Retina Segmentation Toolbox (Version 1.0)

Fabian Rathke

May 2, 2014

Contents

1 Description

This software package provides a framework for the probabilistic segmentation of retina OCT-scans (2-D and 3-D). Details about the probabilistic graphical model and the inference techniques can be found in Rathke et al. (2014).

The software is mainly written in Matlab and C and supports the usage of the GPU via the free Matlab interface GPUmat. The current version of this package can be found here.

2 Installation

Installation in Matlab is accomplished by adding the source code path via

```
addpath('/path/to/source/code');
```

either into the respective Matlab script or for a permanent adding into the startup.m, a script called by Matlab during each startup. Some functions were written in C, to speed up computations. To be able use these files, they have to compiled from within Matlab using the *mex* function. There are three C functions:

```
cd collector/helperFunctions/
mex getRaster.c
cd ../../prediction/predVariational
mex optQCC.c
cd ../../training/helperFunctions
mex cglasso.c
```

Furthermore, several functions have be implemented such that they can be either run on the CPU or on a Nvidia GPU. This functionality requires the GPUmat toolbox for Matlab and CUDA. See their web page for installing instructions.

3 Basic Usage

In order to segment an OCT scan, we need to *train* appearance models and a shape prior model. To do this we need *ground truth* in form of manually labeled OCT scans. If you have no ground truth available yet, you can use our labeling tool to create some, see Section 5.

After we trained appearance and shape models, we can *predict* segmentations of unseen OCT scans. If ground truth is available for these scans, we can *evaluate* the prediction using various error measures.

We were not allowed to publish the data sets that were used for our publication. But to provide a minimal example, the package contains a circular scan of the authors eye and a model trained on the set of 80 circular scans described in (Rathke et al., 2014). The subsequent sections introduce code for the basic steps outlined above using *scripts/genericCallerScript.m*. Subsequently we will show how to segment the provided circular scan, relying in code that can be found in *scripts/useExampleScan.m*.

To keep this introduction as clear as possible, we rely mostly on the default values for parameters. These correspond to those used in (Rathke et al., 2014) to obtain all results. Section 4 will introduce all important parameters and how to set them.

3.1 Configuration

In this section we explain how to set up the *collector*, which is responsible for fetching patches for training and testing/prediction. Most parameters are specified to their default values inside `setCollectorDefaults`.

```
% set folders that hold the data
folderData = '/path/to/data/';
folderLabels = '/path/to/ground/truth/';
files = dir([folderData '*.mat']); % fetch all files
% set up collector; we use the default parameters
% use 'help setCollectorDefaults' for an overview about model parameters
parameters = struct();
% options that define how to collect ground truth and drawing patches from the OCT-scans
collector.options.labelIDs = repmat(1,length(files),1);
collector.options.loadRoutineData = 'routineData';
collector.options.loadRoutineLabels = 'LabelsFromLabelingTool';
% set the default values for the remaining parameters
collector.options =
    setCollectorDefaults(collector.options,parameters,files,folderData,folderLabels);
% set the collector functions for training and testing
collectorTrn.name = @collectTrnData; collectorTrn.options = collector.options;
collectorTest.name = @collectTestData; collectorTest.options = collector.options;
```

3.2 Training

We set up the approaches for creating appearance and shape models. It is very easy to replace the appearance models by user-defined approaches, i.e. a random forest or support vector machines. After set-up, we start the training procedure by calling `trainModels`.

```
% set the appearance model; we use the default options for both appearance models and shape
    prior
trainFunc.options.appearanceModel = struct('appearanceModel',@trainGlasso);
trainFunc.options.shapeModel = struct();
% train the models; spare the last 5 files for testing the model performance
model = trainModels(files(1:end-5),collectorTrn,parameters,trainFunc.options);
```

