
SVB Documentation

Release 0.0.1

Martin Craig

Mar 03, 2021

Contents:

1	Stochastic Variational Bayes - Theory	2
2	Implementation of SVB	5
3	Tests using Biexponential model	11
4	Tests using Arterial Spin Labelling model	47
5	Sample size inflation tests	50
6	Learning rate quenching tests	55
7	Command line usage	61
8	Python API	62
	Python Module Index	72
	Index	73

SVB is a package to perform stochastic Bayesian inference on a nonlinear forward model (i.e. a parameterised model which is able to predict data values from a set of parameter values).

The implementation leverages the TensorFlow framework to perform efficient optimisation of the model parameters given an experimental data set.

Stochastic Variational Bayes - Theory

Stochastic Variational Bayes is a method of performing Bayesian inference on the parameters of a generative model given a data set assumed to be generated by the model with unknown additive noise.

The description below is a highly abbreviated account of the theory behind SVB. For more detailed derivations, see the references cited.

For interactive tutorials implementing variational Bayesian inference on simple examples, see the [Variational Bayes tutorial](#)

1.1 Bayesian inference

Bayes' theorem is a general statement about how ones belief about the value distribution of a random variable should be updated in the light of new data. In the context of model parameter inference it can be described as follows:

$$p(\theta | y) = q(\theta) = \frac{p(y | \theta) p(\theta)}{p(y)}$$

Here θ is the set of parameters of the model which we wish to infer.

$p(\theta | y) = q(\theta)$ is the *posterior* distribution, i.e. the inferred distribution of the model parameters θ given the data y .

$p(\theta)$ is the *prior* probability of the model parameters θ . This describes the distribution we believe the parameters would follow before any data has been seen and might reflect, for example, existing estimates of physiological parameters or other constraints (e.g. that a fractional parameter must lie between 0 and 1).

$p(y | \theta)$ is the *likelihood*, i.e. the probability of getting the data y from a given set of model parameters θ . This is determined by evaluating the model prediction using the parameters θ and comparing it to the data. The difference between the two must be the result of noise, and the likelihood of the noise can be calculated from the noise model.

$p(y)$ is the *evidence* and is chiefly used when comparing one model with another. For an inference problem using a single model it can be neglected as it is independent of the parameters and simply provides a normalizing constant.

1.2 Variational Bayes

The general Bayesian inference problem can, in general, only be solved by a sampling method such as Markov Chain Monte Carlo (MCMC) where random samples are generated in such a way that, through Bayes' theorem, they gradually provide a representative sample of the posterior distribution. Any properties of the posterior, such as mean and variance, can be calculated from the sample once it is large enough to be representative.

MCMC, however, is extremely computationally intensive especially for the kind of applications we are concerned with where we may be fitting between 2 and 20 parameters independently at typically 10^5 voxels. Variational Bayes is an approximate method which re-formulates the inference problem in the form of a variational principle, where we seek to maximise the *Free Energy*.

$$F(\theta) = \int q(\theta) \log \left(p(y | \theta) \frac{p(\theta)}{q(\theta)} \right) d\theta$$

Again θ is the set of model parameters, $q(\theta)$ is the posterior distribution, $p(\theta)$ is the prior distribution and $p(y | \theta)$ is the likelihood of the data given the parameters.

For completely general forms of the prior and posterior distributions, this integral is expensive to compute numerically (and is unlikely to be solvable analytically). However the advantage of the variational approach is that simplified forms can be chosen for the prior and posterior such that the free energy can be calculated and optimized efficiently. The variational principle guarantees that the free energy calculated using this method will be a lower bound on the 'true' free energy and therefore the closest approximation we can find using our simplified distributions.

Typically we assume multivariate Gaussian distributions for the prior and posterior, and a noise model based on a Gaussian or Gamma distribution.

