



Microstructure Diffusion Toolbox

Release 1.2.6

Robbert Harms

Aug 15, 2020

Contents

1	Introduction	1
1.1	Summary	1
1.2	Links	1
1.3	HCP Pipeline	2
1.4	Quick installation guide	2
2	Installation	3
2.1	Ubuntu / Debian Linux	3
2.2	Windows	3
2.3	Mac	4
2.4	Initialization	4
2.5	Test the installation	4
3	Maximum Likelihood Estimation	5
3.1	HCP Pipeline	5
3.2	MDT example data	5
3.3	Graphical interface	6
3.4	Command Line interface	10
3.5	Python interface	12
3.6	Available optimization routines	15
4	MCMC sampling	17
4.1	Post-processing	17
5	Adding components	19
5.1	Defining components	19
5.2	Composite models	21
5.3	Compartment models	24
5.4	Parameters	26
5.5	Library functions	27
5.6	Batch profiles	28
6	Maps visualizer	29
6.1	GUI Layout	29
6.2	Plot configuration	32
7	Design concepts	35
7.1	Protocol	35
7.2	Input data	36
7.3	Dynamic modules	36
7.4	CL code	36
8	Advanced configuration	39
8.1	General config	39
8.2	GUI specific config	39
8.3	Runtime configuration	39
9	Frequently Asked Questions	41
9.1	Installation problems	41
9.2	Analysis	41
10	Scientific articles	43
11	Credits	45

Chapter 1

Introduction

The Microstructure Diffusion Toolbox (MDT) is a framework and library for microstructure modeling of magnetic resonance imaging (MRI) data. The aim of MDT is to provide reproducible and comparable model fitting for MRI microstructure analysis. As such, we provide a common platform for microstructure modeling including many models that can all be processed using the same optimization routines. For maximum performance all models and algorithms were implemented to make use of all parallel processing capabilities of modern computers. MDT combines flexible modeling with fast processing, targeting both model developers and data analysts.

1.1 Summary

- GPU accelerated processing
- Human Connectome Project (HCP) pipelines
- Includes CHARMED, NODDI, BinghamNODDI, NODDIDA, NODDI-DTI, ActiveAx, AxCaliber, Ball&Sticks, Ball&Rockets, Kurtosis, Tensor, VERDICT, qMT, and relaxometry (T1, T2) models.
- Includes Gaussian, Offset-Gaussian and Rician likelihood models
- Includes Powell, Levenberg-Marquardt and Nelder-Mead Simplex optimization routines
- Includes multiple (adaptive) MCMC sampling algorithms
- Supports hyperpriors on parameters
- Supports gradient deviations per voxel and per voxel per volume
- Supports volume weighted objective function
- Supports adding your own models
- Offers Graphical, command line and python interfaces
- Computations are parallelized over voxels and over volumes
- Python and OpenCL based
- Free Open Source Software: LGPL v3 license
- Runs on Windows, Mac and Linux operating systems
- Runs on Intel, Nvidia and AMD GPU's and CPU's.

1.2 Links

- Full documentation: http://mdt_toolbox.readthedocs.io
- Project home: <https://github.com/robbert-harms/MDT>

1.3 HCP Pipeline

MDT comes pre-installed with Human Connectome Project (HCP) compatible pipelines for the MGH and the WuMinn 3T studies. To run, after installing MDT, go to the folder where you downloaded your (pre-processed) HCP data (MGH or WuMinn) and execute:

```
$ mdt-batch-fit . NODDI
```

and it will autodetect the study in use and fit your selected model to all the subjects.

1.4 Quick installation guide

The basic requirements for MDT are:

- Python 3.x
- OpenCL 1.2 (or higher) support in GPU driver or CPU runtime

Linux

For Ubuntu >= 16 you can use:

- `sudo add-apt-repository ppa:robbert-harms/cbclab`
- `sudo apt-get update`
- `sudo apt-get install python3-mdt python3-pip`
- `sudo pip3 install tatsu`

For Debian users and Ubuntu < 16 users, install MDT with:

- `sudo apt-get install python3 python3-pip python3-pyopencl python3-numpy python3-nibabel python3-pyqt5 python3-matplotlib python3-yaml python3-argcomplete libpng-dev libfreetype6-dev libxft-dev`
- `sudo pip3 install mdt`

Note that `python3-nibabel` may need NeuroDebian to be available on your machine. An alternative is to use `pip3 install nibabel` instead.

A Dockerfile and Singularity recipe were kindly provided by Ali Khan (on github: akhanf). These dockers come with Intel OpenCL drivers pre-loaded (e.g. for containerized deployment on a CPU cluster). For example, to install using Docker use `docker build -f containers/Dockerfile.intel ..`

Windows

The installation on Windows is a little bit more complex and the following is only a quick reference guide. For complete instructions please view the [complete documentation](#).

- Install Anaconda Python 3.*
- Install MOT using the guide at <https://mot.readthedocs.io>
- Open an Anaconda shell and type: `pip install mdt`

Mac

- Install Anaconda Python 3.*
- Open a terminal and type: `pip install mdt`

Please note that Mac support is experimental due to the unstable nature of the OpenCL drivers in Mac, that is, users running MDT with the GPU as selected device may experience crashes. Running MDT in the CPU seems to work though.

For more information and full installation instructions see https://mdt_toolbox.readthedocs.org

Chapter 2

Installation

2.1 Ubuntu / Debian Linux

Using the package manager, installation in Ubuntu and Debian is relatively straightforward.

For **Ubuntu >= 16** the MOT package can be installed from our Personal Package Archive (PPA) using:

```
$ sudo add-apt-repository ppa:robbert-harms/cbclab
$ sudo apt-get update
$ sudo apt-get install python3-mdt python3-pip
$ sudo pip3 install tatsu
```

By using a PPA your Ubuntu system can update MDT automatically whenever a new version is out. Unfortunately there is no debian package for the Tatsu requirement yet, as such, we need to install it manually.

For **Debian**, and **Ubuntu < 16**, using a PPA is not possible (because of missing dependent packages) and we need a more manual installation. Please install the dependencies first:

```
$ sudo apt install python3 python3-pip python3-pyopenc1 \
python3-numpy python3-nibabel python3-pyqt5 \
python3-matplotlib python3-yaml python3-argcomplete \
libpng-dev libfreetype6-dev libxft-dev
```

Note that `python3-nibabel` may need NeuroDebian to be available on your machine. An alternative is to use `pip3 install nibabel` instead.

Next, install MDT with:

```
$ sudo pip3 install mdt
```

This might recompile a few packages to use the latest versions.

Alternatively, you could try the Singularity recipe at <https://github.com/akhanf/mdt-singularity>, kindly made available by an user of MDT.

A docker installation is also available. If you do, a tip from an MDT user is that “when you use Docker it is mandatory to mount the <https://github.com/robbert-harms/MDT/tree/master/mdt/data> directory in the `~/mdt/<latest version>/` inside container in order to work.”

After installation please continue with the section *Initialization* below.

2.2 Windows

MDT uses the Microstructure Optimization Toolbox (MOT) for all analysis computations. Please install MOT first (<https://mot.readthedocs.io/en/latest/install.html#windows>). Afterwards this installation should be fairly straightforward.

Note that MDT depends on PyQt5 so make sure you do not attempt to run it in an environment with PyQt4 or earlier. If you followed the MOT install guide and installed the Anaconda version 4.2 or higher with Python3.x, you should be fine. Again, see <https://mot.readthedocs.io/en/latest/install.html#windows> for details.

Having followed the MOT install guide we can now install MDT. Open an Anaconda console and use:

```
$ pip install mdt
```

If that went well please continue with the *Initialization* below.

2.3 Mac

Installation on Mac is pretty easy using the Anaconda 4.2 or higher Python distribution. Please download and install the Python3.x 64 bit distribution, version 4.2 or higher which includes PyQt5, from [Anaconda](#) and install it with the default settings.

Afterwards, open a terminal and type:

```
$ pip install mdt
```

To install MDT to your system. If that went well please continue with the *Initialization* below.

Please note that Mac support is experimental due to the unstable nature of the OpenCL drivers in Mac. Users running Running MDT with the GPU as selected device may experience crashes. Running MDT in the CPU seems to work though.

2.4 Initialization

After installation we need to initialize the MDT components folder in your home folder. Use:

```
$ mdt-init-user-settings
```

in your bash or Anaconda console to install the MDT model library to your home folder.

2.5 Test the installation

If all went well and MDT is installed and initialized, we can now perform some basic tests to see if everything works well. The first command to try is:

```
$ mdt-list-devices
```

which should print to the console a list of available CL devices. If this crashes or if there are no devices returned, please check to see if your OpenCL drivers are correctly installed. If it works but no devices can be found then please refer to the section *No OpenCL device found*.

Next, one could try starting the graphical interface using:

```
$ mdt-gui
```

or, equivalently,

```
$ MDT
```

This should start the GUI. If there are problems in this stage it is most likely related to Qt problems. Please check if you have installed the Qt5 package and not the Qt4 package.

Chapter 3

Maximum Likelihood Estimation

Maximum Likelihood Estimation (MLE), otherwise known as model fitting or model inversion, is one of the core strengths of MDT. Using GPU accelerated fitting routines and a rich library of available models, MDT can fit many types of MRI data within seconds to minutes [Harms2017]. The main workflow is that you load some (pre-processed) MRI data, select a model, and let MDT fit your selected model to the data. The optimized parameter maps are written as nifti files together with variance and covariance estimates and possibly other output maps.

For easy of use with population studies, MDT includes *pipelines* for the Human Connectome Project (HCP) study, which automatically detects the directory layout of the HCP MGH and HCP WuMinn studies and fit your desired model to (a subset of) the data.

Single dataset fitting is accessible via three interfaces, the *Graphical User Interface (GUI)*, *Command Line Interface (CLI)* and/or directly using the *Python interface*.

3.1 HCP Pipeline

MDT comes pre-installed with Human Connectome Project (HCP) compatible pipelines for the MGH and the WuMinn 3T studies. To run, please change directory to where you downloaded your (pre-processed) HCP data (MGH or WuMinn) and execute:

```
$ mdt-batch-fit . NODDI
```

and it will autodetect the study in use and fit your selected model to all the subjects.

For a list of all available models, run the command `cli_index_mdt-list-models`.

3.2 MDT example data

MDT comes pre-loaded with some example data that allows you to quickly get started using the software. This example data can be obtained in the following ways:

- **GUI:** Open the model fitting GUI and find in the menu bar: “Help -> Get example data”.
- **Command line:** Use the command `cli_index_mdt-get-example-data`:

```
$ mdt-get-example-data .
```

- **Python API:** Use the function `mdt.utils.get_example_data()`:

```
import mdt
mdt.get_example_data('/tmp')
```

There are two MDT example datasets, a *blk_b2k* dataset and a *multishell_b6k_max* dataset, both acquired in the same session on a Siemens Prisma system, on the VE11C software line, with the standard product diffusion sequence at 2mm isotropic with GRAPPA in-plane acceleration factor 2 and 6/8 partial fourier (no multi-band/simultaneous multi-slice).

The *blk_b2k* has a shell of $b=1000\text{s/mm}^2$ and of $b=2000\text{s/mm}^2$ and is very well suited for e.g. Tensor, Ball&Stick and NODDI. In this, the $b=1000\text{s/mm}^2$ shell is the standard Jones 30 direction table, including 6

b0 measurements at the start. The $b=2000\text{s/mm}^2$ shell is a 60 whole-sphere direction set created with an electrostatic repulsion algorithm and has another 7 b0 measurements, 2 at the start of the shell and then one every 12 directions.

The *multishell_b6k_max* dataset has 6 b0's at the start and a range of 8 shells between $b=750\text{s/mm}^2$ and $b=6000\text{s/mm}^2$ (in steps of 750s/mm^2) with an increasing number of directions per shell (see [De Santis et al., MRM, 2013](#)) and is well suited for CHARMED analysis and other models that require high b-values (but no diffusion time variations).

3.3 Graphical interface

One of the ways to use MDT for model analysis is by using the Graphical User Interface (GUI). To launch the GUI in Linux and OSX, please open a console and type `mdt-gui` or `MDT` to launch the analysis GUI. In Windows one can either open an Anaconda prompt and type `mdt-gui` or `MDT` or, alternatively, one can type `mdt-gui` or `MDT` in the search bar under the start button to find and launch the GUI.

