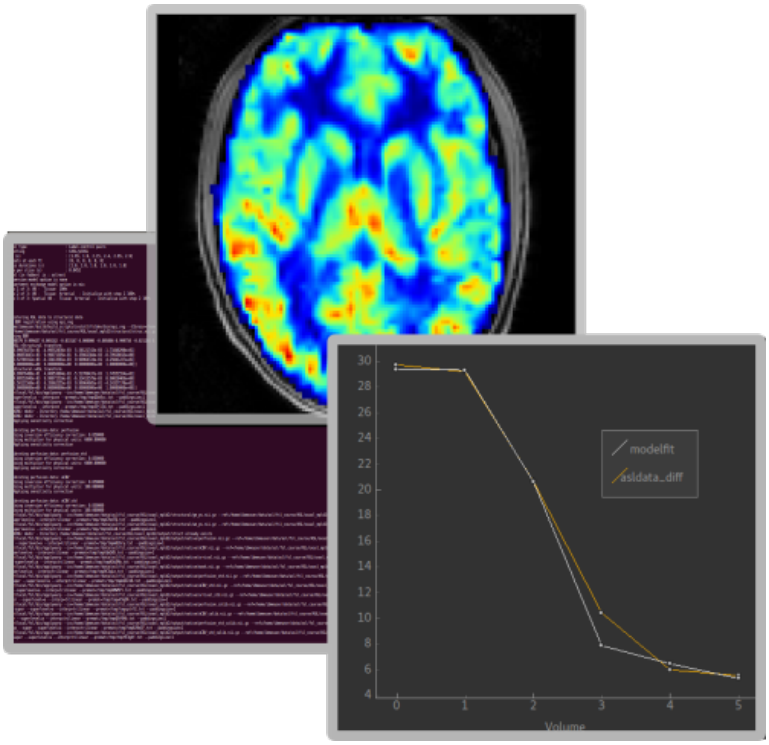
Fabber documentation

Release 0.0.1

Martin Craig

May 11, 2022

Contents:



Fabber is a tool for fitting timeseries data to a forward model using a Bayesian approach. It has been designed for use with fMRI data such as ASL and CEST, however the method is quite general and could be applied to most problems involving fitting discrete data points to a nonlinear model.

Bayesian nonlinear model fitting provides a more flexible alternative to traditional linear analysis of perfusion data.

In multi-echo or multi-inversion-time data, this approach uses all time points to estimate all the parameters of interest simultaneously. This can be more accurate because it uses a detailed nonlinear model of the pulse sequence, rather than assuming that different parts of the signal are linearly related to each quantity of interest.

Note: If you want to process ASL data you should look at the OXASL pipeline which uses Fabber as it's model fitting tool. Similarly if you have CEST data, the FSL Baycest tool uses Fabber. The FSL Verbena tool uses Fabber to process DSC data.

To make this analysis technique fast enough for routine use, we have adapted an approximate method known as Variational Bayes (VB) to work with non-linear forward models. Experimentally we have found that these calculations provide nearly identical results to sampling methods such as MCMC and require only a fraction of the computation time (around a minute per slice).

Fabber has a modular design and new nonlinear forward models can be incorporated into the source code. Models have been developed for ASL, CEST, DSC, DCE and dual echo fMRI data.

Fabber is distributed as part of FSL, however you should ensure that you are using **FSL v6.0.1 as a minimum version**.

The Quantiphyse visual analysis tool contains plugins to analyse ASL, CEST, DSC and DCE fMRI data using Fabber.

CHAPTER 1

Getting Fabber

1.1 From FSL

Fabber is distributed as part of FSL, and this is the easiest way to get Fabber if you want to use existing models for fMRI data . This documentation describes the version of Fabber included with *FSL v6.0.1 and above*.

Addition tools that can use Fabber will work with a correctly installed FSL distribution although currently not all models we have developed are available in FSL.

1.2 Standalone Fabber distribution

Standalone versions of Fabber including a selection of model libraries are available for a number of platforms. These may be useful if you don't want the rest of FSL or if you need a more up to date version of Fabber than the one included with FSL.

The current standalone release can be found at https://github.com/physimals/fabber_core/releases.

The standalone release can be used with tools requiring a Fabber installation such as the Python API, or Fabber-based plugins for Quantiphyse. You should set the environment variable `FABBERDIR` to the unpacked distribution directory to ensure these tools can find Fabber. Note that some Quantiphyse plugins require a full FSL installation, notably the ASL plugin.

1.3 Building from source code

You can build Fabber from the source code available in the Github repository. You will need an FSL installation for this. For instructions see Building Fabber.

CHAPTER 2

Building Fabber

In most cases **you don't need to build Fabber** - executables covering a variety of models are available in FSL (v6.0.1 or later recommended). You might need to use the following instructions, however if you:

- Need to install updated code which has not yet been released in FSL
- Want to write your own model or otherwise modify the code

2.1 Building Fabber using an installed FSL distribution

You will need FSL to build Fabber - it requires a number of libraries distributed as part of FSL. In addition the Fabber `Makefile` is based around the FSL build system.

Note: An additional `cmake` based build system also exists for use particularly on Windows. We will not describe this here.

2.1.1 Setting up an FSL development environment

First you need to have your system set up to compile FSL code. If you're already building other FSL tools from source you've probably already done this, and can skip this section. Otherwise, run the following commands:

```
source $FSLDIR/etc/fslconf/fsl-devel.sh
export FSLDEVDIR=<prefix to install into>
export PATH=$FSLDEVDIR/bin:$PATH
```

Note: you may want to put this in your `.profile` or `.bash_profile` script if you are going to be doing a lot of recompiling

---

FSLDEVDIR is an alternate prefix to FSLDIR which is used to store updated code separately from the official FSL release. You might want to set it to something in your home directory, e.g. $HOME/fsldev. Most FSL-based scripts should use code installed in FSLDEVDIR in preference to the main FSL release code.

2.1.2 Setting up compiler/linker flags

Note: This is not always necessary - it depends on the system you're building on. So try skipping to the 'Building...' sections below and come back if you get compiler/linker errors.

Firstly, if you're getting errors trying to build you need to do `make clean` and (to be safe!) `rm depend.mk` before building again with new settings. Also note that the FSL build system changed in v6.0.3 - here we provide only information for the new build system.

The relevant settings are in $FSLDIR/config/buildSettings.mk. In particular you may need to modify ARCHFLAGS for your system - be careful as there are separate definitions for Linux and Mac ('Darwin') systems, so make sure you change the right one!