One form of variational Bayes uses the calculus of variations to derive a set of update equations for the model and noise parameters which can then be iterated until convergence¹. However this method requires particular choices of the prior and posterior distributions, and the noise model, and thus lacks flexibility. Any change to these distributions requires the update equations to be re-derived.

1.3 Stochastic variational Bayes

The free energy equation can be slightly re-written in the form of an expectation over the posterior distribution $q(\theta)$:

$$F(\theta) = E_{q(\theta)} [\log(p(y | \theta))] - E_{q(\theta)} \left[\log \left(\frac{q(\theta)}{p(\theta)} \right) \right]$$

This suggests an alternative calculation method based on taking a *sample* of values from the posterior distribution. If this sample is large enough to be representative of the distribution, the expectation integrals from above can be approximated by the mean over the samples:

$$E_{q(\theta)} [\log(p(y | \theta))] \approx \frac{1}{S} \sum_s \log(p(y | \theta^s))$$

$$E_{q(\theta)} \left[\log \left(\frac{q(\theta)}{p(\theta)} \right) \right] \approx \frac{1}{S} \sum_s \left[\log \left(\frac{q(\theta^s)}{p(\theta^s)} \right) \right]$$

Where we have S samples of the full set of parameters, denoted θ^s .

The first of these terms is the negative of the *reconstruction loss* and is a measure of how well the model prediction fits the data.

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.

The second term is the *latent loss* and measures the closeness of the posterior to the prior. In fact it is the Kullback-Leibler (KL) divergence between the prior and posterior distributions.

This is more tractable than a numerical integration *provided* we can obtain a representative sample from the posterior. Maximisation of the free energy can then be done using a generic framework such as those developed for machine learning applications which have the ability to automatically calculate gradients of an objective function from a defined set of calculation steps.

1.3.1 Alternative forms of the latent loss

The latent loss term can be alternatively written as follows, removing the stochastic approximation for part of the log:

$$E_{q(\theta)} \left[\log \left(\frac{q(\theta)}{p(\theta)} \right) \right] \approx E_{q(\theta)} \left[\log(q(\theta)) \right] - \frac{1}{S} \sum_s \left[\log(p(\theta^s)) \right]$$

The first term is the *entropy* of the posterior distribution. For many distributions this can be calculated analytically without reference to a sample, so we may be able reduce our dependence on the choice of sample to some degree.

If both the prior and posterior are multivariate Gaussian distributions, we can go further and obtain a fully analytic expression for the latent loss using the known result for the KL divergence of two MVNs²:

$$E_{q(\theta)} \left[\log \left(\frac{q(\theta)}{p(\theta)} \right) \right] = \frac{1}{2} \left\{ \text{Tr}(\Sigma_p^{-1} \Sigma_q) + (\mu_p - \mu_q)^T \Sigma_p^{-1} (\mu_p - \mu_q) - N + \log \left(\frac{\det \Sigma_p}{\det \Sigma_q} \right) \right\}$$

Here N is the number of parameters in θ , and $\mu_p, \Sigma_p, \mu_q, \Sigma_q$ are the mean and covariance of the prior and posterior.

1.3.2 Obtaining the sample from the posterior

The problem of sampling from the posterior is of some significance. If the optimization is to work effectively it would be helpful if the gradients of the sample values with respect to the variable parameters could be calculated. However this is difficult if we simply obtain a random sample from, for example, a Gaussian of given mean and variance. For Gaussian distributions, one way around this is known as the *reparameterization trick*. We obtain a sample from a *fixed* Gaussian (e.g. $N(0, 1)$) and then scale the values using the (variable) mean and variance of the posterior distribution. This enables the gradients to be used in the optimization algorithm. The disadvantage of the method is that it does not immediately generalise to other kinds of distributions.