The following is an example of the GUI running in Linux:

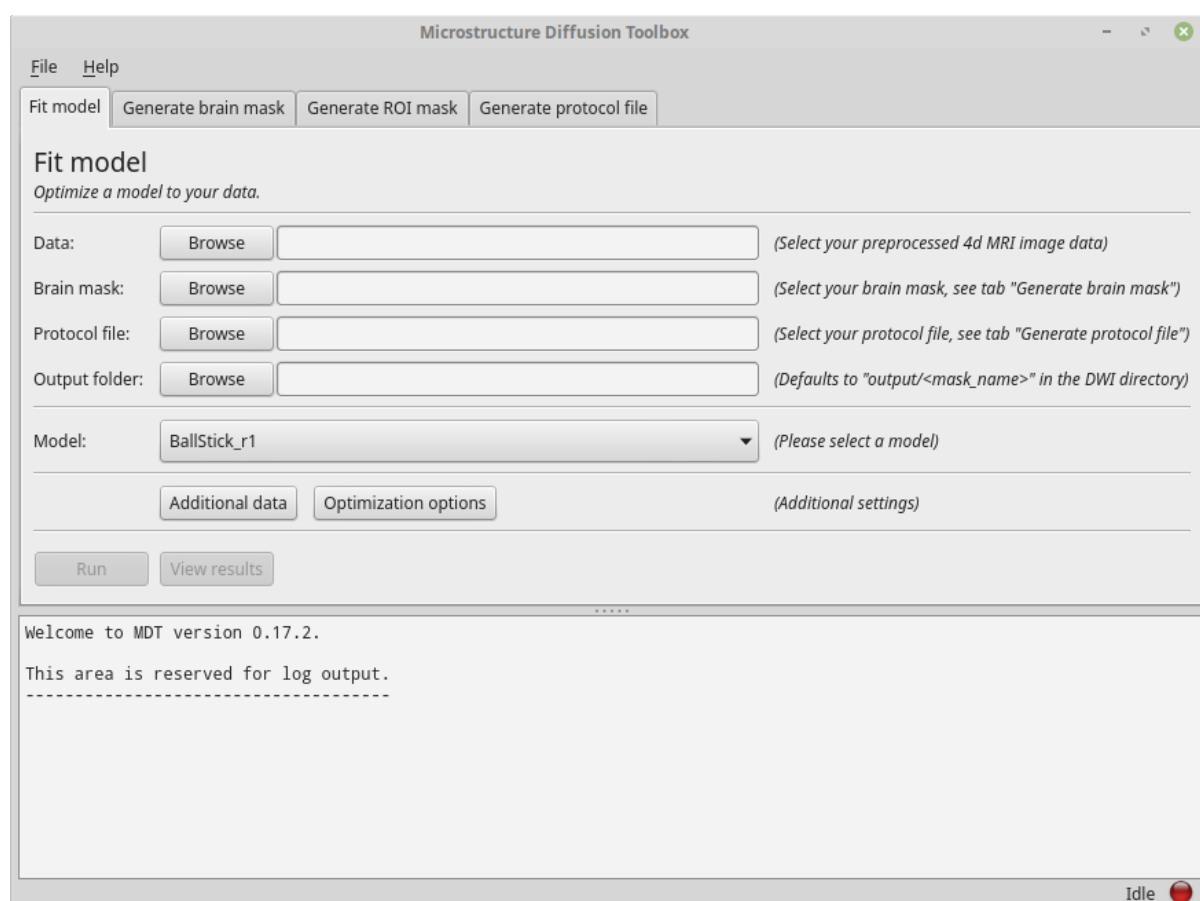


Fig. 1: A screenshot of the MDT GUI in Linux.

Using the GUI is a good starting point for model analysis since it guides you through the steps needed for the model analysis. In addition, as a service to the user, the GUI writes Python and Bash script files for most of the actions performed in the GUI. This allows you to use the GUI to generate a coding template that can be used for further processing.

Creating a protocol file

As explained in [Protocol](#), MDT stores all the acquisition settings relevant for the analysis in a Protocol file. To create one using the GUI, please go to the tab “Generate protocol file”. On the bottom of the tab you can find the button “Load g & b” which is meant to load a b-vec and b-val file into the GUI. Please click the button and, for the sake of this example, load from the MDT example data folder the b-vec and b-val file of the b1k_b2k dataset. This tab should now look similar to this example:

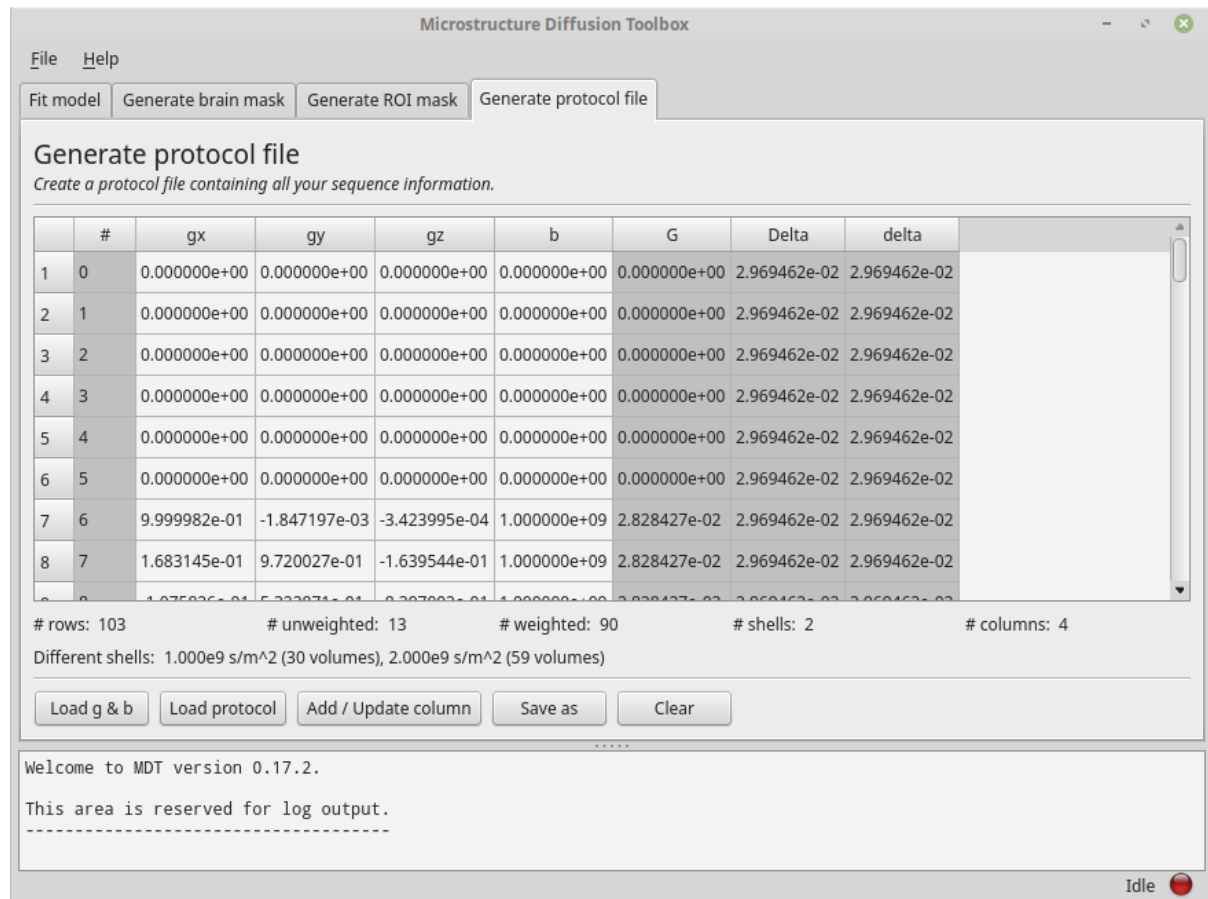


Fig. 2: The Protocol tab after loading a bvec and bval file.

Having loaded a b-vec/b-val pair (or a Protocol file), you are presented with a tabular overview of your protocol, with some basic statistics below the table. The table shows you per volume (rows) the values for each of the columns. Columns in gray are automatically calculated or estimated from the other columns. Note that these derived values are there for your convenience and as a check on protocol validity, but cannot be assumed to be strictly correct. For example, in the screenshot above, G , Δ and δ are estimated from the b -values by assuming $\Delta == \delta$ (this approximation is taken from the NODDI matlab toolbox to be consistent with previous work). Since in reality $\Delta \approx \delta + \text{refocussing RF-pulse length in PGSE}$, this will underestimate both δ and G . The gray columns are not part of the protocol file and will not be saved.

To add or update a column you can use the dialog under the button “Add / update column”. To remove a column, right click the column header and select the “Remove column” option.

For the sake of this example, please add to the loaded b-vec and b-val files the “Single value” columns “Delta”, “delta”, “TE” and “TR” with values 26.5e-3 seconds, 16.2e-3 seconds, 60e-3 and 7.1 seconds respectively. Having done so, the gray columns for δ and Δ should now turn white (as they no longer are estimated but are actually provided). Your screen should now resemble the following example:

As an additional check, you could save the protocol as “b1k_b2k_test.prct1” and compare it to the pre-supplied protocol for comparison (open both in a separate GUI). Alternatively, you could save the file and open with a text editor to study the layout of the protocol file.

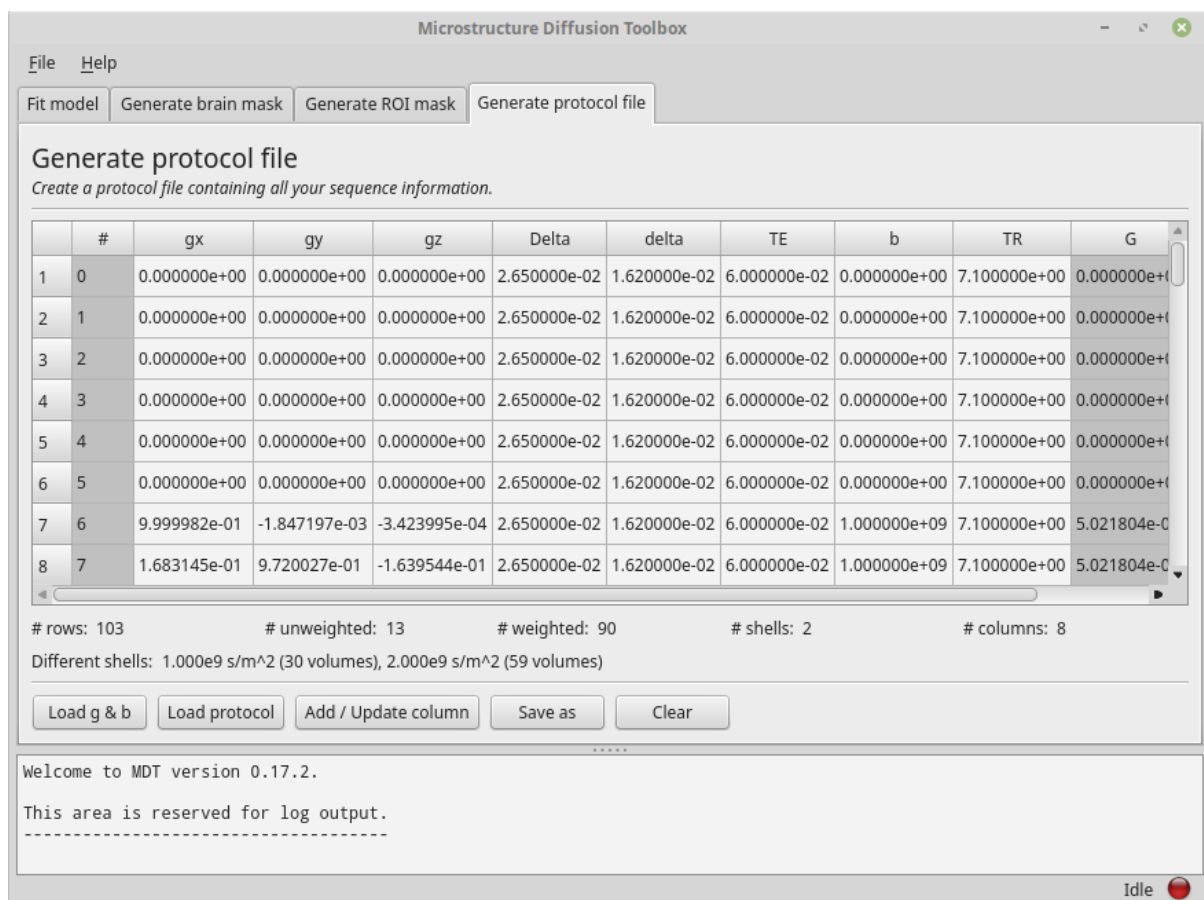


Fig. 3: The Protocol tab after adding various columns.