For recent versions of Ubuntu, you need to turn off use of the C++11 ABI as FSL libraries are not compiled using this. To do this add the following to ARCHFLAGS:

```
-D_GLIBCXX_USE_CXX11_ABI=0 -no-pie
```

If you are having difficulty with other systems, please raise an issue and we will investigate.

2.1.3 Building `fabber_core`

You can probably skip this if you are just building an updated model library. If you need to recompile the core, however, it should be a case of:

```
cd fabber_core
make install
```

This approach uses the same build tools as the rest of FSL which is important on some platforms, notably OSX. It will install the updated code into whatever prefix you selected as FSLDEVDIR.

2.1.4 Building new or updated model libraries

Model libraries are distributed separately from the Fabber core. If you need an updated version of a model library, for example the ASL model library, you first need to get the source code for the models library. A number of model libraries are available in our Github repositories all named fabber_models_<name>.

Then to build and install the updated model libraries you would then run, for example:

```
cd fabber_models_asl
make install
```

2.2 Adding your own models

If you want to create your own model to use with the Fabber core model fitting engine, see Building a new model library. Once you've designed and coded your model there are two ways to incorporate it into the Fabber system:

2.2.1 Adding models directly to the core

If you wish, you can add your own models directly into the Fabber source tree and build the executable as above. This is not generally recommended because your model will be built into the core executable, however it can be the quickest way to get an existing model built in. You will need to follow these steps:

1. Add your model source code into the fabber_core directory, e.g.

```
fabber_core/fwdmodel_mine.cc
fabber_core/fwdmodel_mine.h
```

2. Edit `Makefile` to add your model to the list of core objects, e.g.

```
COREOBJS =  fwdmodel_mine.o noisemodel.o fwdmodel.o inference.o fwdmodel_linear.o␣
↪fwdmodel_poly.o convergence.o motioncorr.o priors.o transforms.o
```

3. Run `make install` again to build and install a new executable

2.2.2 Creating a new models library

This is the preferred approach if you want to distribute your new models. A template for a new model library including a simple sine-function implementation is included with the Fabber source code in `fabber_core/examples`. See Building a new model library for a full tutorial on this example which includes how to set up the build scripts.

CHAPTER 3

Running Fabber

3.1 Specifying options

The simplest way to run Fabber is as a command line program. It uses the following syntax for options:

```
--option
--option=value
```

A simple example command line would be:

```
fabber --data=fmri_volume.nii.gz --mask=roi.nii.gz \
       --model=poly --degree=2 \
       --method=vb --noise=white \
       --output=out --save-model-fit \
```

3.2 Common options

--output=OUTPUTDIR Directory for output files (including logfile)

--method=METHOD Use this inference method

--model=MODEL Use this forward model

--data=DATAFILE Specify a single input data file

--mask=MASKFILE Mask file. Inference will only be performed where mask value > 0

--optfile File containing additional options, one per line, in the same form as specified on the command line

--overwrite If set will overwrite existing output. If not set, new output directories will be created by appending '+' to the directory name

--suppdata 'Supplemental' timeseries data, required for some models

3.3 Selecting what data to output

--save-model-fit Output the model prediction as a 4d volume

--save-residuals Output the residuals (difference between the data and the model prediction)

--save-model-extras Output any additional model-specific timeseries data

--save-mvn Output the final MVN distributions.

--save-mean Output the parameter means.

--save-std Output the parameter standard deviations.

--save-zstat Output the parameter Zstats.

--save-noise-mean Output the noise means. The noise distribution inferred is the precision of a Gaussian noise source

--save-noise-std Output the noise standard deviations.

--save-free-energy Output the free energy, if calculated.

3.4 Help and usage information

--help Print this usage method. If given with –method or –model, display relevant method/model usage information

--version Print version identifier. If given with –model will print the model's version identifier

--listmethods List all known inference methods

--listmodels List all known forward models

--listparams List model parameters (requires model configuration options to be given)

--listoutputs List additional model outputs (requires model configuration options to be given)

3.5 Advanced options

--simple-output Instead of usual standard output, simply output series of lines each giving progress as percentage

--data1=DATAFILE, --data2=DATAFILE Specify multiple data files for n=1, 2, 3...

--data-order If multiple data files are specified, how they will be handled: concatenate = one after the other, interleave = first record from each file, then second, etc.

--mt1=INDEX, --mt2=INDEX List of masked time points, indexed from 1. These will be ignored in the parameter updates

--debug Output large amounts of debug information. ONLY USE WITH VERY SMALL NUMBERS OF VOXELS

--link-to-latest Try to create a link to the most recent output directory with the prefix _latest

--loadmodels Load models dynamically from the specified filename, which should be a DLL/shared library

3.6 Variational Bayes options (used when method=vb)

--noise=NOISE Noise model to use (white or ar1)

--convergence=CONVERGENCE Name of method for detecting convergence - default maxits, other values are fchange, trialmode

--max-iterations=NITS number of iterations of VB to use with the maxits convergence detector

--min-fchange=FCHANGE When using the fchange convergence detector, the change in F to stop at

--max-trials=NTRIALS When using the trial mode convergence detector, the maximum number of trials after an initial reduction in F

--print-free-energy Output the free energy in the log file

--continue-from-mvn=MVNFILE Continue previous run from output MVN files

--output-only Skip model fitting, just output requested data based on supplied MVN. Can only be used with continue-from-mvn

--noise-initial-prior=MVNFILE MVN of initial noise prior

--noise-initial-posterior=MVNFILE MVN of initial noise posterior

--noise-pattern=PATTERN repeating pattern of noise variances for each point (e.g. 12 gives odd and even data points different variances)

--PSP_byname1=PARAMNAME, --PSP_byname2=PARAMNAME Name of model parameter to use for prior specification 1, 2, 3...

--PSP_byname1_type=PRIORTYPE Type of prior to use for parameter 1 - I=image prior

--PSP_byname1_image=FILENAME File containing image for image prior for parameter 1

--PSP_byname1_prec Precision to apply to image prior for parameter 1

--PSP_byname1_transform Transform to apply to parameter 1

--allow-bad-voxels Continue if numerical error found in a voxel, rather than stopping

--ar1-cross-terms=TERMS For AR1 noise, type of cross-linking (dual, same or none)

--spatial-dims=NDIMS Number of spatial dimensions (1, 2 or 3). Default is 3.