1.3.3 Advantages of the stochastic approach

The main advantage of the stochastic approach is that the requirements on the prior and posterior distributions are greatly reduced. The prior distribution needs to be able to generate log probabilities for a set of parameters, the posterior needs to be able to generate samples and its own entropy, and we need some means of calculating the data likelihood - this normally involves a noise model which can calculate the probability of the observed deviations between a model prediction and the actual data. Although we can take advantage of analytic results for Gaussian distribution, the actual forms of the distributions are not constrained by the method (apart from the limitation of not always being able to use the reparameterization trick).

1.4 References

² http://web.stanford.edu/~jduchi/projects/general_notes.pdf

2.1 Use of TensorFlow

The maximisation of the free energy with respect to the parameters of the posterior distribution is implemented using the TensorFlow library which provides efficient calculation of functions of multidimensional arrays.

TensorFlow uses a compiled-graph model where the elements in a calculation are represented as nodes of a graph and dependencies are represented as edges. The graph is set up prior to the actual calculation being performed. In this model, nodes are tensors (multidimensional arrays) or operations which take one or more tensors and output one or more tensor outputs. For example a matrix multiplication operation would take two tensors and output one.

For operations such as these, TensorFlow is designed to operate efficiently on large batches of data, so for example a tensor with dimensions $[1000, 5, 10]$ can be interpreted as 1000 instances of a 5×10 matrix. This can then be matrix-multiplied by another tensor of dimensions $[1000, 10, 8]$ resulting in an output with the same dimensions of $[1000, 5, 8]$ interpreted as the 1000 output matrix products. This system is well suited to our problem where we will be performing calculations on around 10^5 voxels simultaneously.

Operations can also reduce the dimensions of a tensor, for example by calculating the sum or mean across an axis, or all axes, reshape tensors (transpose axes), extract subsets of tensors and perform mathematical operations such as elementwise log, square, etc.

Tensors which are not the output of an operation may be set to some constant value, however they may also be marked as *variables*. When elements of a tensor are marked as variable they can be changed by an optimization operation in order to minimise some *cost function*, defined by a set of operations on the tensors in the graph. In our case since we seek to maximise the free energy we take the cost function as its negative.

Optimizer operations work by calculating the gradient of the cost function with respect to all variables (using the back-propagation technique) and performing a gradient-based minimisation algorithm. This minimisation can be carried out iteratively until the cost function is determined to have converged to some predefined extent.

Variables and constant values can be freely mixed in a calculation, for example one can construct a matrix in which the diagonal elements are variable but the off-diagonal elements are fixed constants.

2.2 Main elements in the calculation graph

The elements in the calculation graph are implemented as Python classes which, contain methods to set up the relevant tensors and operations required to perform their function. We try to avoid assumptions about the nature of these operations in order to increase flexibility, e.g. we do not constrain how the posterior generates samples of values.

In TensorFlow, keeping track of the dimensions of tensors is critical to error-free execution! In the following sections we use the following symbols to maintain clarity:

- V is the number of voxels.
- B is the number of time points in each voxel during optimization.
- S is the number of samples drawn from the posterior in order to approximate integrals over the posterior in the stochastic method.
- P is the number of parameters in the generative model, including any parameters required to model (including any parameters needed to model the noise component).

Note: For clarity with our intended application to modelling timeseries of volumetric data, we refer to *voxels* and *time points*. However more generally voxels can be thought of as independent instances of the data being modelled, and time points could be any kind of series of data values.

Note: B may not be the full set of time points available in the data - see *Mini-batch processing* below.

2.2.1 Prior

In the stochastic variational Bayes method, we need to be able to integrate the expected log PDF of the prior distribution over the posterior, and this is calculated using a (random, but hopefully representative) sample of values from the current posterior.