Generating a brain mask

MDT has some rough functionality for creating a brain mask, similar to the `median_otsu` algorithm in Dipy. This algorithm is not as sophisticated as for example BET in FSL, therefore we will not go in to much detail here. The mask generating functionality in MDT is merely meant for quickly creating a mask within MDT.

Since the MDT example data comes pre-supplied with a mask (generated by BET), we won't cover mask generation here. Also, the process is fairly straightforward by just supplying a DWI volume and a protocol.

Generating a ROI mask

It is sometimes convenient to run analysis on a single slice (Region Of Interest) before running it whole brain. Using the tab "Generate ROI mask" it is possible to load a whole brain mask and create a new mask where only one slice is used. This ROI mask is just another mask with even more voxels masked.

We do not need this step for the MDT example slices since that dataset is already compressed to two slices.

NODDI estimation example

With a protocol and mask ready we can now proceed with model analysis. The first step is to check which devices we are currently using. Please open the runtime settings dialog using the menu bar (typically on the top of the GUI, File -> Runtime settings). This dialog will resemble the following example except that the devices listed will match your system configuration:

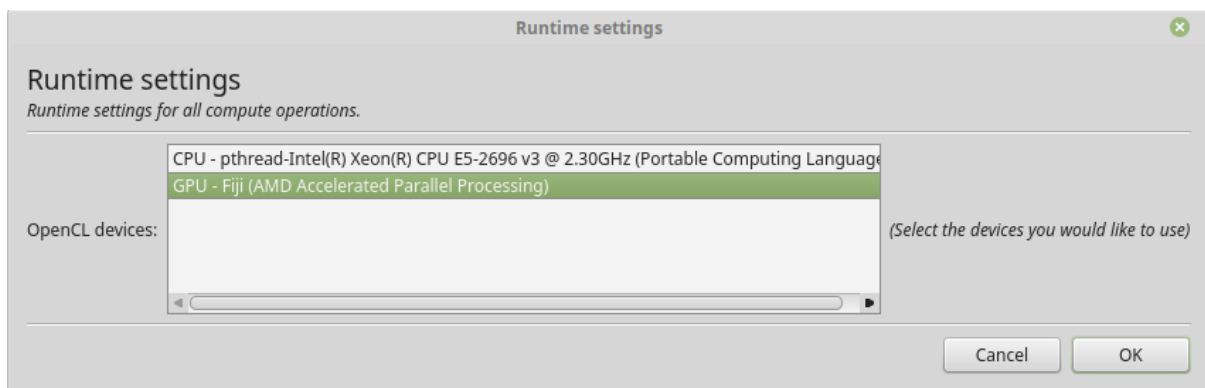


Fig. 4: The runtime settings showing the devices MDT can use for the computations.

Typically you only want to select one or all of the available GPU's (Graphical Processing Units) since they are faster. In contrast, on Apple / OSX the recommendation is to use the CPU since the OpenCL drivers by Apple crash frequently.

Having chosen the device(s) to run on, please open the tab "Fit model" and fill in the fields using the "b1k_b2k" dataset as an example. The drop down menu shows the models MDT can use.

Having filled in all the required fields, select the "NODDI" model, and press "Run". MDT will now compute your selected model on the data. Please note that for some models, MDT will first compute another model to serve as initialization for your selected model. For instance, when running NODDI, MDT first estimates the BallStick_r1 model to use as initialization for the NODDI model. When the calculations are finished you can click the "View results" button to launch the MDT map viewer GUI for visually inspecting the results. See [Maps visualizer](#) for more details on this visualizer.

By default MDT returns a lot of result maps, like various error maps and additional maps like FSL like vector component maps. All these maps are in nifti format (.nii) and can be viewed and opened in any compatible viewer like for example `fslview` or the [Maps visualizer](#).

In addition to the results, MDT also writes a Python and a Bash script file to a "script" directory next to your DWI file. These script files allow you to reproduce the model fitting using a Python script file or command line.

Estimating any model

In general, using the GUI, estimating any model is just a matter of selecting the right model and clicking the run button. Please be advised though that some models require specific protocol values to be present. For example, the CHARMED models requires that the “TE” is specified in the protocol or as a protocol map. MDT will help you by warning you if the available data is not suited for the selected model.

For adding additional data, like protocol maps, a noise standard deviation or a gradient deviations map you can use the button “Additional data”.

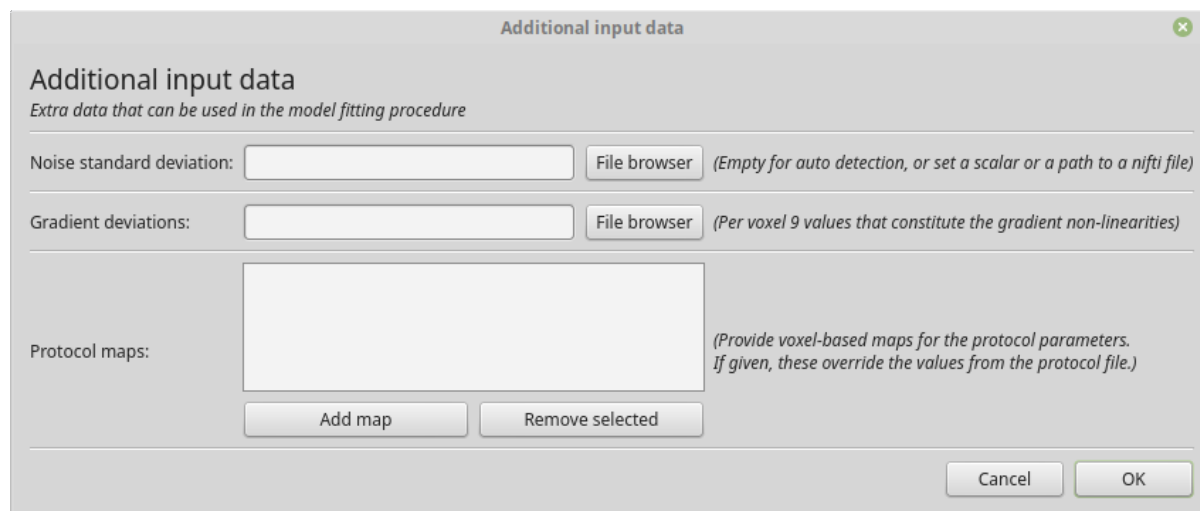


Fig. 5: The dialog for adding additional input data.

If you are providing the gradient deviations map, please be advised that this uses the standard set by the HCP Wuminn consortium.

The button “Optimization options” allows you to set some of the *Available optimization routines*.

3.4 Command Line interface

After installation a few command line functions are installed to your system. These commands, prefixed with `mdt-`, allow you to use various functionality of MDT using the command line. For an overview of the available commands, please see: `cli_index`.

Creating a protocol file

As explained in *Protocol*, MDT stores all the acquisition settings relevant for the analysis in a Protocol file. To create one using the command line, you can use the command `cli_index_mdt-create-protocol`. The most basic usage is to create a protocol file from a b-vec and b-val file:

```
$ mdt-create-protocol data.bvec data.bval
```

which will generate a protocol file named “data.prctl”. For a more sophisticated protocol, one can add additional columns using the `--<column_name> <value>` syntax. For example:

```
$ mdt-create-protocol d.bvec d.bval --Delta 26.5 --delta delta.txt
```

which will add both the column `Delta` to your protocol file (with a static value of 26.5 ms) and the column `delta` which is read from a file. If a file is given it can either contain a column, row or scalar.

If you have already generated a protocol file and wish to change it you can use the `cli_index_mdt-math-protocol` command. This command allows you to change a protocol file using an expression. For example:

```
$ mdt-math-protocol p.prctl 'G *= 1e-3; TE = 60e-3; del(TR)' -o new.prctl
```

this example command scales G, adds (or replaces) TE and deletes the column TR from the input protocol file and writes the results to a new protocol file.

An example usage in the case of the MDT example data would be the command:

```
$ cd blk_b2k
$ mdt-create-protocol blk_b2k.bvec blk_b2k.bval \
  --Delta 26.5 \
  --delta 16.2 \
  --TE 60 \
  --TR 7100 \
```

note that by default the sequence timings are in ms for this function and the elements Delta, delta, TE and TR will automatically be scaled and stored as seconds.

Creating a brain mask

MDT has some rough functionality for creating a brain mask, similar to the `median_otsu` algorithm in Dipy. This algorithm is only meant for generating a rough brain mask and is not as sophisticated as for example BET from FSL.

Creating a mask is possible with the command `cli_index_mdt-create-mask`:

```
$ mdt-create-mask data.nii.gz data.prctl
```

which generates a mask named `data_mask.nii.gz`.

Generating a ROI mask

It is sometimes convenient to run analysis on a single slice (Region Of Interest) before running it whole brain. For the example data we do not need this step since that dataset is already compressed to two slices.

To create a ROI mask for your own data you can either use the `cli_index_mdt-create-roi-slice` command or the `cli_index_mdt-math-img` command. An example with the `cli_index_mdt-create-roi-slice` would be:

```
$ mdt-create-roi-slice mask.nii.gz -d 2 -s 30
```

here we generate a mask in dimension 2 on index 30 (0-based).

The other way of generating a mask is by using the `cli_index_mdt-math-img` command, as a similar example to the previous one:

```
$ mdt-math-img mask.nii.gz 'a[..., 30]' -o mask_2_30.nii.gz
```

Also note that since `cli_index_mdt-math-img` allows general expressions on nifti files, it can also generate more complex ROI masks.

NODDI estimation example

Model fitting using the command line is made easy using the `cli_index_mdt-model-fit` command. Please see the reference manual for all switches and options for the model fit command.

The basic usage is to fit for example NODDI on a dataset:

```
$ cd blk_b2k
$ mdt-model-fit NODDI \
  blk_b2k_example_slices_24_38.nii.gz \
```

(continues on next page)

(continued from previous page)

```
blk_b2k.prctcl \
*mask.nii.gz
```

This command needs at least a model name, a dataset, a protocol and a mask to function. Please note that for some models, MDT will first compute another model to serve as initialization for your selected model. For instance, when running NODDI, MDT first estimates the BallStick_r1 model to use as initialization for the NODDI model. For a list of supported models, please run the command `cli_index_mdt-list-models`.

When the calculations are done you can use the MDT maps visualizer (`cli_index_mdt-view-maps`) for viewing the results:

```
$ mdt-view-maps output/BallStick_r1
```

For more details on the MDT maps visualizer, please see the chapter [Maps visualizer](#).

Estimating any model

In principle every model in MDT can be fitted using the `cli_index_mdt-model-fit`. Please be advised though that some models require specific protocol values to be present. For example, the CHARMED models requires that the “TE” is specified in your protocol. MDT will warn you if the available data is not suited for the selected model.

Just as in the GUI, it is possible to add additional data like protocol maps, a noise standard deviation or a gradient deviations map to the model fit command. Please see the available switches of the `cli_index_mdt-model-fit` command.

3.5 Python interface

The most direct method to interface with MDT is by using the Python interface. Most actions in MDT are accessible using the `mdt` namespace, obtainable using:

```
import mdt
```

When using MDT in an interactive shell you can use the default `dir` and `help` commands to get more information about the MDT functions. For example:

```
>>> import mdt
>>> dir(mdt) # shows the functions in the MDT namespace
...
>>> help(mdt.fit_model) # shows the documentation a function
...
```

Creating a protocol file

As explained in [Protocol](#), MDT stores all the acquisition settings relevant for the analysis in a Protocol file. The simplest way of creating a Protocol is by using the function `create_protocol()` to create a Protocol file and object.