--spatial-speed=SPEED Restrict speed of spatial smoothing

--param-spatial-priors=PRIORSTR Type of spatial priors for each parameter, as a sequence of characters. N=nonspatial, M=Markov random field, P=Penny, A=ARD

--locked-linear-from-mvn=MVNFILE MVN file containing fixed centres for linearization

3.7 Model-specific options

These are usually quite extensive and control the fine details of the model that is being implemented. For example the generic ASL model will need to be told the TIs/PLDs of the sequence, the number of repeats, the structure of the data, bolus duration and what components to include in the model (arterial as well as tissue, dispersion and exchange options, ...).

The best way to look at model options is to use `--help`, e.g.:

```
fabber_asl --help --model=aslrest
```

CHAPTER 4

Priors in Fabber

Each parameter in a Fabber model has a *prior* which describes our existing knowledge of the parameter's value before we see any data.

A model must provide a set of priors for all of it's parameters and in general it is not good practice to modify them - especially in light of knowledge derived from the data as this undermines the Bayesian principles.

Nevertheless there are a number of options that can be set for priors which can be used reasonably.

4.1 Spatial priors

A spatial prior applies spatial regularization to the parameter so that the spatial variation in it's value is limited by the information present in the data. This has the effect of smoothing the parameter map in areas where there is not enough information in the data to justify more detail.

This can be beneficial since it produces smoother parameter maps with clearer structure but done in a principled way which treats each parameter independently and applies a degree of smoothing related to the information in the data.

A spatial prior would be defined as follows:

```
--PSP_byname1=myparam
--PSP_byname1_type=M
```

The first options specifies which named parameter any additional `--PSP_byname1_*` options refer to. The second option sets the prior type as `M` which is the most common type of spatial prior. Other supported types are `m`, `P` and *p*.

Spatial priors are normally only applied to a single parameter which is representative of the overall scale of the data. Since all the parameters are linked in the model, the result will generally be that all parameters are smoothed appropriately.

The following descriptions of the spatial prior types are based on Penny et al 2004.

4.1.1 Markov random field spatial prior (type M)

In this case the spatial matrix $S^T S$ is defined as 1 for nearest neighbour voxels and 0 otherwise. The actual number of nearest neighbours is used so there is no bias at boundaries (e.g. at the surface of the volume)

4.1.2 Markov random field spatial prior without boundary correction (type m)

In this case the spatial matrix $S^T S$ is defined as 1 for nearest neighbour voxels and 0 otherwise. The number of nearest neighbours is defined by the number of spatial dimensions (i.e. 8 for 3D spatial inference). This can cause bias at the image/mask boundaries hence spatial prior type `M` is generally used instead.

4.2 ARD priors

Automatic Relevance Detection (ARD) is a type of prior in which a parameter's value can 'collapse' to zero if there is not sufficient information in the data to justify it having a nonzero value. This is useful for parameters which may be relevant only in certain parts of the data, for example an arterial signal component which only exists in large arteries.

An ARD prior would be defined as follows:

```
--PSP_byname1=myparam
--PSP_byname1_type=A
```

4.3 Image priors

An image prior is a prior whose mean value may be different at each voxel. For example if the tissue's local T1 value is a model parameter, it may be useful to use a T1 map calculated by some independent means (e.g. VFA or MOLLI sequences) to provide the prior value at each voxel, while still allowing for the possibility of variation.

Image priors can be specified as follows:

```
--PSP_byname1=myparam
--PSP_byname1_type=I
--PSP_byname1_prec=100
```

Note that the precision can be specified, this controls how free the model is to vary the parameter. Choosing a high precision (e.g. 1e6) effectively makes the image 'ground truth'. In this case we have given a precision of 100 which translates into a standard deviation of 0.1, allowing *some* variation in the inferred value but ensuring it will remain close to the image value.

4.4 Customizing priors

Warning: Customizing priors, especially in response to information from the data is opposed to the Bayesian methodology and should not be done unless you have good reason!

It is possible to override the model's built-in priors and specify their mean and precision directly. This is done as follows:

```
--PSP_byname1=myparam
--PSP_byname1_mean=1.5
--PSP_byname1_prec=0.1
```

This would set the prior for parameter 'myparam' to have a mean of 1.5 and a precision of 0.1 (variance=10).

While this is normally discouraged, there are cases where it may be appropriate, for example when studying a population whose physiological parameters are known to differ systematically from the average, or for similar reasons to allow a parameter to vary more from the 'standard' prior value than the model normally allows.

4.5 Parameter transformations

Parameter transformations can be used when the default Gaussian distribution does not seem appropriate for a parameter. An example would be a parameter which for physical reasons cannot be negative. In this case we might guess that a log-normal distribution would be more appropriate. This can be handled in Fabber by telling the core inference engine to work with the log of the parameter value (which is distributed as a Gaussian) and transform it to the actual value when evaluating the model.

Warning: Transformations are normally built into the model where they are appropriate. Inappropriate transformations can lead to numerical instability and poor fitting.

Since transformations are transparent to the model they can be modified as follows:

```
--PSP_byname1=myparam
--PSP_byname1_trans=L
```

This sets the parameter named `myparam` to have a log-transform.

4.5.1 Prior mean/precision and transformations

A natural question is how should the prior mean and variance be modified when using a transformation. For example suppose we have a parameter representing a transit time and it's normal prior has a mean of 1.3s and a precision of 5. Unfortunately this defines a Gaussian which has a significant probability of being negative, which is probably not physically reasonable.

We might choose to apply a log-transform to this parameter to avoid this problem. But what should the mean and variance of the underlying Gaussian distribution (i.e. the distribution of the log of the value) be.

We might naively assume that the same transform applies fir the mean, however this is not the case. If we choose $log(1.3)$ as our mean we are modelling the prior as a log-normal distribution with a *geometric* mean of 1.3, which is subtly different.

CHAPTER 5

Building a new model library

For most new applications, a model will need to be constructed. This will include adjustable parameters which Fabber will then fit.

The example shown below is included in the `examples` subdirectory of the Fabber source code. This provides an easy template to implement a new model.

We will assume only some basic knowledge of `C++` for this example.

5.1 A simple example

To create a new Fabber model it is necessary to create an instance of the class `FwdModel`. As an example, we will create a model which fits the data to sum of exponential functions, each with an amplitude and decay rate.

$$\sum_n A_n \exp(-R_n t)$$

Note: The source code and `Makefile` file for this example are in the Fabber source code, in the `examples` subdirectory. We will assume you have this to hand as we go through the process!