A prior must therefore provide an operation node which takes a sample tensor of dimension $[V, P, S]$ and returns a tensor of dimensions $[V]$ which contains the mean log PDF for each voxel.

```
def mean_log_pdf(self, samples):
    """
    :param samples: A tensor of shape [V, P, S] where V is the number
                    of voxels, P is the number of parameters in the prior
                    (possibly 1) and S is the number of samples

    :return: A tensor of shape [V] where V is the number of voxels
             containing the mean log PDF of the parameter samples
             provided
    """
```

2.2.2 Posterior

The posterior must be able to provide samples from itself, i.e. it must provide an operation node which takes a sample size parameter, S and returns a tensor of dimension $[V, P, S]$ which returns S samples for each of P parameters at each of V voxels.


```
def sample(self, nsamples):
    """
    :param nsamples: Number of samples to return per voxel / parameter

    :return: A tensor of shape [V, P, S]`` where V is the number
             of voxels, P is the number of parameters in the distribution
             (possibly 1) and S is the number of samples
    """
```

In addition the posterior must be able to calculate the expectation integral of the log PDF over its own distribution. This is by definition the entropy of the distribution and therefore in many cases it can be calculated without reference to a sample. However the sample is available if it is required. This must provide an operation which returns a tensor of dimension [V] containing the entropy at each voxel.

```
def entropy(self, samples):
    """
    :param samples: A tensor of shape [V, P, S] where V is the number
                    of voxels, P is the number of parameters in the prior
                    (possibly 1) and S is the number of samples.
                    This parameter may or may not be used in the calculation.

    :return Tensor of shape [V] containing voxelwise distribution entropy
    """
```

2.2.3 Generative model

The job of the model is to provide a predicted set of data points given a set of parameters. However in the stochastic method it must provide a full prediction for each time point in the input data for each sample of parameter values derived from the posterior.

Hence we require an operation which takes a tensor of dimension $[P \times V \times S \times 1]$ containing the values of the parameters for each sample at each voxel and a tensor of dimension $[V, 1, B]$ of time points at each voxel and outputs a tensor of dimension $[V, S, B]$ containing the model prediction at each time point for each sample at each voxel.

```
def evaluate(self, params, t):
    """
    Evaluate the model

    :param t: Time values to evaluate the model at, supplied as a tensor of shape
              [1x1xB] (if time values at each voxel are identical) or [Vx1xB]
              otherwise.

    :param params Sequence of parameter values arrays, one for each parameter.
                  Each array is VxSx1 tensor where V is the number of voxels and
                  S is the number of samples per parameter. This
                  may be supplied as a PxVxSx1 tensor where P is the number of
                  parameters.

    :return: [VxSxB] tensor containing model output at the specified time values
              for each voxel, and each sample (set of parameter values).
    """
```

Note: The dimensions of the sampled input parameter values are transposed from those returned by the posterior. This is because it is more convenient for the model to have the parameter index first so individual parameter tensors

can easily be extracted by indexing. This is helpful as different parameters typically play different roles in the model.

Note: The dimension of size 1 in the input parameter values is designed to align with the last dimension in the time points tensor (of size B) to allow the parameter values to be broadcast across all time points. Similarly the dimension of size 1 in the time points tensor allows the same set of time points to be broadcast across the sample dimension S.

Note: The time points may be identical at all voxels, in which case a time point tensor of shape $[1, 1, B]$ may be provided instead. This can typically be handled automatically by broadcasting.

2.2.4 Noise model

The noise model is required to calculate the mean log likelihood of the data over the sampled values, given the prediction returned by the model. It must define an operation which takes the actual data tensor with dimensions $[V, B]$, the model prediction tensor $[V, S, B]$ and the sampled values of the noise parameter $[V, S]$. The operation must return a voxelwise mean log likelihood tensor with dimensions $[V]$.

```
def log_likelihood(self, data, pred, noise, nt):
    """
    Calculate the log-likelihood of the data

    :param data: Tensor of shape [V, B]
    :param pred: Model prediction tensor with shape [V, S, B]
    :param noise: Noise parameter samples tensor with shape [V, S]
    :return: Tensor of shape[V] containing mean log likelihood of the
            data at each voxel with respect to the noise parameters
    """
```