To (re-)create the protocol file for the `blk_b2k` dataset you can use the following command:

```
protocol = mdt.create_protocol(
    bvecs='blk_b2k.bvec', bvals='blk_b2k.bval',
    out_file='blk_b2k.prctcl',
    Delta=26.5e-3, delta=16.2-3, TE=60e-3, TR=7.1)
```

Please note that the Protocol class is a singleton and adding or removing columns involves a copy operation. Also note that we require the columns to be in **SI units**.

Generating a brain mask

MDT has some rough functionality for creating a brain mask, similar to the `median_otsu` algorithm in Dipy. This algorithm is not as sophisticated as for example BET in FSL, therefore we will not go in to much detail here. The mask generating functionality in MDT is merely meant for quickly creating a mask within MDT.

Creating a mask with the MDT Python interface can be done using the function `create_median_otsu_brain_mask()`. For example:

```
mdt.create_median_otsu_brain_mask(
    'blk_b2k_example_slices_24_38.nii.gz',
    'blk_b2k.prtcl',
    'data_mask.nii.gz')
```

which generates a mask named `data_mask.nii.gz`.

Generating a ROI mask

It is sometimes convenient to run analysis on a single slice (Region Of Interest) before running it whole brain. For the example data we do not need this step since that dataset is already compressed to two slices.

Since we are using the Python interface we can use any Numpy slice operation to cut the data as we please. An example of operating on a nifti file is given by:

```
nifti = mdt.load_nifti('mask.nii.gz')
data = nifti.get_data()
header = nifti.header

roi_slice = data[:, :, 30]

mdt.write_nifti(roi_slice, header, 'roi_mask.nii.gz')
```

this generates a mask in dimension 2 on index 30 (be wary, Numpy and hence MDT use 0-based indexing).

NODDI estimation example

For model fitting you can use the `fit_model()` command. This command allows you to optimize any of the models in MDT given only a model, input data and output folder.

The basic usage is to fit for example NODDI on a dataset:

```
input_data = mdt.load_input_data(
    '../blk_b2k/blk_b2k_example_slices_24_38',
    '../blk_b2k/blk_b2k.prtcl',
    '../blk_b2k/blk_b2k_example_slices_24_38_mask')

inits = mdt.get_optimization_inits('NODDI', input_data, 'output')

mdt.fit_model('NODDI', input_data, 'output',
              initialization_data={'inits': inits})
```

First, we load the input data (see `load_input_data()`) with all the relevant modeling information. Second, we try to find a good starting position for our model using the `mdt.get_optimization_inits()` command. This function returns a dictionary with initialization, fixation and boundary condition information which is suitable for your model and data. Please note that this function only works for models that ship by default with MDT, but due to the general nature of the `initialization_data` attribute of the `fit_model()` function you can easily generate your own initialization values.

By default, this model fit function tries to initialize the model with a good starting point using the function `mdt.get_optimization_inits()`. You can also disable this feature by setting `use_cascaded_inits` to False, i.e.:

```
mdt.fit_model('NODDI', input_data, 'output',
              use_cascaded_inits=False)
```

For precise control of the initialization, you can also find the starting point yourself and initialize the model manually using:

```
input_data = ...

inits = mdt.get_optimization_inits('NODDI', input_data, 'output')

mdt.fit_model('NODDI', input_data, 'output',
              use_cascaded_inits=False,
              initialization_data={'inits': inits})
```

When the calculations are done you can use the MDT maps visualizer for viewing the results:

```
mdt.view_maps('../blk_b2k/output/BallStick_r1')
```

Full example

To summarize the code written above, here is a complete and ease of use MDT model fitting example:

```
import mdt

protocol = mdt.create_protocol(
    bvecs='blk_b2k.bvec', bvals='blk_b2k.bval',
    out_file='blk_b2k.prtcl',
    Delta=26.5e-3, delta=16.2-3, TE=60e-3, TR=7.1)

input_data = mdt.load_input_data(
    'blk_b2k_example_slices_24_38',
    'blk_b2k.prtcl',
    'blk_b2k_example_slices_24_38_mask')

mdt.fit_model('NODDI', input_data, 'output')
```

Estimating any model

In principle every model in MDT can be fitted using the model fitting routines. Please be advised though that some models require specific protocol values to be present. For example, the CHARMED models requires that the “TE” is specified in your protocol. MDT will help you by warning you if the available data is not suited for the selected model.

To add additional data to your model computations, you can use the additional keyword arguments to the `load_input_data()` command.

Fixing parameters

To fix parameters, as for example fibre orientation parameters, one can use the `initialization_data` keyword of the `fit_model()` command. This keyword allows fixing and initializing parameters just before model optimization and sampling. The following example shows how to fix the fibre orientation parameters of the NODDI model during optimization:

```
theta, phi = <some function to generate angles>

mdt.fit_model('NODDI',
    ...
```

(continues on next page)

(continued from previous page)

```

initialization_data={
    'fixes': {'NODDI_IC.theta': theta,
             'NODDI_IC.phi': phi}
})

```

The syntax of the `initialization_data` is:

```
initialization_data = {'fixes': {...}, 'inits': {...}}
```

where both `fixes` and `inits` are dictionaries with model parameter names mapping to either scalars or 3d/4d volumes. The `fixes` indicates parameters that will be fixed to those values, which will actively exclude those parameters from optimization. The `inits` indicate initial values (starting position) for the parameters.

3.6 Available optimization routines

MDT features GPU accelerated optimization routines, carefully selected for their use in microstructure MRI imaging [Harms2017]. For reusability, these optimization routines are available in the [Multithreaded Optimization Toolbox \(MOT\)](#). This optimization toolbox contains various optimization routines programmed in OpenCL for optimal performance on both CPU and GPU. The following optimization routines are available to MDT:

- Powell [Powell1964] ([wiki](#))
- Levenberg-Marquardt [Levenberg1944], ([wiki](#))
- Nelder-Mead simplex [Nelder1965], ([wiki](#))
- Subplex [Rowan1990]

the number of iterations of each of these routines can be controlled by a variable named `patience` which sets the number of iterations to $p * (n + 1)$ where p is the patience and n the number of parameters to be fitted. By setting the patience indirectly we can implicitly account for model complexity.

By default, MDT uses the Powell routine for all models, although with some models a different routine might perform better for a specific model. We advice experimenting with the different routines to see which performs best for your model.

Setting the routine

From within the graphical interface, the options can be set using the *Optimization Options* button in the model fit tab. Showing you a frame similar to:

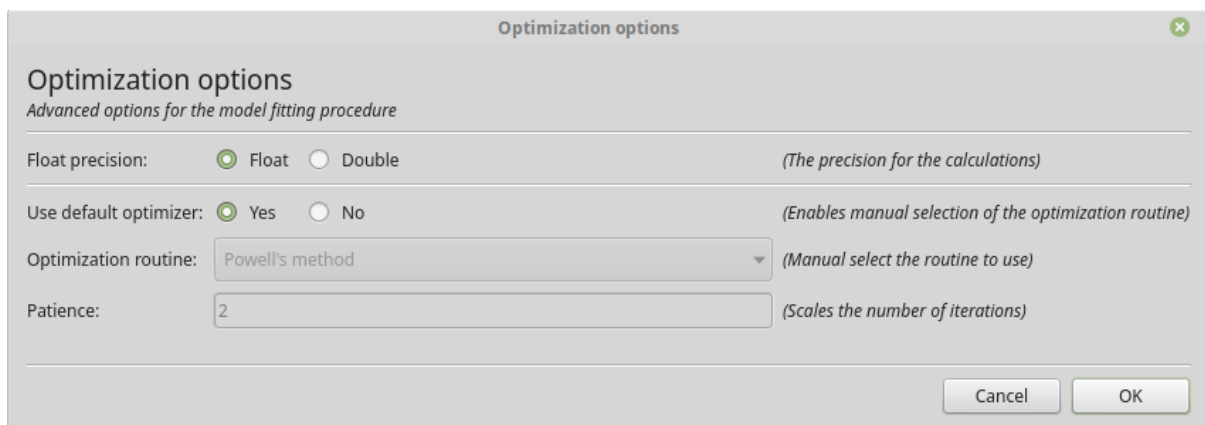


Fig. 6: The graphical dialog to setting the optimization options.

Using the command line the same options can be set using `--method` and `--patience`, as:

```
$ cd blk_b2k
$ mdt-model-fit \
  ... \
  --method Powell
  --patience 2
```

Using the Python API the options are specified as:

```
mdt.fit_model(
    ...,
    method='Powell',
    optimizer_options={'patience': 2}
)
```

Chapter 4

MCMC sampling

MDT supports Markov Chain Monte Carlo (MCMC) sampling of all models as a way of recovering the full posterior density of model parameters given the data. While model fitting provides you only with a maximum likelihood estimate and a standard deviations using the Fisher Information Matrix, MCMC sampling approximates the full posterior distribution by drawing many samples of the parameters.

In contrary to model fitting, model sampling is currently only available using the Python function `mdt.sample_model()`.

The main workflow is similar to the model fitting in that that you load some (pre-processed) MRI data, select a model, and let MDT do the sampling for you. In most cases you will also want to initialize the sampling routine using a better initialization point, which is typically provided by the model fitting.

The common workflow is to first optimize the model using the model fitting routines and use that as a starting point for the MCMC sampling. A full example would be:

```
import mdt

input_data = mdt.load_input_data(
    '../b1k_b2k/b1k_b2k_example_slices_24_38',
    '../b1k_b2k/b1k_b2k.prtcl',
    '../b1k_b2k/b1k_b2k_example_slices_24_38_mask')

inits = mdt.get_optimization_inits('NODDI', input_data, 'output')

mle = mdt.fit_model('NODDI', input_data, 'output',
                    initialization_data={'inits': inits})

samples = mdt.sample_model(
    'NODDI',
    input_data,
    'output',
    nmr_samples=1000,
    burnin=0,
    thinning=0,
    initialization_data={'inits': mle}
)
```

here, we load the input data, get a suitable optimization starting point, fit the NODDI model and then finally use that as a starting point for the MCMC sampling.

4.1 Post-processing

Instead of storing the samples and post-processing the results afterwards, it is also possible to let MDT do some post-processing while it still has the samples in memory. For example, when you are low on disk space you could do all the post-processing directly after sampling such that you have all your statistic maps without storing all the samples. Common post-processing options are available using the `post_processing` argument to the `mdt.sample_model()` function. More advanced (custom) functionality can be called using the `post_sampling_cb` argument.

Common post-processing

Common post-processing, with their defaults are given by:

```
mdt.sample_model(  
    ...  
    post_processing={  
        'univariate_ess': False,  
        'multivariate_ess': False,  
        'maximum_likelihood': False,  
        'maximum_a_posteriori': False,  
        'average_acceptance_rate': False,  
        'model_defined_maps': True,  
        'univariate_normal': True  
    }  
)
```

The individual options are:

- *Univariate ess*: the Effective Sample Size for every parameter
- *Multivariate ess*: a multivariate Effective Sample Size using all parameters
- *Maximum likelihood*: take (per voxel) the samples with the highest log-likelihood
- *Maximum a posteriori*: take (per voxel) the samples with the highest posterior (log-likelihood times log-prior)
- *Average acceptance rate*: compute the average acceptance rate per parameter
- *Model defined maps*: output the additional maps defined in the model definitions
- *Univariate normal*: output a mean and standard deviation per model parameter

Custom post-processing

In addition to the common post-processing options, MDT also allows you to specify a generic callback function for post-processing. This function takes as input the sampling results and the current model and should output a (nested) dictionary with output maps / folders (when nested).

As an example, to replicate the `maximum_a_posteriori` post-processing option we can use:

```
def maximum_a_posteriori(sampling_output, composite_model):  
    from mdt.utils import split_array_to_dict  
  
    samples = sampling_output.get_samples()  
    log_likelihoods = sampling_output.get_log_likelihoods()  
    log_priors = sampling_output.get_log_priors()  
  
    posteriors = log_likelihoods + log_priors  
  
    map_indices = np.argmax(posteriors, axis=1)  
    map_samples = samples[range(samples.shape[0]), :, map_indices]  
  
    result_maps = split_array_to_dict(  
        map_samples,  
        composite_model.get_free_param_names()  
    )  
  
    return {'my_post_processing': result_maps}  
  
mdt.sample_model(  
    ...  
    post_sampling_cb=maximum_a_posteriori  
)
```

this will save all the maps from `result_maps` to a sub-directory called `my_post_processing`.

