case of the 2-norm orthogonality regularizer, this exact line search is approximated by exact line search for the residual gradient only. If this results in a value that leads in the opposing direction of the negative gradient, the learning rate is set to a fixed value 1e-6. This is done since the corresponding regularizer leads to a non-quadratic problem.  
  
The remaining steps are like in the case of T\_update, replacing infinities and nan values by zero in the learning rate, the update of the value of S, and finally the projection back into the constraints.







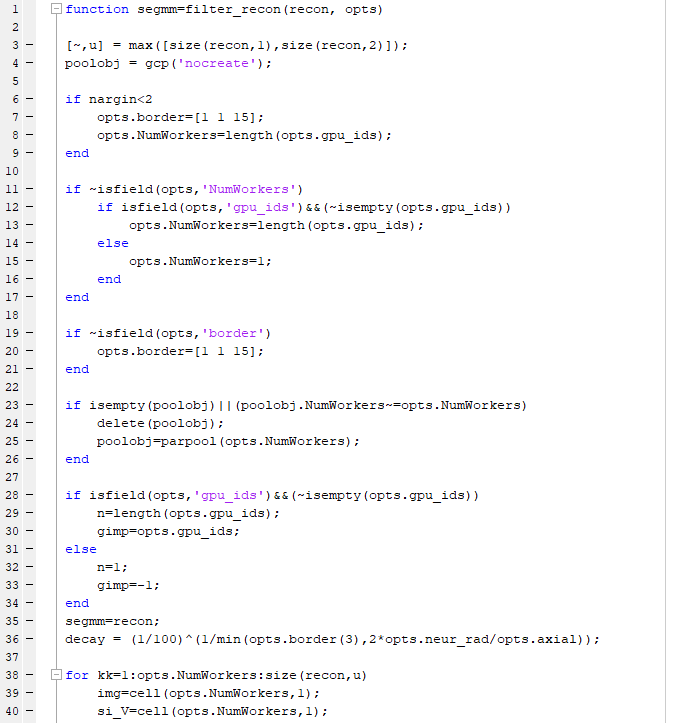


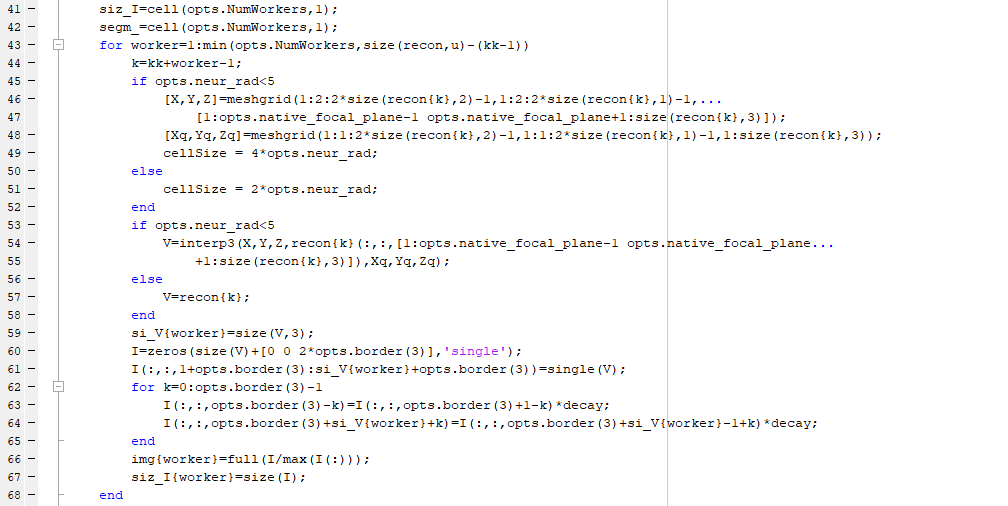


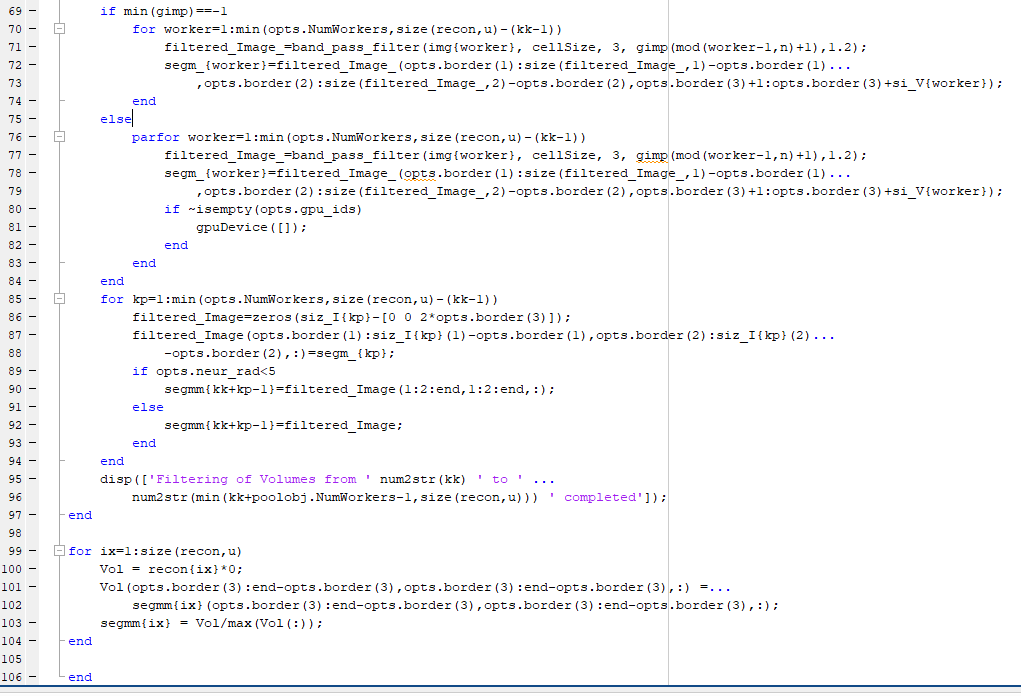


Line 37 to 67: Set default values for parameters not set by user.  
Line 69: Compute the first term of the gradient that won’t change through the algorithm  
Line 71 to 82: Check if the dimensions of the input correspond to an nnls problem, if not return; also Compute the second term (the gram matrix) that does not change over the algorithm.  
Line 97 while loop: Loop until all sub-problems have been solved up to accuracy.  
Line 99 to 106: Projected gradient descent with exact line search update.  
Line 108 to 131: Check if the number of iterations has exceeded *opts.max\_iter*, if not generate log consecutive error history (as described in function help), wait for 2 times *opts.sample* and then use the error estimator.

Line 133 to 147: If one of the nnls-sub-problems k is finished (id(k)==true), transfer k from the GPU to the output array and delete it from the current job description.