Note: Currently we are assuming a single noise parameter. This will be relaxed in future and the input noise parameter tensor will have dimension $[V, S, P_n]$ where P_n is the number of noise parameters

2.2.5 Cost function

The total loss is defined by summing two loss tensors. The *Reconstruction loss* is the negative of the mean log likelihood returned by the noise model. This is essentially a measure of how well the model prediction fits the data. The *Latent loss* is the posterior distribution entropy minus the mean log PDF of the prior and penalises large deviations from the prior values of the parameters. Each of these is defined by operations on the tensors returned by the prior, posterior and noise models and has dimension $[V]$.

Note: If both prior and posterior are multivariate Gaussian distributions an analytic expression for the latent loss is available which does not require the use of a sample. In this case we use this instead of the calculation described above, and an additional operation is defined in the MVN posterior for this.

The final cost function is then defined as a mean over voxels of the loss tensor, i.e. a scalar. This is to ensure that the optimizer has a single value to optimize the parameters over.

2.3 Optimization strategy

Optimization is performed using the `AdamOptimizer`, a gradient based optimization algorithm that seeks to minimise the given cost function. The key parameter in configuring the optimizer is the *learning rate* which determines the size of the step in parameter space that the optimizer takes in order to reduce the cost function. High learning rates move further and may therefore reduce the cost function more quickly, however they may also ‘overshoot’ the actual minimum and fail to converge, or find a local minimum instead. Low learning rates by contrast move more cautiously towards the minimum however the resulting convergence may be too slow to be useful.

Unfortunately there is no obvious way to select the optimal learning rate for a given problem. Typically in machine learning applications a process of trial and error is involved, with learning curves used as a way to assess the convergence. This is not too problematic as the training is often a one-off or occasional step with the trained model then re-used for multiple applications. In our case, however, we need to train a model for each application (data set) we process and the ability to select a suitable learning rate is critical. For this reason we will need to devote some effort to identifying how to select this parameter for the kind of data we face.

Optimization is divided into *Epochs*, each of which involves the entire data set being processed and the parameters and cost function updated. This can be done in a single iteration of the optimizer, passing all the data in, however it is also possible to use a *mini-batch* method which can offer some advantages.

2.3.1 Mini-batch training

In mini-batch training, the data set is divided into chunks and an optimization step is performed for each chunk. When all chunks have been processed an epoch is complete and we start the next epoch with the first chunk again.

There are two main potential advantages to mini-batch training:

1. *Efficiency* - the information contained in the data set does not scale linearly with the number of points included, whereas the computational effort often does. Processing half of the data may take half as much time and yet yield an optimization iteration nearly as effective as processing the full data. An epoch is then be comprised of two optimization iterations rather than one. which should mean faster convergence by epochs.
2. *Increasing noise* - One danger of gradient based optimization is local minima. A way to reduce the likelihood of the optimization getting stuck in one is to introduce an element of noise to the gradients so the optimization will explore a wider range of parameter space during the minimization. Smaller batches of data will give noisier gradients and may help alleviate this problem to some extent.

While this is persuasive it is important to recognize that assumptions about the efficiency of an optimization must be tested in practice. Typical machine learning applications often have extremely large numbers of training examples and often use mini-batch sizes of 20-50. In our case the number of time points in real data rarely exceeds 100 so it may be the case that mini-batch training is only useful for larger data sets.

A mini-batch can be extracted from the data in two main ways, either by dividing up the data into sequential chunks or by taking strided subsamples through the data. The latter seems more appropriate when the data forms a continuous timeseries since we are always using information from across the time series, however for the same reason the former method may be preferred when our data consists of repeated blocks of measurements of the same timeseries (as is sometimes the case for ASL data). Our implementation supports both via the `sequential_batches` parameter.