Chapter 5

Adding components

Components are a modular sub-system of MDT allowing you to add models and other functionality to MDT.

On the moment MDT supports the following components:

- *Composite models*: the models fitted by MDT, these are built out of the compartment models
- *Compartment models*: reusable models of diffusion and relaxometry models
- *Parameters*: definitions of common parameters shared between models
- *Library functions*: library functions for use in composite models
- *Batch profiles*: for the batch functionality in MDT

5.1 Defining components

There are two ways of adding or updating components in MDT, *Global definitions*, by adding a component to your configuration folder or *Dynamic definitions* by defining it dynamically in your modeling scripts.

Global definitions

For persistent model definitions you can use the `.mdt` folder in your home folder. This folder contains diffusion MRI models and other functionality that you can extend without needing to reinstall or recompile MDT.

The `.mdt` folder contains, for every version of MDT that existed on your machine, a directory containing the configuration files and a folder with the dynamically loadable modules. A typical layout of the `.mdt` directory is:

- **.mdt/**
 - **<version>/**
 - * `mdt.default.conf`
 - * `mdt.conf`
 - * `components/`

The configuration files are discussed in *Advanced configuration*, the components folder is used for the global model definitions.

The components folder consists of two sub-folders, *standard* and *user*, with an identical folder structure for the contained modules:

- **components/**
 - **standard**
 - * `compartment_models`
 - * `composite_models`
 - * `...`
 - **user**
 - * `compartment_models`

```
* composite_models  
* ...
```

By editing the contents of these folders, the user can add, extend and/or remove functionality globally and persistently. The folder named *standard* contains modules that come pre-supplied with MDT. These modules can change from version to version and any change you make in in this folder will be lost after an update. To make persistent changes you can add your modules to the *user* folder. The content of this folder is automatically copied to a new version.

Dynamic definitions

Alternatively, it is also possible to define components on the fly in your analysis scripts. This is as simple as defining a template in your script prior to using it in your analysis. For example, prior to calling the fit model function, you can define a new model as:

```
from mdt import CompositeModelTemplate  
  
class BallZeppelin(CompositeModelTemplate):  
    model_expression = '''  
        S0 * ( (Weight(w_csf) * Ball) +  
                (Weight(w_res) * Zeppelin) )  
    '''  
  
mdt.fit_model('BallZeppelin', ...)
```

It is also possible to overwrite existing models on the fly, for example:

```
import mdt  
  
class Tensor(mdt.get_template('composite_models', 'Tensor')):  
    likelihood_function = 'Rician'  
  
mdt.fit_model('Tensor', ...)
```

Breaking this up, in the first part:

```
class Tensor(mdt.get_template('composite_models', 'Tensor')):  
    likelihood_function = 'Rician'
```

we load the last available Tensor model template from MDT (using `get_template('composite_models', 'Tensor')`) and use it as a basis for an updated template. Then, since this class is also named Tensor (by saying `class Tensor(...)`) this new template will override the previous Tensor. The body of this template then updates the previous Tensor, in this case by changing the likelihood function.

In the second part:

```
mdt.fit_model('Tensor', ...)
```

we just call `mdt.fit_model` with as model `Tensor`. MDT will then load the model by taking the last known definitions. As such, the new `Tensor` model with the updated likelihood function will be used in the model fitting.

To remove an entry, you can use, for example:

```
mdt.remove_last_entry('composite_models', 'Tensor')
```

This functionality allows you to overwrite and add models without adding them to your home folder.

5.2 Composite models

The composite models, or, multi-compartment models, are the models that MDT actually optimizes. Composite models are formed by a combination / composition, of compartment models.

When asked to optimize (or sample) a composite model, MDT combines the CL code of the compartments into one objective function and combines it with a likelihood function (Rician, OffsetGaussian, Gaussian). Since the compartments already contain the CL code, no further CL modeling code is necessary in the multi-compartment models.

Composite models are defined by inheriting from `CompositeModelTemplate`. The following is an minimal example of a composite (multi-compartment) model in MDT:

```
class BallStick_r2(CompositeModelTemplate):

    model_expression = '''
        S0 * ( (Weight(w_ball) * Ball) +
                (Weight(w_stick0) * Stick(Stick0)) +
                (Weight(w_stick1) * Stick(Stick1)) )
    '''
```

The model expression is a string that expresses the model in a MDT model specific mini-language. This language, which only accepts the operators `*`, `/`, `+` and `-` can be used to combine your compartments in any way possible (within the grammar of the mini-language). MDT parses this string, loads the compartments from the compartment models and uses the CL code of these compartments to create the CL objective function for your complete composite model.

The example above combines the compartments (`Ball` and `Stick`) as a weighted summation using the special compartment `Weight` for the compartment weighting (these weights are sometimes called volume fractions).

The example also shows compartment renaming. Since it is possible to use a compartment multiple times, it is necessary to rename the double compartments to ensure that all the compartments have a unique name. This renaming is done by specifying the nickname in parenthesis after the compartment. For example `Stick(Stick0)` refers to a `Stick` compartment that has been renamed to `Stick0`. This new name is then used to refer to that specific compartment in the rest of the composite model attributes.

The composite models have more functionality than what is shown here. For example, they support parameter dependencies, initialization values, parameter fixations and protocol options.

Parameter dependencies

Parameter dependencies make explicit the dependency of one parameter on another. For example, some models have both an intra- and an extra-axonal compartment that both feature the `theta` and `phi` fibre orientation parameters. It could be desired that these angles are exactly the same for both compartments, that is, that they both reflect the exact same fibre orientation. One possibility to solve this would be to create a new compartment having the features of both the intra- and the extra-axonal compartment. This however lowers the reusability of the compartments. Instead, one could define parameter dependencies in the composite model. For example:

```
class NODDI(CompositeModelTemplate):

    ...
    fixes = {
        ...
        'Ball.d': 3.0e-9,
        'NODDI_EC.dperp0': 'NODDI_EC.d * (w_ec.w / (w_ec.w + w_ic.w))',
        'NODDI_EC.kappa': 'NODDI_IC.kappa',
        'NODDI_EC.theta': 'NODDI_IC.theta',
        'NODDI_EC.phi': 'NODDI_IC.phi'
    }
```

In this example, we used the attribute `fixes` to specify dependencies and parameter fixations. The attribute `fixes` accepts a dictionary with as key the name of the parameter and as value a scalar, a map or a dependency. The dependency can either be given as a string or as a dependency object.

In the example above we added two simple assignment dependencies in which the theta and phi of the NODDI_EC compartment are locked to that of the NODDI_IC compartment. This dependency locks the NODDI_EC theta and phi to that of NODDI_IC assuring that both the intra cellular and extra cellular models reflect the same orientation.

Weights sum to one

Most composite models consist of a weighted sum of compartments models. An implicit dependency in this set-up is that those weights must exactly sum to one. To ensure this, MDT adds, by default, a dependency to the last Weight compartment in the composite model definition. This dependency first normalizes (if needed) the n-1 Weight compartments by their sum $s = \sum_i^{n-1} w_i$. Then, the last Weight, which is not optimized explicitly, is then either set to zero, i.e. $w_n = 0$ or set as $w_n = 1 - s$ if s is smaller than zero.

If you wish to disable this feature, for example in a model that does not have a linear sum of weighted compartments, you can use set the attribute `enforce_weights_sum_to_one` to false, e.g.:

```
class MyModel(CompositeModelTemplate):
    ...
    enforce_weights_sum_to_one = False
```

Protocol options

It is possible to add dMRI volume selection to a composite model using the “protocol options”. These protocol options allow the composite model to select, using the protocol, only those volumes that it can use for optimization. For example, the Tensor model is defined to work with b-values up to 1500 s/mm², yet the user might be using a dataset that has more shells, with some shells above the b-value threshold. To prevent the user from having to load a separate dataset for the Tensor model and another dataset for the other models, we implemented in MDT model protocol options. This way, the end user can provide the whole protocol file and the models will pick from it what they need.

Please note that these volume selections only work with columns in the protocol, not with the `extra_protocol` maps.

There are two ways to enable this mechanism in your composite model. The first is to add the `volume_selection` directive to your model:

```
class Tensor(CompositeModelTemplate):
    ...
    volume_selection = {'b': [(0, 1.5e9 + 0.1e9)]}
```

This directive specifies that we wish to use a subset of the weighted volumes, that is, a single b-value range with b-values between `b=0` and `b=1.5e9 s/m2`. Each key in `volume_selection` should refer to a column in the protocol file and each value should be a list of ranges.

The second method is to add the bound function `_get_suitable_volume_indices` to your model definition. For example:

```
...
from mdt.component_templates.base import bind_function

class Tensor(CompositeModelTemplate):
    ...

    @bind_function
    def _get_suitable_volume_indices(self, input_data):
        return protocol.get_indices_bval_in_range(start=0, end=1.5e9 + 0.1e9)
```

This function should then return a list of integers specifying the volumes (and therefore protocol rows) you wish to use in the analysis of this model. To use all volumes you can use something like this:

```
@bind_function
def _get_suitable_volume_indices(self, input_data):
    return list(range(input_data.protocol.length))
```

Extra result maps

It is also possible to add additional parameter maps to the fitting and sampling results. These maps are meant to be forthcoming to the end-user by providing additional maps to the output. Extra results maps can be added by both the composite model as well as by the compartment models.

Just as with compartment models, one can add extra output maps to the optimization results and to the sampling results as:

```
class MyModel(CompositeModelTemplate):
    ...
    extra_optimization_maps = [
        lambda results: ...
    ]

    extra_sampling_maps = [
        lambda samples: ...
    ]
```

where each callback function should return a dictionary with extra maps to add to the output.

Likelihood functions

Models are optimized by finding the set of free parameter values $x \in R^n$ that minimize the likelihood function of the modeling errors ($O - S(x)$) with O the observed data and $S(x)$ the model signal estimate. In diffusion MRI the common likelihood models are the *Gaussian*, *Rician* and *OffsetGaussian* models. Each has different characteristics and implements the modeling ($O - S(x)$) in a slightly different way. Following (Harms 2017) we use, by default, the Offset Gaussian likelihood model for all models. To change this to another likelihood model for one of your models you can override the `likelihood_function` attribute, for example:

```
class MyModel(CompositeModelTemplate)
    ...
    likelihood_function = 'Rician'
```

By default the `likelihood_function` attribute is set to `OffsetGaussian`. The likelihood function can either be defined as a string or as an object. Using a string, the possible options are `Gaussian`, `OffsetGaussian` and `Rician`. Using an object, you must provide an instance of `mdt.model_building.likelihood_functions.LikelihoodFunction`. For example:

```
...
from mdt.model_building.likelihood_functions import RicianLikelihoodFunction

class MyModel(CompositeModelTemplate)
    ...
    likelihood_function = RicianLikelihoodFunction()
```

All listed likelihood functions require a standard deviation σ representing the noise in the input data. This value is typically taken from the noise of the images in the complex domain and is provided in the input data (see [Input data](#)).

Constraints

It is possible to add additional inequality constraints to a composite model, using the `constraints` attribute. These constraints need to be added as the result of the function $g(x)$ where we assume $g(x) \leq 0$.

For example, in the NODDIDA model we implemented the constraint that the intra-cellular diffusivity must be larger than the extra-cellular diffusivity, following Kunz et al., NeuroImage 2018. Mathematically, this constraint can be stated as $d_{ic} \geq d_{ec}$. For implementation in MDT, we will state it as $d_{ec} - d_{ic} \leq 0$ and implement it as:

```
class NODDIDA(CompositeModelTemplate)
    ...
    constraints = '''
        constraints[0] = NODDI_EC.d - NODDI_IC.d;
    '''
```

This `constraints` attribute can hold arbitrary OpenCL C code, as long as it contains the literal `constraints[i]` for each additional constraint `i`.

From this `constraints` string, MDT creates a function with the same dependencies and parameters as the composite model. This function is then provided to the optimization routines, which enforce it using the *penalty* method (https://en.wikipedia.org/wiki/Penalty_method).

5.3 Compartment models

The compartment models are the building blocks of the composite models. They consists in basis of two parts, a list of parameters (see *Parameters*) and the model code in OpenCL C (see *CL code*). At runtime, MDT loads the C/CL code of the compartment model and combines it with the other compartments to form the composite model.