First we will create the interface `fwdmodel_exp.h` file which shows the methods we will need to implement:

```
// fwdmodel_exp.h - A simple exponential sum model
#pragma once

#include "fabber_core/fwdmodel.h"

#include "newmat.h"

#include <string>
#include <vector>

```

(continues on next page)

(continued from previous page)

```
class ExpFwdModel : public FwdModel {
public:
    static FwdModel* NewInstance();

    ExpFwdModel()
        : m_num(1), m_dt(1.0)
    {
    }

    std::string ModelVersion() const;
    std::string GetDescription() const;
    void GetOptions(std::vector<OptionSpec> &opts) const;

    void Initialize(FabberRunData &args);
    void EvaluateModel(const NEWMAT::ColumnVector &params,
                       NEWMAT::ColumnVector &result,
                       const std::string &key="") const;

protected:
    void GetParameterDefaults(std::vector<Parameter> &params) const;

private:
    int m_num;
    double m_dt;
    static FactoryRegistration<FwdModelFactory, ExpFwdModel> registration;
};
```

We have not made our methods virtual, so nobody will be able to create a subclass of our model. If we wanted this to be the case all the non-static methods would need to be virtual, and we would need to add a virtual destructor. This is sometimes useful when you want to create variations on a basic model (for example we have a variety of DCE models all inheriting from a common base class).

Most of the code above is completely generic to any model. The only parts which are specific to our exp-function model are:

- The name `ExpFwdModel`
- The private variables `m_num` (the number of exponentials in our sum) and `m_dt` (the time between data points).

We will now implement these methods one by one. Many of them are straightforward. We start our implementation file `fwdmodel_exp.cc` as follows:

```
//  fwdmodel_exp.cc - Implements a simple exp curve fitting model
#include "fwdmodel_exp.h"

#include <fabber_core/fwdmodel.h>

#include <math.h>

using namespace std;
using namespace NEWMAT;
```

This just declares some standard headers we will use. If you prefer to fully qualify your namespaces you can leave out the `using namespace` lines.

We need to implement a couple of methods to ensure that our model is visible to the Fabber system:

```
FactoryRegistration<FwdModelFactory, ExpFwdModel> ExpFwdModel::registration("exp");

FwdModel* ExpFwdModel::NewInstance()
{
    return new ExpFwdModel();
}
```

The first line here registers our model so that it is known to Fabber by the name `exp` The second line is a *Factory method* used so that Fabber can create a new instance of our model when its name appears on the command line:

```
string ExpFwdModel::ModelVersion() const
{
    return "1.0";
}

string ExpFwdModel::GetDescription() const
{
    return "Example model of a sum of exponentials";
}
```

We've given our model a version number, if we update it at some later stage we should change the number returned so anybody using the model will know it has changed and what version they have. There's also a brief description which fabber will return when the user requests help on the model:

```
static OptionSpec OPTIONS[] = {
    { "dt", OPT_FLOAT, "Time separation between samples", OPT_REQ, "" },
    { "num-exps", OPT_INT, "Number of independent exponentials in sum", OPT_NONREQ, "1
↪" },
    { "" }
};

void ExpFwdModel::GetOptions(vector<OptionSpec> &opts) const
{
    for (int i = 0; OPTIONS[i].name != ""; i++)
    {
        opts.push_back(OPTIONS[i]);
    }
}
```

This is the suggested way to declare the options that your model can take - in this case the user can choose how many exponentials to include in the sum and what the time resolution in the data is. Each option is listed in the `OPTIONS` array which **ends with an empty option** (important!).

An option is described by:

- It's name which generally should *not* include underscores (hyphen is preferred as in this case). The name translates into a command line option e.g. `--num-exps`.
- **An option type. Possibilities are:**
  - `OPT_BOOL` for a Yes/No option which is considered 'off' unless it is specified
  - `OPT_FLOAT` for a decimal number
  - `OPT_INT` for a whole number (integer)
  - `OPT_STR` for text
  - `OPT_MATRIX` for a small matrix (specified by giving the filename of a text file which contains the matrix data in tab-separated form)

  - OPT_IMAGE for a 3D image specified as a Nifti file
  - OPT_TIMESERIES for a 4D image specified as a Nifti file
  - OPT_FILE for a generic filename

- A brief description of the option. This will be displayed when --help is requested for the model
- OPT_NONREQ if the option is not mandatory (does not need to be specified) or OPT_REQ if the option must be provided by the user.
- An indication of the default value. This value is *not* used to initialize anything but is shown in --help to explain to the user what the default is if the option is not given. So it can contain any text (e.g. "0.7 for PASL, 1.3 for pCASL". You should not specify a default for a mandatory option (OPT_REQ)

In this case we have made the time resolution option mandatory because we have no reasonable way to guess this, but the number of exponentials defaults to 1.

This option system is a little cumbersome when there is only a couple of options, but if you have many it will make it clear to see what they are. Most real models will have many configuration options, for example an ASL model will need to know details of the sequence such as the TIs/PLDs, the bolus duration, the labelling method, number of repeats, etc...

Options specified by the user are captured in the FabberRunData object which we use to set the variables in our model class in the Initialize method. Initialize is called before the model will be used. Its purpose is to allow the model to set up any internal variables based on the user-supplied options. Here we capture the time resolution option and the number of exponentials - note that the latter has a default value:

```
void ExpFwdModel::Initialize(FabberRunData& rundata)
{
    m_dt = rundata.GetDouble("dt");
    m_num = rundata.GetIntDefault("num-exps", 1);
}
```

The lack of a default value for dt means that an exception will be thrown if this option is not specified.

We use the term *Options* to distinguish user-specified or default model configuration from *Parameters* which are the variables of the model inferred by the Fabber process. Next we need to specify what parameters our model includes:

```
void ExpFwdModel::GetParameterDefaults(std::vector<Parameter> &params) const
{
    params.clear();

    int p=0;
    for (int i=0; i<m_num; i++) {
        params.push_back(Parameter(p++, "amp" + stringify(i+1), DistParams(1, 100),␣
↪DistParams(1, 100), PRIOR_NORMAL, TRANSFORM_LOG()));
        params.push_back(Parameter(p++, "r" + stringify(i+1), DistParams(1, 100),␣
↪DistParams(1, 100), PRIOR_NORMAL, TRANSFORM_LOG()));
    }
}
```

GetParameterDefaults is quite important. It declares the parameters our model takes, and their prior and initial posterior distributions. It is always called *after* Initialize so you can use whatever options you have set up to decide what parameters to include.

The code above declares two parameters named amp<n> and r<n> for each exponential in the sum, where <n> is 1, 2, ... As well as a name, each parameter has two DistParams instances defining the *prior* and *initial posterior* distribution for the parameter. DistParams take two parameters - a mean and a variance. At this point we will diverge slightly to explain what these mean.

5.1.1 Priors and Posteriors

Priors are central to Bayesian inference, and describe the extent of our belief about a parameter's value *before we have seen any data*.