3.3 Prediction

After we trained our segmentation model, we are now able to make predictions for unseen scans. Again we rely on mostly default parameters, but we want intermediate and final results to be plotted, so we provide a folder and switch on plotting. We then start the prediction procedure by calling `predVariational`.

```
% setup predictor for appearance terms
testFunc.options.predAppearance = @predGlasso;
testFunc.options.plotting = 1;
testFunc.options.folderPlots = '/folder/to/store/plots/';
% make predictions for the last 5 files in the set
prediction =
    predVariational(files(end-4:end),collectorTest,parameters,model,testFunc.options);
```

3.4 Evaluation

Finally, if ground truth is available, we can evaluate segmentation performance in terms of the signed and unsigned error. User-defined error measures can be defined alternatively. We obtain the final results by calling `signedUnsigned`.

```
results = signedUnsigned(prediction,struct());
```

3.5 Make Prediction for Sample Scan

We set up the collector and load the pre-trained segmentation model provided with the package in *scripts/circularScanModel*. We then calculate transition matrices needed for the inference. Usually these are contained in the model after training, but were left out to keep the model file small.

```
% set scriptFolder to the folder 'script' inside the package
scriptFolder = '/path/to/script/Folder/';
exampleFile = dir([scriptFolder 'circularScan.mat']);
% set collector defaults
collector.options =
    setCollectorDefaults(struct(),struct(),exampleFile,scriptFolder,scriptFolder);
% load the trained model provided with the package
load([scriptFolder 'circularScanModel']);
% calculate transition matrices
model.shapeModel = preCalcTransitionMatrices(collector,model.shapeModel);
```

We now configure the inference part and perform the segmentation. We print some error measures and plot the predicted segmentation.

```
collectorTest.name = @collectTestData; collectorTest.options = collector.options;
prediction = predVariational(exampleFile,collectorTest,struct(),model,struct());

% evaluate the prediction, calculate unsigned and signed error measures
results = signedUnsigned(prediction,struct());
str = [sprintf('The unsigned error for each layer in mum is:\n') sprintf('%d:
    %.3f\n',[(1:9); results.unsigned_per_layer'*3.87]) sprintf('Avg:
    %.3f',results.unsigned*3.87)]; disp(str);

% load the example scan for plotting
data = load([scriptFolder 'circularScan.mat'],'B0'); B0 = sqrt(sqrt(data.B0));
% plot the scan and its segmentation, the ground truth and the scan itself
plotBScan(B0,prediction.prediction{1},prediction.columnsPred,[]);
plotBScan(B0,prediction.trueLabels{1},[1:768],[]);
figure; imagesc(B0); colormap gray;
```

4 Parameter Glossary

4.1 Collector (`collector.options`)

Defaults are set in the function `setCollectorDefaults.m`. An overview over all parameters and their corresponding default values is given in Table 1.

4.1.1 `width` and `height`

The patch size in pixel, `width` has to be an odd number.

4.1.2 `clip` and `clipRange`

Enables to clip B-Scans at the left and right boundary, defined via by `clipRange`.

4.1.3 `preprocessing`

Enables preprocessing of each scan at the patch-level or scan-level. `preprocessing` is a struct itself with fields `patchlevel` and `scanlevel`. Each field consists of a cell array where each entry holds the link to the function as the first value and all following values denote parameters. See the corresponding folders in *collector/preprocessing* for available filter functions and their help functions for the required parameters. Feel free to add your own functions.

4.1.4 `loadRoutineData` and `loadRoutineLabels`

Specifies the loading routines for B-scans and ground truth. For the former, the user has to add a small routine in *loadData.m*. For the latter, if the labels were established with our labeling tool, one can use the default setting for `loadRoutineLabels`. Otherwise, there also must be added a user-defined routine this time in *loadLabels.m*.