Compartment models can be defined using the templating mechanism by inheriting from `CompartmentTemplate`. For example, the Stick model can be defined as:

```
from mdt.component_templates.compartment_models import CompartmentTemplate

class Stick(CompartmentTemplate):

    parameters = ('g', 'b', 'd', 'theta', 'phi')
    cl_code = '''
        float4 n = (float4)(cos(phi) * sin(theta),
                                sin(phi) * sin(theta),
                                cos(theta),
                                0);

        return exp(-b * d * pown(dot(g, n), 2));
    '''
```

This `Stick` example contains all the basic definitions required for a compartment model: a parameter list and CL code.

Defining parameters

The elements of the parameter list can either be string referencing one of the parameters in the library (like shown in the example above), or it can be a direct instance of a parameter. For example, this is also a valid parameter list:

```
class special_param(FreeParameterTemplate):
    ...

class MyModel(CompartmentTemplate):

    parameters = ('g', 'b', special_param())
    ...
```

where the parameters `g` and `b` are loaded from the dynamically loadable parameters while the `special_param` is given as a parameter instance. It is also possible to provide a nickname for a parameter by stating something like:

```
parameters = ('my_theta(theta)', ...)
```

Here, the parameter `my_theta` is loaded with the nickname `theta`. This allows you to use simpler names for the parameters of a compartment and allows you to swap a parameter for a different type while still using the same (external) name.

Dependency list

Some models may depend on other compartment models or on library functions. These dependencies can be specified using the `dependencies` attribute of the compartment model definition. As an example:

```
dependencies = ('erfi', 'MRIConstants', 'CylinderGPD')
```

This list should contain strings with references to either library functions or other compartment models. In this example the `erfi` library function is loaded from MOT, `MRIConstants` from MDT and `CylinderGPD` is another compartment model which our example depends on.

Adding items to this list means that the corresponding CL functions of these components are included into the optimized OpenCL kernel and allows you to use the corresponding CL code in your compartment model.

For example, in the dependency list above, the `MRIConstants` dependency adds multiple constants to the kernel, like `GAMMA_H`, the gyromagnetic ratio of in the nucleus of H in units of (rad s⁻¹ T⁻¹). By adding `MRIConstants` as a compartment dependency, this constant can now be used in your compartment model function.

Defining extra functions for your code

It is possible that a compartment model needs some auxiliary functions that are too small for an own library function. These can be added to the compartment model using the `cl_extra` attribute. For example:

```
class MyCompartment (CompartmentTemplate):

    parameters = ('g', 'b', 'd')
    cl_code = 'return other_function(g, b, d);'
    cl_extra = '''
        double other_function(
            float4 g,
            mot_float_type b,
            mot_float_type d){

            ...

        }
    '''
```

Extra result maps

It is possible to add additional parameter maps to the fitting and sampling results. These maps are meant to be forthcoming to the end-user by providing additional maps to the output. Extra results maps can be added by both the composite model as well as by the compartment models. By defining them in a compartment model one ensures that all composite models that use that compartment profit from the additional output maps.

Just as with composite models, one can add extra output maps to the optimization results and to the sampling results as:

```
class MyCompartment (CompartmentTemplate):
    ...
    extra_optimization_maps = [
        lambda results: ...
```

(continues on next page)

(continued from previous page)

```

]

extra_sampling_maps = [
    lambda samples: ...
]

```

where each callback function should return a dictionary with extra maps to add to the output.

Constraints

It is possible to add additional inequality constraints to a compartment model, using the `constraints` attribute. These constraints need to be added as the result of the function $g(x)$ where we assume $g(x) \leq 0$.

For example, in the Tensor model we implemented the constraint that the diffusivities must be in a strict order, such that $d_{\parallel} \geq d_{\perp_0} \geq d_{\perp_1}$.

For implementation in MDT, we will state this as the two constraints $d_{\perp_0} \leq d_{\parallel}$ and $d_{\perp_1} \leq d_{\perp_0}$, and implement it as:

```

class Tensor(CompartmentTemplate)
    ...
    constraints = '''
        constraints[0] = dperp0 - d;
        constraints[1] = dperp1 - dperp0;
    '''

```

This `constraints` attribute can hold arbitrary OpenCL C code, as long as it contains the literal `constraints[i]` for each additional constraint `i`.

From this constraints string, MDT creates a function with the same dependencies and parameters as the compartment model. This function is then provided to the optimization routines, which enforce it using the *penalty* method (https://en.wikipedia.org/wiki/Penalty_method).

5.4 Parameters

Parameters form the building blocks of the compartment models. They define how data is provided to the model and form a bridge between the model and the *Input data*.

The type of a parameters determines how the model uses that parameter. For example, compare these two parameters:

```

class theta(FreeParameterTemplate):
    ...

class b(ProtocolParameterTemplate):
    ...

```

In this example the `theta` parameter is defined as a *free parameter* while the `b` parameter is defined as a *protocol parameter*. The type matters and defines how MDT handles the parameter. There are only two types available:

- `FreeParameterTemplate`, for *Free parameters*
- `ProtocolParameterTemplate`, for *Protocol parameters*

See the sections below for more details on each type.

Free parameters

These parameters are normally supposed to be optimized by the optimization routines. They contain some meta-information such as a lower- and upper- bound, sampling prior, parameter transformation function and more. During optimization, parameters of this type can be fixed to a specific value, which means that they are no longer optimized but that their values (per voxel) are provided by a scalar or a map. When fixed, these parameters are still classified as free parameters (you can consider them as fixed free parameters).

To fix these parameters you can either define so in a composite model or using the Python API before model optimization

```
mdt.fit_model('CHARMED_r1',
             ...,
             initialization_data=dict(
                 fixes={},
                 inits={}
             ))
```

A free parameter is identified by having the super class `FreeParameterTemplate`.

Hereunder we list some details that are important when adding a new free parameter to MDT.

Parameter transformations

Panagiotaki (2012) and Harms (2017) describe the use of parameter transformations to limit the range of each parameter to biophysical meaningful values and to scale the parameters to a range better suited for optimization. They work by injecting a parameter transformation before model evaluation that limits the parameters between bounds. See (Harms 2017) for more details on which transformations are used in MDT. You can define the transformation function used by setting the `parameter_transform` attribute. For an overview of the available parameter transformations, see `transformations` in MOT.

Sampling

For sampling one needs to define per parameter a prior and a proposal function. These can easily be added in MDT using the attributes `sampling_proposal` and `sampling_prior`. Additionally, one can define a sampling statistic function `sampling_statistics` which is ran after sampling and returns statistics on the observed samples.

Protocol parameters

These parameters are meant to be fulfilled by the values in the Protocol file (see [Protocol](#)) or in the extra protocol maps. During model optimization, MDT checks the model for protocol parameters and tries to match the names of the protocol parameters with available protocol data. Protocol data can be submitted using either the Protocol file, or using voxel-based protocol maps.

By matching parameter names with input values, the user can add protocol values dynamically by ensuring a common name between the protocol parameter and the provided values.

A protocol parameter is identified by having the super class `ProtocolParameterTemplate`.

5.5 Library functions

Library functions are Python wrappers around reusable CL functions. Having specified them in the compartment model they are included in the CL kernel and hence their CL functions are usable from within the compartment models.

To create one, please look at one of the existing library functions. To use one, add to your compartment model the attribute `dependencies` and add there one or more of the library functions to include in the CL code.

Next to the library functions available in MDT, there are also a few loadable functions in MOT, see `library_functions` for a list.

5.6 Batch profiles

Batch profiles are part of the batch processing engine in MDT. They specify how all the necessary data for model fitting (protocol, nifti volumes, brain mask, noise std, etc.) can be loaded from a directory containing one or more subjects.

The general idea is that the batch profiles indicate how the data is supposed to be loaded per subject. This batch profile can then be used by functions like `batch_apply()` to apply a function to all subjects in the batch profile.

For example, suppose you have a directory containing a lot of subjects on which you want to run the NODDI analysis using MDT. One way to do this would be to write a script that loops over the directories and calls the right fitting commands per subject. Another approach would be to write a batch profile and use the batch processing utilities in MDT to process the datasets automatically. The advantage of the latter is that there are multiple batch processing tools in MDT such as `batch_fit()`, `batch_apply()` and `run_function_on_batch_fit_output()` that allow you to easily manipulate large groups of subjects.

Since the batch profiles form an specification of the directory structure, MDT can guess from a given directory which batch profile to use for that directory. This makes batch processing in some instances as simple as using:

```
$ mdt-batch-fit . 'BallStick_r1'
```

To fit `BallStick_r1` to all subjects found (assuming a suitable batch profile was found).

Chapter 6

Maps visualizer

The MDT maps visualizer is a small convenience utility to visually inspect multiple nifti files simultaneously. In particular, it is useful for quickly visualizing model fitting results.

This viewer is by far not as sophisticated as for example `fslview` and `itksnap`, but that is also not its intention. The primary goal of this visualizer is to quickly display model fitting results to evaluate the quality of fit. A side-goal of the viewer is the ability to create reproducible and paper ready figures showing slices of various volumetric maps.

Features include:

- output figures as images (`.png` and `.jpg`) and as vector files (`.svg`)
- the ability to store plot configuration files that can later be loaded to reproduce figures
- easily display multiple maps simultaneously

Some usage tip and tricks are:

- Click on a point on a map for an annotation box, click outside a map to disable
- Zoom in by scrolling in a plot
- Move the zoom box by clicking and dragging in a plot
- Add new nifti files by dragging them from a folder into the GUI

The following is a screenshot of the GUI displaying NODDI results of the `b1k_b2k` MDT example data slices.

The MDT maps visualizer can be started in three ways, from the MDT analysis GUI, from the command line and using the Python API. With the command line, the visualizer can be started using the command `cli_index_mdt-view-maps`, for example:

```
$ mdt-view-maps .
```

In Windows, one can also type this command in the start menu search bar to load and start the GUI. Using Python, the GUI can be started using the command `mdt.view_maps()`, for example:

```
>>> import mdt
>>> mdt.view_maps('output/NODDI')
```

Finally, using the MDT analysis GUI, the maps visualizer can be started from the menu bar, “File” -> “Maps visualizer”

6.1 GUI Layout

The main body of the GUI is split into two parts, the control panel on the left and the display panel on the right. All plot configuration options can be updated using the control panel, which includes things like colormaps, zooming, font sizes, rotations, clipping and more. Changes in the plot settings are automatically reflected in the display panel. This panel shows the current plot using `matplotlib` as a plotting backend.