One factor that needs to be accounted for when doing mini-batch training is the scaling of different contributions to the total cost. The latent loss depends only on the prior and posterior distributions and not on the size of the training data, however the reconstruction loss is a sum of log probabilities over the points in the training data. Correct Bayesian inference only occurs when this is scaled by $\frac{N_t}{N_b}$ where N_t is the number of time points in the full data and N_b is the number in the mini-batch, i.e. the batch is being used to estimate the reconstruction loss for the full data set.

2.3.2 Learning rate quenching

There is no requirement to keep the learning rate constant throughout the optimization. It can, and often is, changed after each epoch or training iteration. One simple strategy is to gradually reduce ('quench') the learning rate, starting off with a high value that quickly explores the parameter space, and reducing it to home in on the minimum with high accuracy. Currently we have a very simple implementation of this idea using the following parameters:

- `max_trials` If this number of epochs passes without the cost function improving over the previous best, the learning rate will be reduced
- `quench_rate` - a factor to reduce the learning rate by (e.g. 0.5 means the learning rate will be halved)
- `min_learning_rate` - The learning rate will never be reduced lower than this value

This scheme gives us some freedom to start with relatively high learning rates and reduce them if they are not getting us anywhere. We also adopt the same strategy where a numerical error is detected. Often this occurs when parameters stray out of 'reasonable' ranges, suggesting an excessively large optimization step. In this case we reset to the previous best cost state and reduce the learning rate by `quench_rate` and continue.

It is worth noting that this is far from being the only strategy for modifying learning rates during training, not is it an agreed best practice! Other ideas include:

- Starting with a low learning rate and *increasing* it until the cost stops decreasing, thus determining an optimal learning rate which is then selected.
- Cycling the learning rate to explore a varied region of parameter space and aid escape from local minima (possibly combined with quenching over time)
- Increasing the batch size rather than the learning rate to reduce gradient noise as convergence is approached.

It remains to be seen if any of these strategies are useful in our application - again they are typically the product of machine learning applications which, although they resemble our problem in some ways, differ greatly in others so not all recommended strategies may be useful.

2.3.3 Voxelwise convergence

Our implementation seeks to minimise the mean cost over all voxels, however it is clear in practice that some voxels converge more rapidly than others. If we can identify 'converged' voxels and exclude them from the calculation in subsequent epochs we may attain overall convergence faster (or alternatively be able to use larger numbers of epochs to ensure we converge 'difficult' voxels without penalising our runtime too much).

Two ways we might accomplish this are:

- A voxelwise mask which selects out a subset of the data for cost calculation. This would need to be applied at an early stage in the calculation graph in order to save computational time.
- 'Zeroing' the gradients of converged voxels so they do not contribute to the minimisation.

We have not attempted to implement these strategies yet because currently we want to understand convergence generally and are less concerned with absolute performance. However this would be useful to investigate as we start to apply the method to real examples.

Tests using Biexponential model

The biexponential model outputs a sum of exponentials:

$$M(t) = A_1 \exp(-R_1 t) + A_2 \exp(-R_2 t)$$

The model parameters are the amplitudes A_1 , A_2 and the decay rates R_1 and R_2 .

Although the model is straightforward it can be challenging as an inference problem as the effect of the two decay rates on the output is nonlinear and can be difficult to distinguish in the presence of noise.