4.1.5 `numRegionsPerVolume` and `labelIDs`

For 3-D scans, one has to set the number of B-scans (= regions) that are used from each volume. The accompanying variable labelIDs holds numerical identifier for choosing `numRegionsPerVolume` scans from each 3-D volume. For example there are 50 B-scans per volume, and one wants to work with only every 5-th, one sets `numRegionsPerVolume=10` and `labelIDs = repmat(1:5:50,numFiles,1)`.

4.1.6 `BScanRegions`

Divide each B-scan into different regions, for which separate sets of appearance models are trained. Array with two columns, where each row indicates left and right boundary of one segment, e.g. $[1\ 250; 251\ 500]$ for two segments of 250 pixels width.

4.1.7 `calcOnGPU`

The switch activates calculation on the GPU. Requires a Nvidia GPU with CUDA and the Matlab toolbox GPUmat installed.

Table 1: Parameters for the collector module, stored in the structure `collector.options`. If not set by user, default values are set in `setCollectorsDefaults.m`.

Variable	**Type**	**Default**
width	int	15
height	int	15
clip	boolean	false
clipRange	array	[1 scan-width]
preprocessing	struct	options.preprocessing.patchLevel = {{@projToEigenspace,20}}
loadRoutineData	string	'spectralis'
loadRoutineLabels	string	'LabelsFromLabelingTool'
numRegionsPerVolume	int	1
labelIDs	array	zeros(numFiles,numRegionsPerVolume)
BScanRegions	array	[1 numColumns]
calcOnGPU	boolean	false
numPatches	int	30
patchPosition	string	'middle'
centerPatches	boolean	true
columnsShape	cell-array	columnsShape{i} = [1:2:scan-width]
columnsPred	array	1:2:scan-width
printTimings	boolean	false
verbose	int	1
saveAppearanceTerms	boolean	0

4.1.8 `numPatches`, `patchPosition` and `centerPatches`

The first variables denotes the number of patches drawn per B-scan and appearance class. The columns are chosen evenly distributed across the scan. The second variables determines from where the patches for layer classes are drawn, from the center of each layer ('middle') or from a random position ('random'). The last variable indicates, whether the mean intensity value is subtracted from each patch.

4.1.9 `columnsShape` and `colummnsPred`

`columnsShape` is an cell-array, with one entry for each used scan in the volume. Each entry is an array that gives the columns which will be used for the shape prior. Intermediate columns are interpolated. This allows for sparse shape prior representations. The second variable `columnsPred` is an array with the columns in one B-Scan for each a prediction will be made. Can only be set once for all scans in a volume.

4.1.10 `printTiming` and `verbose`

Prints timings for several subroutines during prediction. The latter determines the level of printed output, takes values between 0 and 2.

4.1.11 `saveAppearanceTerms`

In addition to the segmentation, the output of the probabilistic appearance models for all pixels in the scan is returned.

4.2 Training (`trainFunc.options`)

This options are set in the struct `trainFunc.options`, they control the creation of the appearance and shape models. An overview over all parameters and their corresponding default

Table 2: Parameters for the training part, stored in the structure `trainFunc.options`. If not set by user, default values are set in `setAppearanceDefaults.m` and `setShapeDefaults.m`. The dashed line separates variables for the appearance and shape part of the training.

Variable	**Type**	**Default**
appearanceModel	function-handle	@trainGlasso
glasso	float	0.01
numModes	int	20
loadShape	boolean	false
loadShapeName	string	hash of the concatenated filenames
saveShape	boolean	false
saveShapeName	string	hash of the concatenated filenames
saveCompressedModel	boolean	true
shapeFolder	string	'$HOME/shapemodels'

values is given in Table 2. Default values are set in `setAppearanceDefaults.m`, `trainGlasso.m` and `setShapeDefaults.m`.

4.2.1 trainAppearance.m (`trainFunc.options.appearanceModel`)

4.2.1.1 `appearanceModel`

Holds the handle of the appearance model to be used for training.

4.2.1.2 `glasso`

Parameter that governs the sparseness of the precision matrices to be estimated if using `predGlasso`. Smaller values correspond to denser matrices.