On the bottom of the control panel are a few switches to control the rendering of the plots in the display panel. The checkbox “Auto render” disables/enables the automatic rerendering of the plots, allowing you to change multiple

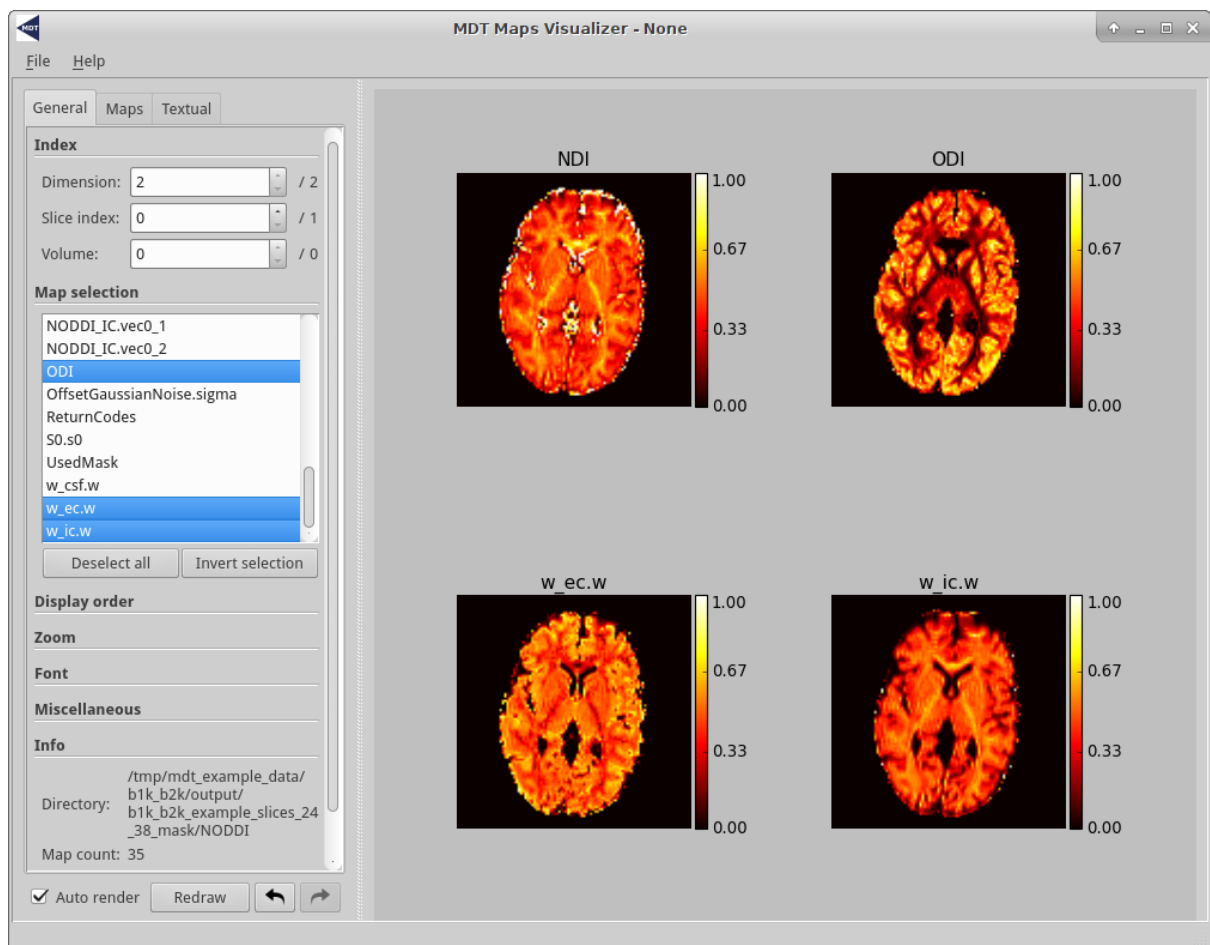


Fig. 1: The MDT map visualizer in Linux

settings without rerendering delays. A manual redraw can be forced using the “Redraw” button. The back and forward arrows allow you to undo or redo updates to the plot settings.

When hovering a map, the bottom right of the window shows some basic voxel statistics for the current mouse position. The first tuple ((63, 50) in the example screenshot below), shows the position of the hovered voxel in the current viewport. The second tuple ((63, 50, 0) in the example) shows the absolute position of the hovered voxel inside the nifti file (the two tuples can be different when the map is zoomed in or rotated). Finally, the last item shows the value/intensity of the hovered voxel.

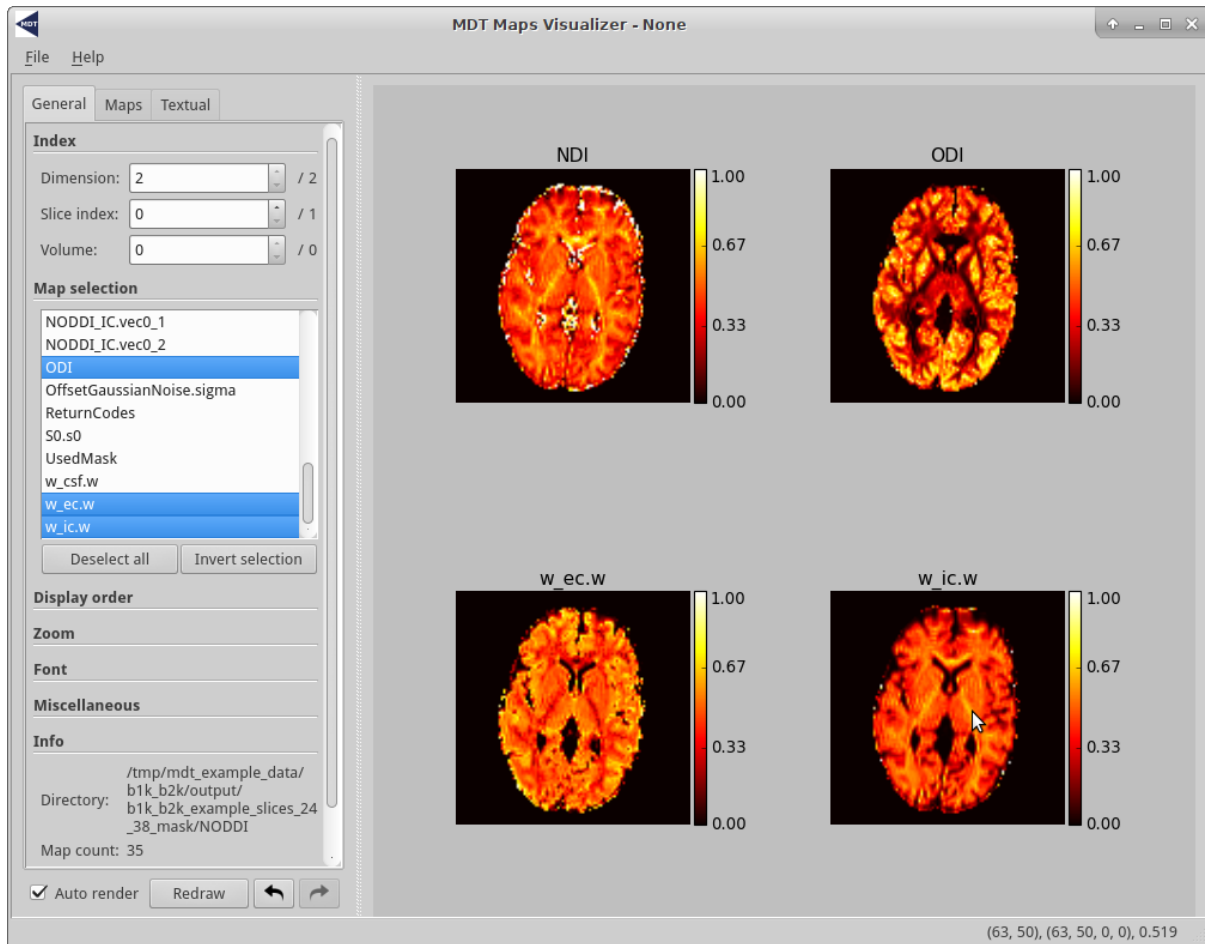


Fig. 2: Screenshot of the GUI running in Linux

Control panel

The control panel consists of three tabs, “General”, “Maps” and “Textual”. The first tab contains general options that all apply to the figure in total. For instance, the zoom settings allows you to zoom in on all maps at the same time and the rotate option under miscellaneous rotates all displayed figures.

The second tab is for map specific options, here one can set plot configuration options that apply only to a single map. After having selected the map you wish to change using the drop down box on the top of the panel you can then update all the values in the tab and the changes will be applied to the chosen map. A common thing to change is the “Scale” of the map, which sets the range of the colormap to the defined scale, left empty this will auto-select a good scaling. Another thing that can be set is the “Clipping” which will actively clip the data to be within the defined range.

The last tab of the control panel contains a live text area that allows you to change all plot settings (the general and the map specific) using a text editor. This text box is automatically updated whenever one of the settings on the other tabs changes and vice versa. This text box can be used to, for example, copy paste a configuration from

one plot into another to let both reflect the exact same settings. For more information on this feature, please see the next section.

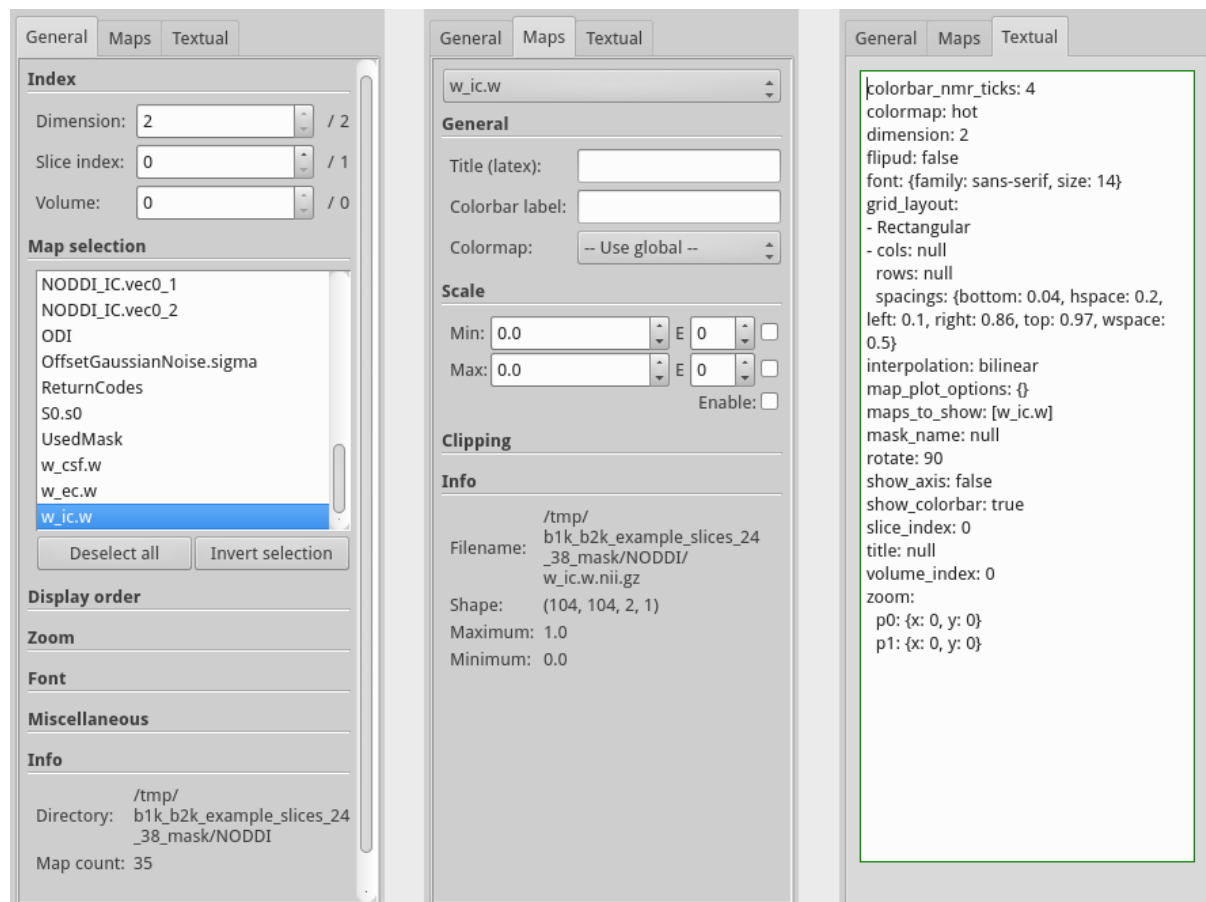


Fig. 3: Figure showing the three tabs of the control panel combined into one figure, with the general tab on the left, the map specific options in the center and the textual input tab on the right.

6.2 Plot configuration

Any instance of the visualization routine consists of two things, data and a plot configuration. The data is commonly loaded by selecting a directory with maps to load (or, using the Python API, a dictionary with maps). Then, the selected maps or a subset of the maps, are visualized according to the plot configuration. This plot configuration can be configured implicitly by using the “General” and “Maps” tag or explicitly using the “Textual” tab.

The plot configuration is commonly stored as a YAML formatted string that lists the various options as dictionary elements. For example, the following configuration is a configuration for BallStick_r1 model fitting results where we set the zoom and the plot titles using the control panels. As an example, after having followed the analysis getting started guide with the BallStick_r1 model, you could try to copy paste this example configuration in the “Textual” tab in the viewer. It should then update the plot to reflect this configuration.

```
maps_to_show: [w_ball.w, w_stick.w]
slice_index: 0
zoom:
  p0: {x: 18, y: 4}
  p1: {x: 85, y: 98}
map_plot_options:
  w_ball.w:
```

(continues on next page)

(continued from previous page)

```
scale: {use_max: true, use_min: true, vmax: 1.0, vmin: 0.0}
title: Isotropic (w_ball.w)
w_stick.w:
scale: {use_max: true, use_min: true, vmax: 1.0, vmin: 0.0}
title: Anisotropic (w_stick.w)
```

An alternative way of saving this configuration file is by using the “Export settings” and “Import settings” in the menu. This will provide easy ways of loading and saving the configuration file as a `.conf` file in YAML format.