3.1 Test data

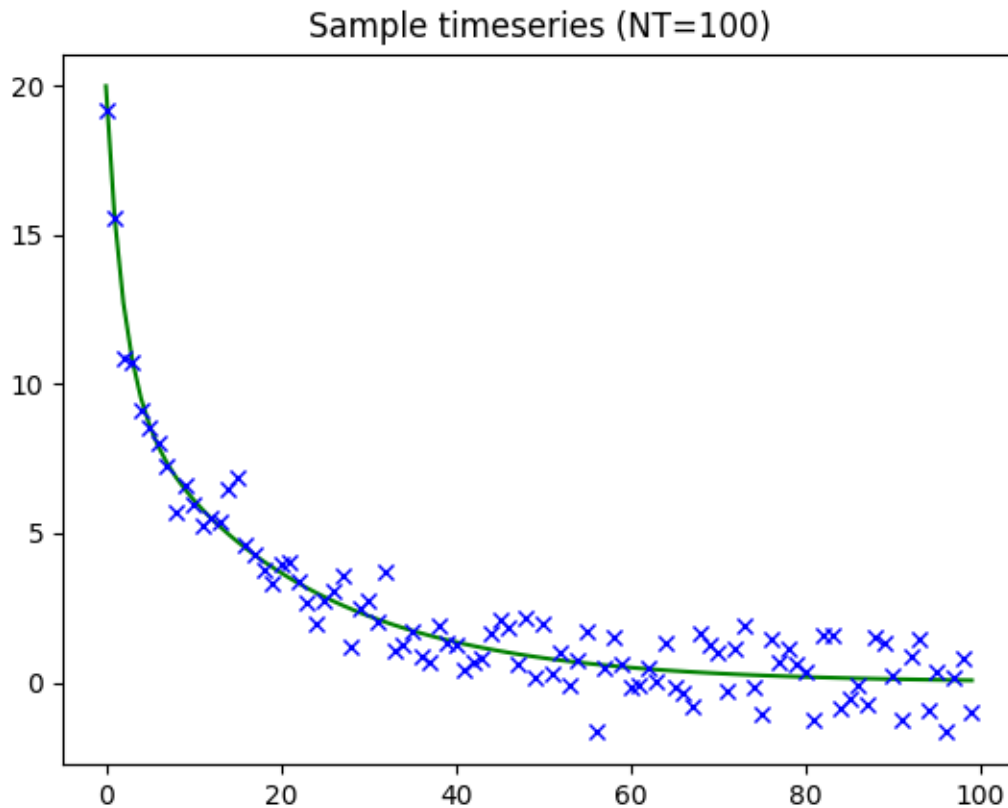
For testing purposes we define the ground truth parameters as:

- $A_1 = 10$
- $A_2 = 10$
- $R_1 = 1$
- $R_2 = 10$

The variables within the test data are:

- The level of noise present. For this test data we use Gaussian noise with a standard deviation of 1.0.
- The number of time points generated. We generate data sets with 10, 20, 50 and 100 time points (in each case the value of t ranges from 0 to 5 so only the data resolution changes in each case)

An example timeseries with these parameters is show below (100 time points, ground truth overlaid on noisy data):



1000 timeseries instances were generated and used for each test.

One issue with the biexponential model is that there are always two equivalent solutions obtained by exchanging A_1, R_1 with A_2, R_2 . To prevent this from confusing reports of mean parameter values, we normalize the results of each run such that in each voxel A_1, R_1 is the exponential with the lower rate.

3.2 Test variables

The following variables were investigated

- The learning rate
- The size of the sample taken from the posterior when set independently of the batch size
- The batch size when using mini-batch processing (NB this cannot exceed the number of time points)
- The prior distribution of the parameter
- The initial posterior distribution of the parameters
- The use of the numerical (sample-based) calculation of the KL divergence on the posterior vs the analytic solution (possible in this case only because both prior and posterior are represented by a multivariate normal distribution).
- Whether covariance between parameters is modelled. The output posterior distribution can either be modelled as a full multivariate Gaussian with covariance matrix, or we can constrain the covariance matrix to be diagonal so there is no correlation between parameter values.