For example if a parameter represents the T_1 value of grey matter in the brain there is a well known range of plausible values. By declaring a suitable prior we ensure that probabilities are calculated correctly and unlikely values of the parameter are avoided unless the data very strongly supports this.

In our case we have no real prior information, so we are using an *uninformative* prior. This has a large variance so the model has a lot of freedom in fitting the parameters and will try to get as close to matching the data as it can. This is reflected in the high variance we are using (`1e6`). For the mean values, `a` and `b` are multiplicative so it makes sense to give them defaults of `1` wherease `c` and `d` are additive so prior means of `0` seems more appropriate.

The second `DistParams` instance represents the initial *posterior*. This is the starting point for the optimisation as it tries to find the best values for each parameter. Since the optimization process should iterate to the correct posterior, this may not matter too much and can often be set to be identical to the prior.

When using a non-informative prior, however, it may be better to give the initial posterior a more restrictive (lower) variance to avoid numerical instability. We have done that here, using 100 for the initial posterior variance.

There is rarely a good reason to set the initial posterior to have a different mean to the prior globally. However it is possible to adjust the initial posterior on a per-voxel basis using the actual voxel data. We will not do that here, but it can be useful when fitting, for example, a constant offset, where we can tell the optimisation to start with a value that is the mean of the data. This may help avoid instability and local minima.

In general it is against the spirit of the Bayesian approach to modify the priors on the basis of the data, and we don't provide a method for doing thsi. It is possible for the user to modify the priors on a global basis but this is not encouraged and in general a model should try to provide good priors that will not need modification.

We now go back to our code where we finally reach the point where we calculate the output of our model:

```
void ExpFwdModel::EvaluateModel(const NEWMAT::ColumnVector &params,
                                NEWMAT::ColumnVector &result,
                                const std::string &key) const
{
    result.ReSize(data.Nrows());
    result = 0;

    for (int i=0; i<m_num; i++) {
        double amp = params(2*i+1);
        double r = params(2*i+2);
        for (int i=0; i < data.Nrows(); i++)
        {
            double t = double(i) * m_dt;
            double val = amp * exp(-r * t);
            result(i+1) += val;
        }
    }
}
```

We are given a list of parameter values (`params`) and need to produce a time series of predicted data values (`result`). We do this by looping over the parameters and adding the result of each exponential to the output result.

The additional argument `key` is not required in this case. It is used to allow a model to evaluate 'alternative' outputs such as an interim residual or AIF curve. These are not used in the fitting process but can be written out using the `--save-model-extras` option.

Note that the variable `data` is available at this point and contains the current voxel's time series. We are using it here to determine how many time points to generate.

5.2 Making the example into an executable

We need one more file to build our new model library into it's own Fabber executable. This is called `fabber_main.cc` and it is very simple:

```
#include "fabber_core/fabber_core.h"

int main(int argc, char **argv)
{
    return execute(argc, argv);
}
```

Any number of models can be included in a library. The resulting executable will contain all the new models we define alongside the default generic models `linear` and `poly`.

Note: It is also possible to build Fabber models into a shared library which can be loaded dynamically by any Fabber executable. We will not do that in this example but if you're interested look at the additional source files `exp_models.cc` and `exp_models.h` for details.

5.3 Building an executable with our new model

The example template comes with a `Makefile` which can be used to build the model library using the FSL build system. First you need to set up an FSL build environment as described in Building Fabber. Then to build and install our new model library we can just do:

```
make install
```

This creates an executable `fabber_exp` which installs into `$FSLDEVDIR/bin`. This executable contains the built-in generic models and also our new model - you can see this by running:

```
fabber_exp --listmodels
fabber_exp --help --model=exp
```

5.4 Testing the model - single exponential

A Python interface to Fabber is available which includes a simple self-test framework for models. To use this you will need to get the `pyfab` package - see pyfab.readthedocs.io for more information on installing this package.

Once installed a simple test script for this model might look like this (this script is included in the example with the name `test_single.py`:

```
#!/bin/env python
import sys
import traceback

from fabber import self_test, FabberException

save = "--save" in sys.argv
try:
    rundata= {
```

(continues on next page)

(continued from previous page)

```
        "model" : "exp",       # Exponential model
        "num-exps" : 1,             # Single exponential function
        "dt" : 0.02,           # With 100 time points time values will range from 0 to␣
↪2
    }
    params = {
        "amp1" : [1, 0.5],     # Amplitude
        "r1" : [1.0, 0.8],     # Decay rate
    }
    test_config = {
        "nt" : 100,            # Number of time points
        "noise" : 0.1,         # Amplitude of Gaussian noise to add to simulated data
        "patchsize" : 20,      # Each patch is 20 voxels along each dimension
    }
    result, log = self_test("exp", rundata, params, save_input=save, save_output=save,
↪ invert=True, **test_config)
except FabberException, e:
    print e.log
    traceback.print_exc()
except:
    traceback.print_exc()
```

The test script generates a test Nifti image containing 'patches' of data chequerboard style, each of which corresponds to a combination of true parameter values. As Fabber is designed to work on 3D timeseries data you can only vary three model parameters in each test - others must have fixed values.