4.2.2 trainShape.m (`trainFunc.options.shapeModel`)

4.2.2.1 `numModes`

The number of modes used for the Probabilistic Principle Component Analysis (Tipping and Bishop, 1999) used to obtain a regularized estimate of the shape distribution covariance matrix.

4.2.2.2 `loadShape` and `loadShapeName`

Can be used to load previously calculated shape models. The name only refers to the filename, the folder is given with `shapeFolder`, see below.

4.2.2.3 `saveShape`, `saveShapeName` and `saveCompressedModel`

Can be used to save the just calculated shape model. Again, the name only refers to the filename, the folder is given with `shapeFolder`, see below. The last variable determines whether only a reduced model is saved and some parts are recalculated upon loading. This significantly reduces storage requirements.

4.2.2.4 `shapeFolder`

The folder were saved shape models are stored.

Table 3: Parameters for the prediction part, stored in the structure `testFunc.options`. If not set by user, default values are set within `predAppearance.m` and in `setSegmentationDefaults.m`. The dotted line seperates this two sets of parameters.

Variable	**Type**	**Default**
predAppearance	function-handle	@predGlasso
normalizeDataTerm	boolean	true
logOutput	boolean	false
iterations	int	30
threshold	float	0.01
plotting	boolean	false
folderPlots	string	'$HOME/plots'
alpha	float	0.1
calcFuncValue	boolean	false
detailedOutput	boolean	false
threshold_q_c	float	10^{-6}

4.3 Prediction (`testFunc.options`)

This options are set in the struct `testFunc.options`, they control the calculation of predictive appearance terms and the variational inference approach. Defaults are set inside `predAppearance.m` and in `setSegmentationDefaults.m`

4.3.1 predAppearance.m

4.3.1.1 `normalizeDataTerm`

If true, generative appearance terms will be converted to discriminative ones, such that the sum over models for each pixel adds to one.

4.3.1.2 `logOutput`

If true, the appearance terms will be returned in log-space.

4.3.1.3 `predAppearance`

Handle of the appearance model used for prediction, should be compatible to the one used for training.

4.3.2 predVariational.m

4.3.2.1 `iterations` and `threshold`

Controls the number of iterations performed and the amount of mean change in pixel between iterations that causes the optimization to finish.

4.3.2.2 `plotting` and `folderPlots`

Plots will be made after initialization and after each iteration and stored in `folderPlots`.

4.3.2.3 `alpha`

Determines the coupling between the distributions q_c and q_b during optimization. Smaller numbers correspond to less coupling.

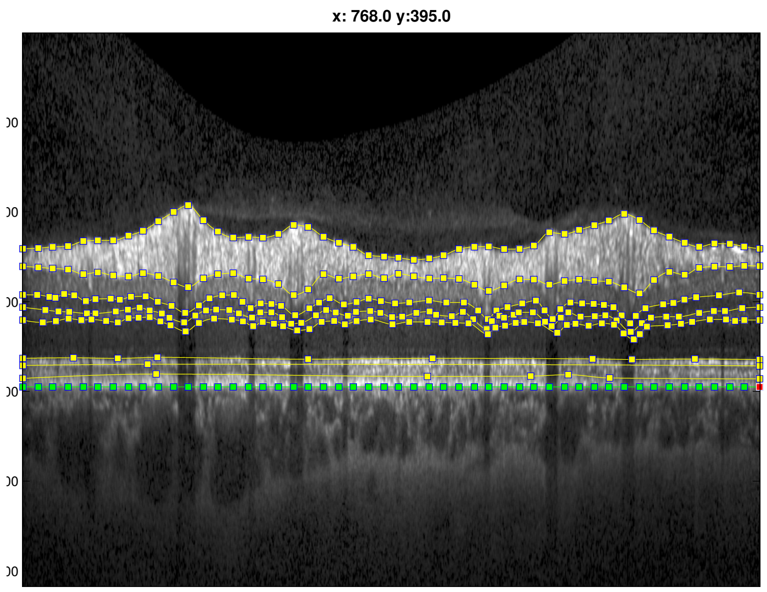

Figure 1: B-Scan with labels provided for 9 boundaries. The scan was flattened along the 9th boundary to ease labeling (press 'r' with the respective boundary selected).