Chapter 7

Design concepts

This chapter introduces the reader to a few of the design concepts in MDT.

7.1 Protocol

In MDT, the Protocol contains the MRI measurement settings and is, by convention, stored in a protocol file with suffix `.prtml`. In such a protocol file, every row represents an MRI volume, and every column (tab separated) represents a specific protocol setting. Since every row (within a column) can have a distinct value, this setup automatically enables *multi-shell protocol* files (just change the b-value per volume/row).

The following is an example of a simple MDT protocol file:

```
#gx,gy,gz,b
0.000e+00  0.000e+00  0.000e+00  0.000e+00
5.572e-01  6.731e-01  -4.860e-01  1.000e+09
4.110e-01  -5.254e-01  -7.449e-01  1.000e+09
...
```

And, for a more advanced protocol file:

```
#gx,gy,gz,Delta,delta,TE,b
-0.00e+0  0.00e+0  0.00e+0  2.18e-2  1.29e-2  5.70e-2  0.00e+0
2.92e-1  1.71e-1  -9.41e-1  2.18e-2  1.29e-2  5.70e-2  3.00e+9
-9.87e-1  -8.54e-3  -1.60e-1  2.18e-2  1.29e-2  5.70e-2  5.00e+9
...
```

The header (starting with #) is a single required line with per column the name of that column. The order of the columns does not matter but the order of the header names should match the order of the value columns. MDT automatically links protocol columns to the protocol parameters of a model (see [Protocol parameters](#)), so make sure that the columns names are identical to the protocol parameter names in your model.

The pre-provided list of column names is:

- **b**, the b-values in s/m^2 ($b = \gamma^2 G^2 \delta^2 (\Delta - \delta/3)$ with $\gamma = 2.675987E8 \text{ rads} \cdot s^{-1} \cdot T^{-1}$)
- **gx, gy, gz**, the gradient direction as a unit vector
- **Delta**, the value for Δ in seconds
- **delta**, the value for δ in seconds
- **G**, the gradient amplitude in T/m (Tesla per meter)
- **TE**, the echo time in seconds
- **TR**, the repetition time in seconds

Note that MDT expects the columns to be in **SI units**.

The protocol dependencies change per model and MDT will issue a warning if a required column is missing from the protocol.

A protocol can be created from a `bvec/bval` pair using the command line, python shell and/or GUI. Please see the relevant sections in [Maximum Likelihood Estimation](#) for more details on creating a Protocol file.

7.2 Input data

In MDT, all data that is needed to fit a model is stored in a `SimpleMRIInputData` object. An instance of this object needs to be created before fitting a model. Then, during model fitting, the model loads the relevant data for the computations.

The easiest way to instantiate a input data object is by using the function `load_input_data()`. At a bare minimum, this function requires:

- `volume_info`, a path to the diffusion weighted volume
- `protocol`, an `Protocol` instance containing the protocol information
- `mask`, the mask (3d) specifying which voxels to use for the computations

Additionally you can provide *noise standard deviation*, a *gradient deviations* file and a *dictionary of extra protocol values*. For the *noise standard deviation* you have the choice to either provide a single value, an ndarray with a value per voxel, or the string 'auto'. If 'auto' is given, MDT will try to estimate the noise standard deviation from the unweighted volumes. While this typically works, it is advised to estimate the noise std. manually from air or from noise lines in your k-space. The *gradient deviations* is a map specifying how the g vector differs in different areas in the scan. This can be provided in HCP Wu-Minn format, as a 3x3 matrix per voxel, or as a 3x3 matrix per voxel per volume (i.e. as a $(x, y, z, n, 3, 3)$ matrix for a dataset with n volumes). With the *extra protocol values* the user can provide additional protocol values, these can be scalars, vectors, and/or volumes. If given, these values take precedence over the values in the protocol file.

7.3 Dynamic modules

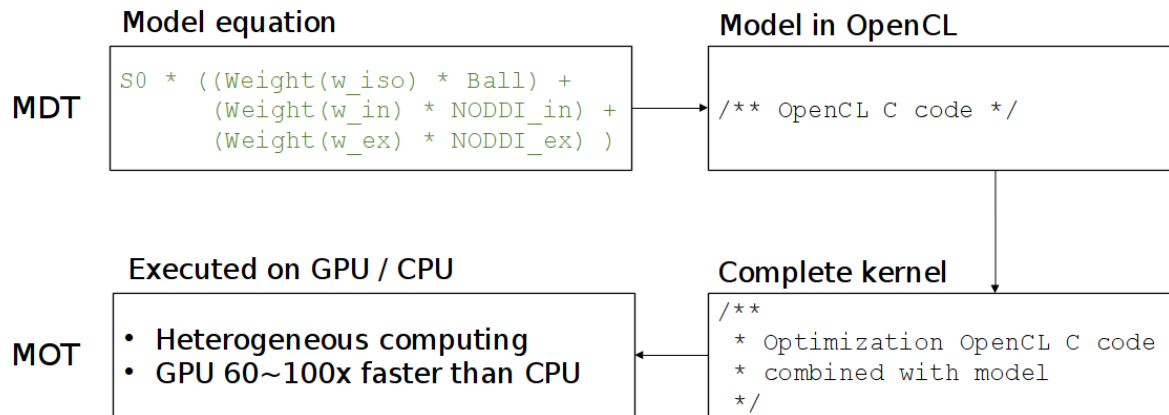
Extending and adapting MDT with new models is made easy using the dynamic library system allowing you to (re)define models anywhere. Users are free to add, remove and modify components and MDT will pickup the changes automatically. See [Adding components](#) for more information.

7.4 CL code

While MDT (and MOT) are programmed in Python, the actual computations are executed using OpenCL. OpenCL is a platform and language specification that allows you to run C-like code on both the processor (CPU) and the graphics cards (GPU). The reason MDT can be fast is since it a) uses a compiled language (OpenCL C) for the computations and b) executes this on the graphics card and/or all CPU cores.

The compartment models in MDT are programmed in the OpenCL C language (CL language from hereon). See (<https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/mathFunctions.html>) for a quick reference on the available math functions in OpenCL.

When optimizing a multi-compartment model, MDT combines the CL code of all your compartments into one large function and uses MOT to optimize this function using the OpenCL framework. See this figure for the general compilation flow in MDT:



To support both single and double floating point precision, MDT uses the `mot_float_type` instead of `float` and `double` for most of the variables and function definitions. During optimization and sampling, `mot_float_type` is type-defined to be either a `float` or a `double`, depending on the desired precision. Of course this does not limit you to use `double` and `float` as well in your code.

Chapter 8

Advanced configuration

The default MDT configuration can be viewed in your home folder in `.mdt/<version>/mdt.default.conf`. This file is merely there for reference and is not read by MDT (rather, those defaults are loaded within MDT). To override the default configuration you can copy this file to `.mdt/<version>/mdt.conf` and set (only) the options you wish to override. These configuration files change the configuration of MDT at application startup, to apply a new configuration file you will need to restart MDT.

8.1 General config

The default configuration file is stored in `.mdt/<version>/mdt.default.conf` and is only there for reference. The configuration is in **YAML** format. To change part of the configuration you can make a copy of this file to `.mdt/<version>/mdt.conf` and set the specific options you want to change.

For example, suppose you want to disable the automatic zipping of the nifti files after optimization. You can create an *empty* `mdt.conf` file and add to this the lines:

```
output_format:
  optimization:
    gzip: False
  sampling:
    gzip: False
```

This leaves the rest of the configuration file at the defaults. If you also wish to change the default value of `tmp_results_dir` you can add it (in any order) to your `mdt.conf` file:

```
output_format:
  ...
tmp_results_dir: /tmp
```

The `mdt.conf` file is automatically copied to the directory of a new version. The configuration options are stable in general and may only break over major versions.

8.2 GUI specific config

Next to the more general `.mdt/<version>/mdt.conf` which is applied in general, you can also have a GUI specific configuration file, `.mdt/<version>/mdt.gui.conf`. When you start any of the MDT graphical interfaces, this file is loaded after the regular `mdt.conf` configuration and hence takes priority over the other settings. Its purpose is to contain a few GUI specific configuration values as well as to be able to set a specific configuration for the GUI only. The structure of the file is the same as that of the other configuration files.

8.3 Runtime configuration

It is also possible to change the MDT configuration during code execution using configuration contexts. Contexts are a Python programming pattern that allows you to run code before and after another piece of code. MDT uses

this pattern to allow for temporarily changing the MDT configuration for the duration of the context. An advantage of this pattern is that it makes sure that the configuration is reset back to its previous state after the code has been executed. An example of using the configuration contexts:

```
with mdt.config_context(ConfigAction()):  
    mdt.fit_model(...)
```

See the `mdt.configuration` module for more details on how to use this functionality.

Chapter 9

Frequently Asked Questions

Table of Contents

- *Installation problems*
 - *No OpenCL device found*
 - * *Check devices*
 - * *Check drivers*
- *Analysis*
 - *Why does the noise standard deviation differ when using another mask?*

9.1 Installation problems

No OpenCL device found

If there are no CL devices visible in the GUI and the shell command `mdt-list-devices` returns nothing, no computations can be done using MDT. To continue using MDT, OpenCL enabled hardware and corresponding drivers must be installed in your system.

Check devices

First, make sure that the graphics card and/or CPU in your system is capable of OpenCL acceleration. To do so, look up the device name in your computer and find its specifications on the internet. The device must support OpenCL and at least OpenCL version 1.2.

Check drivers

If your preferred device supports OpenCL and it does not show in MDT, you may be missing the device drivers.

If you would like to run the computations on a GPU (graphics card), please install the correct drivers for that card. If you would like to run the computations on the CPU, you have two possibilities. The first is to install an AMD graphics card, their drivers come pre-supplied with OpenCL drivers for CPU's (for both Intel and AMD). If you do not have a graphics card, or you have an NVidia card, you will have to install the [Intel OpenCL Drivers](#) to your system.

9.2 Analysis

Why does the noise standard deviation differ when using another mask?

By default MDT tries to estimate the noise standard deviation of the images in the complex domain. This standard deviation is used in the analysis as the standard deviation in the likelihood function (commonly Offset-Gaussian).

This standard deviation is commonly estimated using an average of per-voxel estimations. When a different mask is used there are different voxels used for the standard deviation estimation and hence the resulting value differs.

To prevent this from happening it is suggested that researchers estimate the noise std. beforehand with a whole brain mask and use the obtained std. in all other analysis.

Chapter 10

Scientific articles

Please consider citing the following article when using MDT in your research:

- R.L. Harms, F.J. Fritz, A. Tobisch, R. Goebel, and A. Roebroek. Robust and fast nonlinear optimization of diffusion MRI microstructure models. *NeuroImage*, 155(October 2016):82–96, jul 2017. URL: <http://dx.doi.org/10.1016/j.neuroimage.2017.04.064>, doi:10.1016/j.neuroimage.2017.04.064.
- R.L. Harms and A. Roebroek. Robust and Fast Markov Chain Monte Carlo Sampling of Diffusion MRI Microstructure Models. *Frontiers in Neuroinformatics*, 12(December):1–18, dec 2018. URL: <https://www.frontiersin.org/article/10.3389/fninf.2018.00097/full>, doi:10.3389/fninf.2018.00097.

Chapter 11

Credits

The Microstructure Diffusion Toolbox is a model recovery toolbox primarily meant for diffusion MRI analysis.

List of contributors:

- **Robbert Harms**
 - Primary developer
- **Alard Roebroek**
 - Advisor and Phd. supervisor
- **Francisco Fritz**
 - Added Relaxometry, MPM and SSFP models
 - Quality Control on first public version
- **Mark Drakesmith**
 - Contributed the Poisson distributed ActiveAx model

This work was (partially) sponsored by:

- ERC Starting Grant (MULTICONNECT, #639938)
- Dutch science foundation (NWO) VIDI Grant (#14637)
- Maastricht University, the Netherlands
- Brain Innovation, Maastricht, the Netherlands