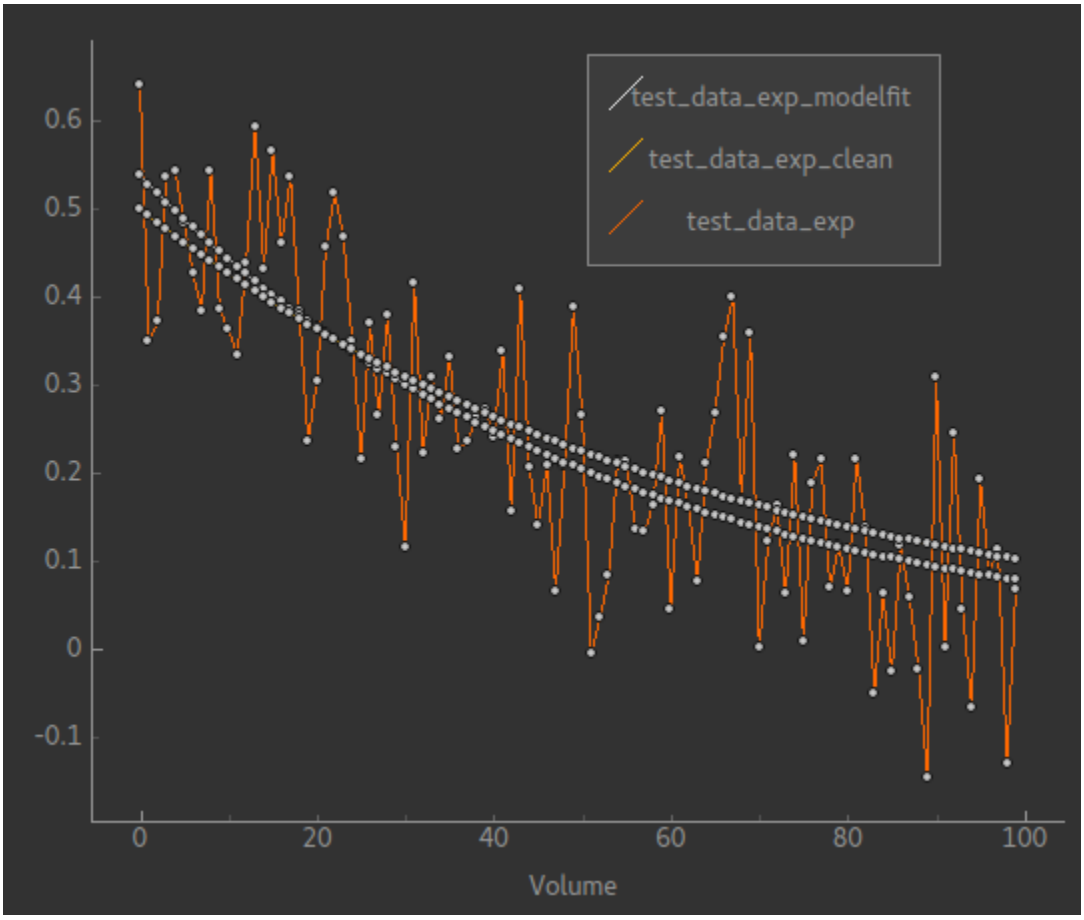
The test data is generated both 'clean' and with added Gaussian noise of specified amplitude. The model is then run on the noisy data to determine how closely the true parameter values can be recovered. In this case we get the following output:

```
python test_single.py --save

Running self test for model exp
Saving test data to Nifti file: test_data_exp
Saving clean data to Nifti file: test_data_exp_clean
Inverting test data - running Fabber: 100%

Parameter: amp1
Input 1.000000 -> 0.999701 Output
Input 0.500000 -> 0.500674 Output
Parameter: r1
Input 1.000000 -> 1.000728 Output
Input 0.800000 -> 0.801230 Output
Noise: Input 0.100000 -> 0.099521 Output
```

For each parameter, the input (*ground truth*) value is given and also the mean inferred value across the patch. In this case it has recovered the parameters pretty well on average. An example plot of a single voxel might look like this:



The orange line is the noisy data it's trying to fit while the two smooth lines represent the 'true' data and the model fit. In fact for this example typically the model fit is much closer to the true data - we have chosen this voxel as an example so it is possible to see them separately!

5.5 Testing the model - bi-exponential

Fitting to a single exponential is not too challenging - here we will test fitting to a bi-exponential where there are two different decay rates. We will find that we need to improve the model to get a better fit.

First we can modify the test script to test a bi-exponential (`test_biexp.py` in examples):

```
#!/bin/env python

import sys
import traceback

from fabber import self_test, FabberException

save = "--save" in sys.argv
try:
    rundata= {
        "model" : "exp",
        "num-exps" : 2,
        "dt" : 0.02,
```

(continues on next page)

(continued from previous page)

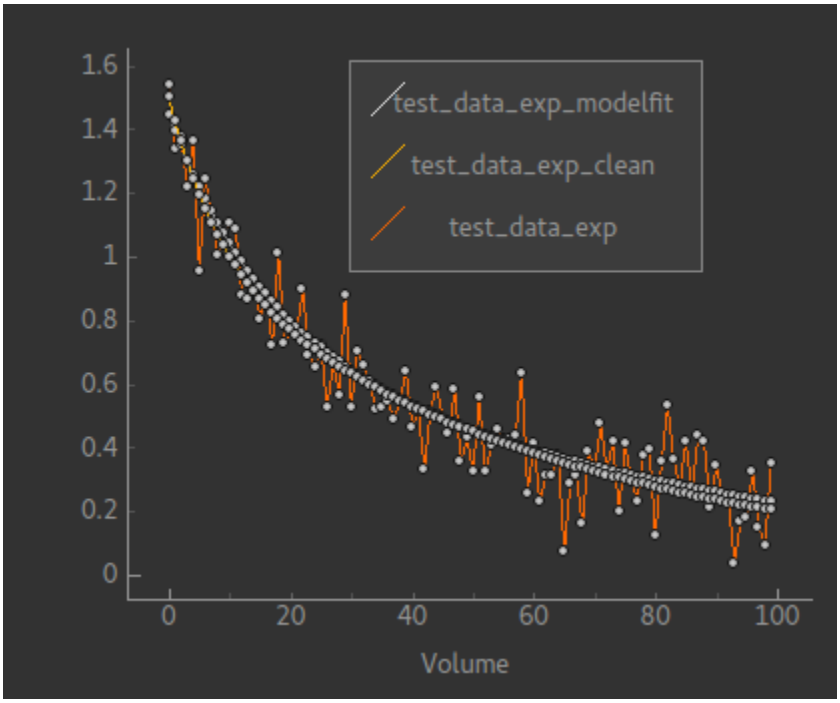
```
        "max-iterations" : 50,
    }
    params = {
        "amp1" : [1, 0.5],     # Amplitude first exponential
        "amp2" : 0.5,          # Amplitude second exponential
        "r1" : [1.0, 0.8],     # Decay rate of first exponential
        "r2" : 6.0,            # Decay rate of second exponential
    }
    test_config = {
        "nt" : 100,            # Number of time points
        "noise" : 0.1,         # Amplitude of Gaussian noise to add to simulated data
        "patchsize" : 20,      # Each patch is 20 voxels along each dimension
    }
    result, log = self_test("exp", rundata, params, save_input=save, save_output=save,
↪ invert=True, **test_config)
except FabberException, e:
    print e.log
    traceback.print_exc()
except:
    traceback.print_exc()
```

This is similar to the last test but we have set `num-exps` to 2 and added parameters for a fixed second exponential curve with a faster decay rate. If we run this we get output something like this:

```
python test_biexp.py --save
Running self test for model exp
Saving test data to Nifti file: test_data_exp
Saving clean data to Nifti file: test_data_exp_clean
Inverting test data - running Fabber: 100%

Parameter: amp1
Input 1.000000 -> 0.633822 Output
Input 0.500000 -> 0.309912 Output
Parameter: r1
Input 1.000000 -> 19693700210770313216.000000 Output
Input 0.800000 -> -324689116576874496.000000 Output
Noise: Input 0.100000 -> 0.150277 Output
```

This isn't looking too encouraging. If we examine the model fit against the data we find that actually most voxels have fitted quite well:



However a few voxels have ended up with very unrealistic parameter values. This kind of behaviour is a risk with model fitting - in trying to find the best solution the inference can end up finding a local minimum which is a long way from the true minimum.