4.3.2.4 `calcFuncValue`

Triggers the calculation of data terms that can be used for pathology detection and to access the uncertainty of the segmentation. See Rathke et al. (2014, Sec. 6.1.2) for details.

4.3.2.5 `detailedOutput`

Outputs information how the error for q_c and q_b evolved during the iterations. Also outputs the mode of q_b (in general less accurate than the expectation over q_c).

4.3.2.6 `threshold_q_c`

The all values below this threshold are set to zero during the optimization. This is done to speed up computations.

5 Labeling Tool

Coming with our software package we provide a labeling tool that facilitates the creation of ground truth. Fig. 1 shows a B-Scan with 9 boundaries labeled. We provide a generic configuration script and the most important options in the next to sections.

5.1 Configuration

```
% the ID of the B-scan (for Spectralis scans either 0 for 2-D or {0,...,61} for volumes)
collector.options.labelID = ID;
collector.options.loadRoutineData = 'dataRoutine';
collector.options.folder_data = '/path/to/oct/scans/';
% if restore == 1 and we have already labeled data with the label tool
collector.options.saveDir = '/path/to/save/labels/';
% if restore == 1, we have not labeled anything yet but there are labels provided with the
    dataset
```

```
collector.options.loadRoutineLabels = 'loadRoutine';
collector.options.folder_labels = '/path/to/initial/labels';
% we want no preprocessing (we could add here a Gaussian filter for example or any
    user-defined function to ease labeling)
collector.options.preprocessing.scanLevel = {};
% using existing labels?
restore = 1;
% we label the B-Scan with ID stored inside 'filename.mat';
labelTool('filename.mat',collector,restore);
```

This script can be found in *scripts/exampleLabelingTool.m*.

5.2 General Usage

After you call the function you will see the scan and labels already provided, in case you choose to restore them. First you can crop the scan in order to cut away parts of the scan that are of no interest, first the vitreous body, then the choroid. You can skip each step by pressing '`c`'.

After that you can insert new boundaries by pressing '`n`' and then clicking inside the scan at the correct height. You can hide boundary lines by pressing '`b`'. You can also hide the label markers by pressing '`ctrl + b`'. To ease labeling, you can flatten the B-Scan along a boundary by selecting this boundary (use '`q`' and '`w`' to switch between boundaries) and pressing '`r`'. You can fine-tune each label marker by selecting it first (use '`z`' and '`x`' to cycle through all markers within the current boundary) and then using the arrow keys. To delete the selected boundary press '`ctrl + d`'.

Different modes control the behavior of a left-click. The standard mode is 'add' ('`a`'), which adds a new label marker at the clicked position. The 'edit' ('`e`') mode moves the nearest marker to the click position. The 'fuse' ('`f`') mode is useful, if you want to label boundaries around the optical nerve. New marker are added on top of the nearest marker. Finally, the 'delete' ('`d`') mode erases the closest marker.

These and further options are documented inside the m-file and can be accessed by typing '`help labelTool`'.

6 Contact

In case you find any bugs or have questions regarding the code, please do not hesitate to contact Fabian Rathke. Alternatively, you can check the project page for updates and bugfixes.

References

Rathke, F., Schmidt, S., and Schnörr, C. (2014). Probabilistic intra-retinal layer segmentation in 3-D OCT images using global shape regularization. *Med. Image Anal.* in press.

Tipping, M. E. and Bishop, C. M. (1999). Probabilistic principal component analysis. *J. R. Stat. Soc.*, 61(3):pp. 611–622.