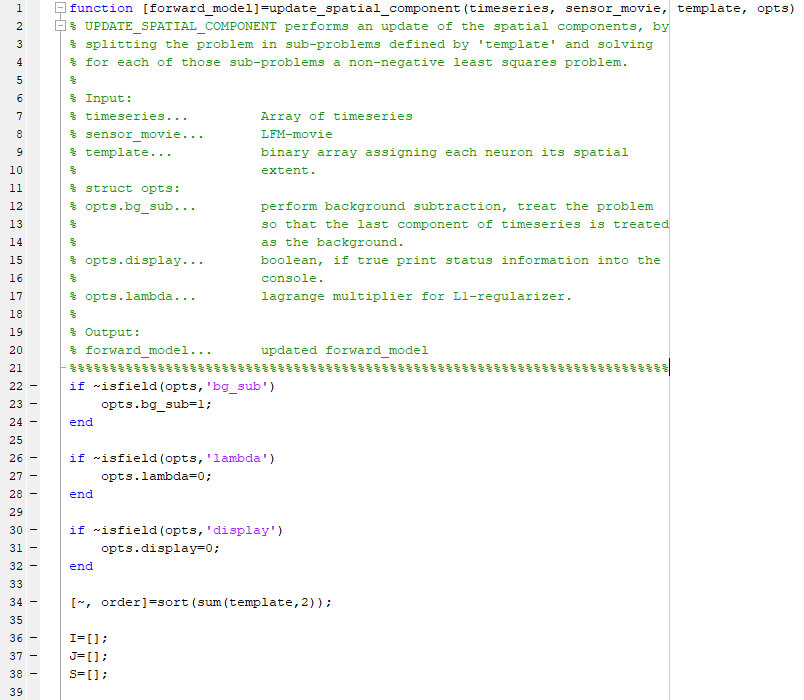


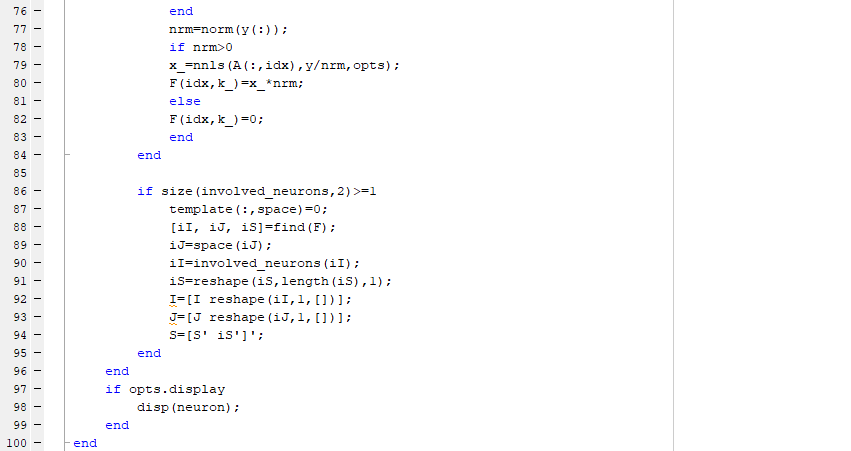
This Algorithm manages the band-pass filtering of the reconstructed volumes.

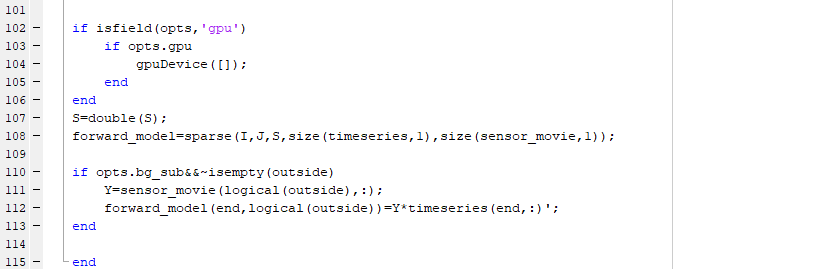
Line 6 to 21: Set default values for parameters not set by user.  
Line 23 to 26: Check if poolobj has the right number of workers, if not delete it and start a parallel pool with the right number of workers.  
Line 38: Loop over problem subsets of size NumWorker.  
Line 43 to 68: Prepare all jobs for the current problem subset  
 Line 45 to 58: If the filter size is smaller than 5, up-sample the Volumes by linear interpolation.  
 Line 59 to 65: Set smooth boundaries. (This is necessary, since a sharp fall to zero, as it would

occur with zero padding, would lead to artefacts.

Line 69 to 84: Check if GPU support exists, perform serial band pass filtering if not, on the other hand if GPU support exists, each worker controls exactly one GPU and send its job to that GPU. The number of workers can exceed the number of GPUs. This is only recommended if the multiple jobs on the same gpu do not decrease computation time too much.  
Line 84 to 94: Collect current jobs in the right cell of the segmm cell array, remove the lateral smooth boundary sheets and if necessary down-sample to counter the previous up-sampling.  
Line 99 to 104: cut away the smooth boundaries along the z-direction.





Line 22 to 32: Set default values for parameters not set by user.  
Line 34: Order neurons according to the size of their template, this is done so that sub-problems tend to be smaller.  
Line 36 to 38: Initialize the components for the sparse matrix generation.  
Line 40: Determine which part of the image is not covered by the template, therefore needs to be updated separately if ***opts.bg\_sub*** is true.  
Line 42: Loop over neurons:  
Line 43: Get the next neuron number,  
Line 44: Get all indices of pixels, where the spatial component can be non-zero according to ‘template’,  
Line 45: Check of there are any such pixels remaining, if so proceed to the next line.  
Line 46: Find all neurons that are non-zero at those indices found in Line 44,  
Line 45 to 62: check if the dimensions of timeseries and template have the right number of components, according to whether ***opts.bg\_sub*** is true, or not. Generate the sub array of template corresponding to the indices found in line 44. If ***opts.bg\_sub*** is true add a ones line add the end of the array, to incorporate the background.  
Line 63: Get the corresponding sub-movie, corresponding to the indices in line 44.  
Line 66 and 67: Generate the matrix A for the objective function of the sub-problem and F to store the solution of the sub-problem.  
Line 68 to 84: Loop over each pixel corresponding to indices found in line 44. Solve for each pixel separately.  
Line 86 to 95: Add the current results to the components I,J and S for the sparse matrix generation.  
Line 110 to 113: compute the values for the spatial background outside of the area covered by template.