We will show two additions we can make to our model to improve this behaviour.

5.5.1 Initialising the posterior

The initial posterior is a 'first guess' at the parameter values and can be based on the data. Fabber models can use their knowledge of the model to make a better guess by overriding the `InitVoxelPosterior` method. We firstly add this method to `fwdmodel_exp.h`:

```
void InitVoxelPosterior(MVNDist &posterior) const;
```

Now we implement it in `fwdmodel_exp.cc`:

```
void ExpFwdModel::InitVoxelPosterior(MVNDist &posterior) const
{
    double data_max = data.Maximum();

    for (int i=0; i<m_num; i++) {
        posterior.means(2*i+1) = data_max / (m_num + i);
    }
}
```

Our implementation only affects the amplitude and sets an initial guess so that the sum of all our exponentials is close to the maximum data value. Note that we make the posterior means different for each exponential - this helps break the symmetry of the inference problem.

5.5.2 Parameter transformations

A major reason for the failure of some voxels to fit is that the decay rate in particular could become negative, generating an exponential increase curve which may be so far away from the data that it does not successfully converge back to the correct value. In many models we want to restrict parameters to positive values to prevent this sort of unphysical solution. One way to do this is to use a log-transform of the parameter (i.e. assuming the parameter takes a log-normal distribution rather than a standard Gaussian). We can do this by modifying `GetParameterDefaults` as follows:

```
void ExpFwdModel::GetParameterDefaults(std::vector<Parameter> &params) const
{
    params.clear();

    int p=0;
    for (int i=0; i<m_num; i++) {
        params.push_back(Parameter(p++, "amp" + stringify(i+1), DistParams(1, 100),␣
↪DistParams(1, 100), PRIOR_NORMAL, TRANSFORM_LOG()));
        params.push_back(Parameter(p++, "r" + stringify(i+1), DistParams(1, 100),␣
↪DistParams(1, 100), PRIOR_NORMAL, TRANSFORM_LOG()));
    }
}
```

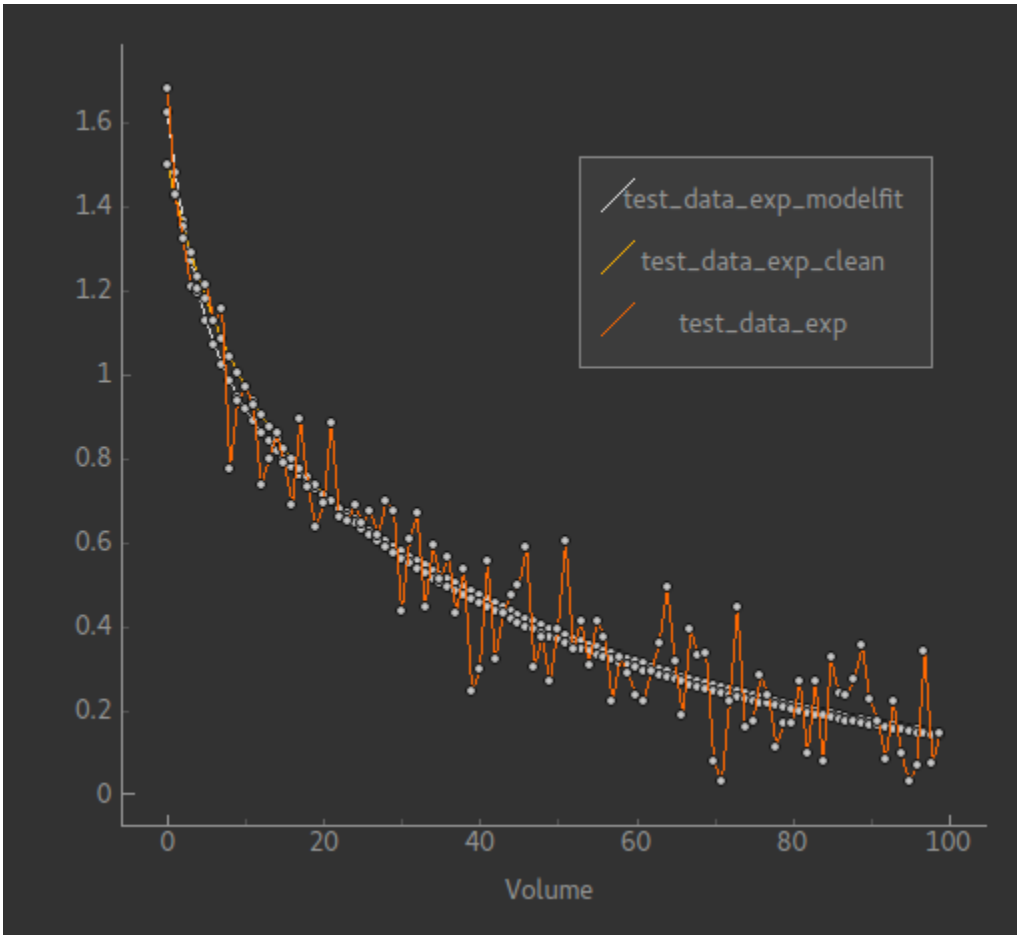
(we also need to add `#include <fabber_core/priors.h>` at the top of `fwdmodel_exp.cc`.

With these changes we still retain some bad fitting voxels but fewer than previously. The output of the test script is now:

```
python test_biexp.py --saveike this::
Running self test for model exp
Saving test data to Nifti file: test_data_exp
Saving clean data to Nifti file: test_data_exp_clean
Inverting test data - running Fabber: 100%

Parameter: amp1
Input 1.000000 -> 0.714108 Output
Input 0.500000 -> 0.498471 Output
Parameter: r1
Input 1.000000 -> 4.898833 Output
Input 0.800000 -> 4.674414 Output
Noise: Input 0.100000 -> 0.099399 Output
```

So we clearly have a reduction in the number of extreme values. In this case we can't actually trust the self-test output because sometimes the inference 'swaps' the exponentials around making `amp1` = `amp2` and `r1` = `r2`. But viewing the model fit visually shows sensible fitting in the overwhelming majority of voxels:



5.6 Changing the example to your own model

To summarize, these are the main steps you'll need to take to change this example into your own new model:

- Edit the `Makefile` to change references to `exp` and `Exp` to the name of your model
- Rename source files, e.g. `fwdmodel_exp.cc` -> `fwdmodel_<mymodel>.cc`
- Add your model options to the options list in the `.cc` file
- Add any model-specific private variables in the `.h` file
- Implement the `Initialize`, `GetParameterDefaults`, `Evaluate` methods for your model.
- If required, implement `InitVoxelPosterior`

CHAPTER 6

Theory behind Fabber

Fabber uses a technique called *Variational Bayes* to perform Bayesian inference in a computationally efficient way. This section gives a brief overview of the theory behind Fabber, its advantages and limitations. For a fuller description see[1] .

6.1 Forward model

The forward model M varies according to application, and produces a predicted *time series* for a set of parameter values P_n:

$$S(t) = M(t; P_0, P_1, P_2, ...)$$

A model may have any number of parameters, however we are principally interested in those whose values we wish to estimate (infer) from the data. For example in a CASL model the bolus duration is a parameter of the model which predict the ASL signal but we may choose to regard this as fixed by the acquisition sequence and not infer it.

From here we will use the term *parameter* only for parameters of the model whose values we intend to infer.

6.2 'Best Fit' modelling

One conventional approach to this problem is to calculate the 'best fit' values of the parameters. This is done relative to some *cost function* for example the squared difference between the model prediction and the actual data in non-linear least-squares (NLLS) fitting. The partial derivatives of the model prediction for each parameter are calculated numerically and used to change their values to reduce the cost function. When a minimum has been achieved to within some tolerance, the resulting parameter values are returned.

[1] *Chappell, M.A., Groves, A.R., Woolrich, M.W., "Variational Bayesian inference for a non-linear forward model", IEEE Trans. Sig. Proc., 2009, 57(1), 223–236.*

6.3 Linear modelling

For some kinds of forward model, a pseudo-linear approach can be used where certain features of the data such as the mean are assumed to be linearly dependent on a model parameter. For example if the model predicts a peak, it may have a parameter which (approximately) determines the height of the peak, another which is related to the peak width, another which affects the time position of the peak and another which adds a constant signal offset. These parameters can then be related to measurable features of the data and estimated independently.

The effectiveness of this kind of approach depends on the model and in general it is less reliable with more complex nonlinear models.

6.4 Bayesian inference

In the Bayesian picture of inference, we always start with some *prior* distribution for the value of each parameter. This represents what we know about the parameter's value *before* we have seen the data we are fitting it to.

The prior may be based on experimental evidence in which case it may have a mean (the accepted experimental value) and a limited variance (reflecting the range of values obtained in previous experiments). This is an example of an *informative* prior.

Alternatively if the value of the parameter could vary by a very wide degree we our prior may be given some default mean with very large variance that effectively allows it to take on any possible value. This is an *uninformative* prior.

Prior distributions should not be informed by the data we are fitting, instead Bayesian inference calculates a *posterior* distribution for each parameter value which takes into account both our prior knowledge (if any) and what the data says. Mathematically this is based on Bayes's theorem:

$$P(parameters \mid data) = \frac{P(data \mid parameters)\, P(parameters)}{P(data)}$$

Here $P(parameters \mid data)$: is the posterior distribution of the parameter values given the data we have. $P(parameters)$ is the prior distribution of the parameter values. $P(data \mid parameters)$ is the probability of getting the data we have given a set of parameter values and is determined from the forward model together with some model of random noise. This term is known as the *likelihood*. The final term $P(data)$ is known as the *evidence*, and can often be neglected as it only provides a normalization of the posterior distribution.

So, rather than determining the 'best fit' values of model parameters, the output is a set of posterior distributions for each parameter which include information about the predicted mean value, and also its variance. If the variance of a parameter's posterior distribution is high this means it's value is not precisely determined by the data (i.e. there are a range of probable value which are consistent with the data), whereas if the variance of the posterior is low the value is well determined by the data (and/or the prior).

Advantages of the Bayesian approach include:

- The ability to incorporate prior information into our inference. For example this allows us to treat a tissue T1 value as a parameter in the model and constrain the inference by the known range of expected T1 values. In conventional model fitting we would either need to fix the value of T1 or allow it to vary in any way to fit the data, potentially resulting in physically unlikely parameter values (which nevertheless fit the data well!)
- Related to the above, we can potentially fit models which are formally 'over specified' by parameters (i.e. where there are different combinations of parameter values which produce identical output). Because of the priors, these different combinations of values will not be equally likely and hence we can choose between them.
- The output includes information about how confident we can be in the parameter values as well as the values themselves.
- It is possible to incorporate other kinds of prior information into the inference, for example how rapidly a parameter may vary spatially.

The 'gold standard' approach to Bayesian inference is the Markov chain Monte Carlo (MCMC) method where the posterior distribution is determined by sampling the prior distributions and applying Bayes's theorem. However this is generally too computationally intensive to use routinely with problems involving 10^5 - 10^6 voxels as in fMRI applications.

6.5 Variational Bayes

The method used in fabber makes a variational approximation which nevertheless is able to reproduce the results of MCMC closely. One consequence of this is that the prior and posterior distributions must be of the same type. In the Fabber approach these are modelled as a multi-variable Gaussian distributions which are characterised by parameter means, variances and covariances.

The key measure used by Fabber in fitting the model is the `Free energy` which is effectively the negative of the Kullback-Leibler divergence between the approximate multi-variate Gaussian posterior and the `true` posterior distribution. In[1] an expression for the free energy based on this form of the posterior is derived, along with a set of update equations to maximise the free energy (and hence minimise the KL divergence) by varying the parameter means, variances and covariances.

6.6 References

Referencing Fabber

If you use Fabber in your research, please make sure that you reference at least Chappell et al 2009[1], and ideally[2] and[3] also.

[1] *Chappell, M.A., Groves, A.R., Woolrich, M.W., "Variational Bayesian inference for a non-linear forward model", IEEE Trans. Sig. Proc., 2009, 57(1), 223–236.*

[2] *Woolrich, M., Chiarelli, P., Gallichan, D., Perthen, J., Liu, T. "Bayesian Inference of Haemodynamic Changes in Functional ASL Data", Magnetic Resonance in Medicine, 56:891-906, 2006.*

[3] *Groves, A. R., Chappell, M. A., & Woolrich, M. W. (2009). "Combined spatial and non-spatial prior for inference on MRI time-series." NeuroImage, 45(3), 2009.*doi:10.1016/j.neuroimage.2008.12.027.