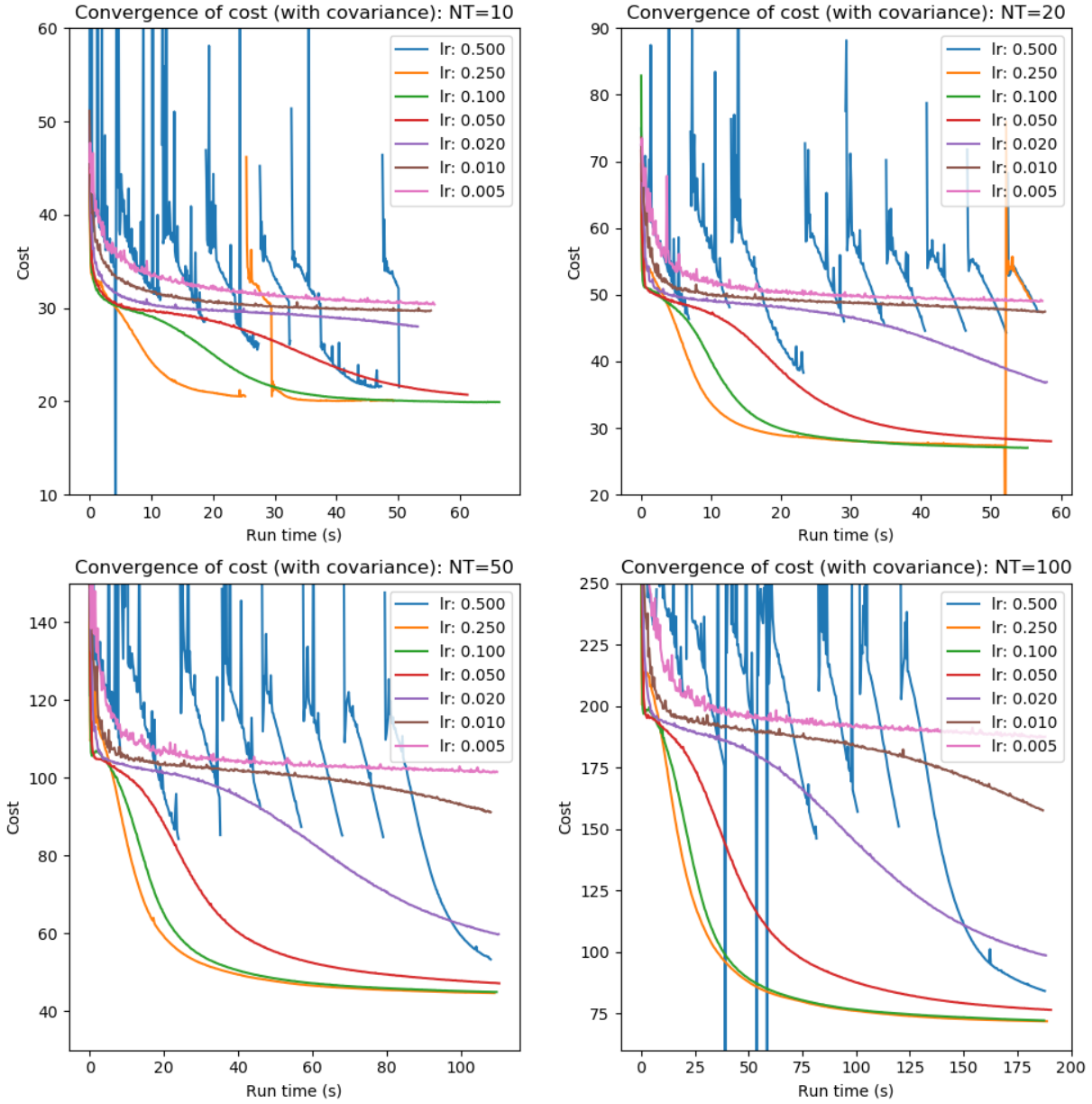
We investigate convergence by calculating the mean of the cost function across all test instances by epoch. Note that this measure is not directly comparable when different priors are used as the closeness of the posterior to the prior is part of the cost calculation. Convergence is plotted by runtime, rather than number of epochs for two reasons: Firstly since this is the measure of most interest to the end user, and also because in the case of mini-batch processing one epoch may represent multiple iterations of the optimization loop.

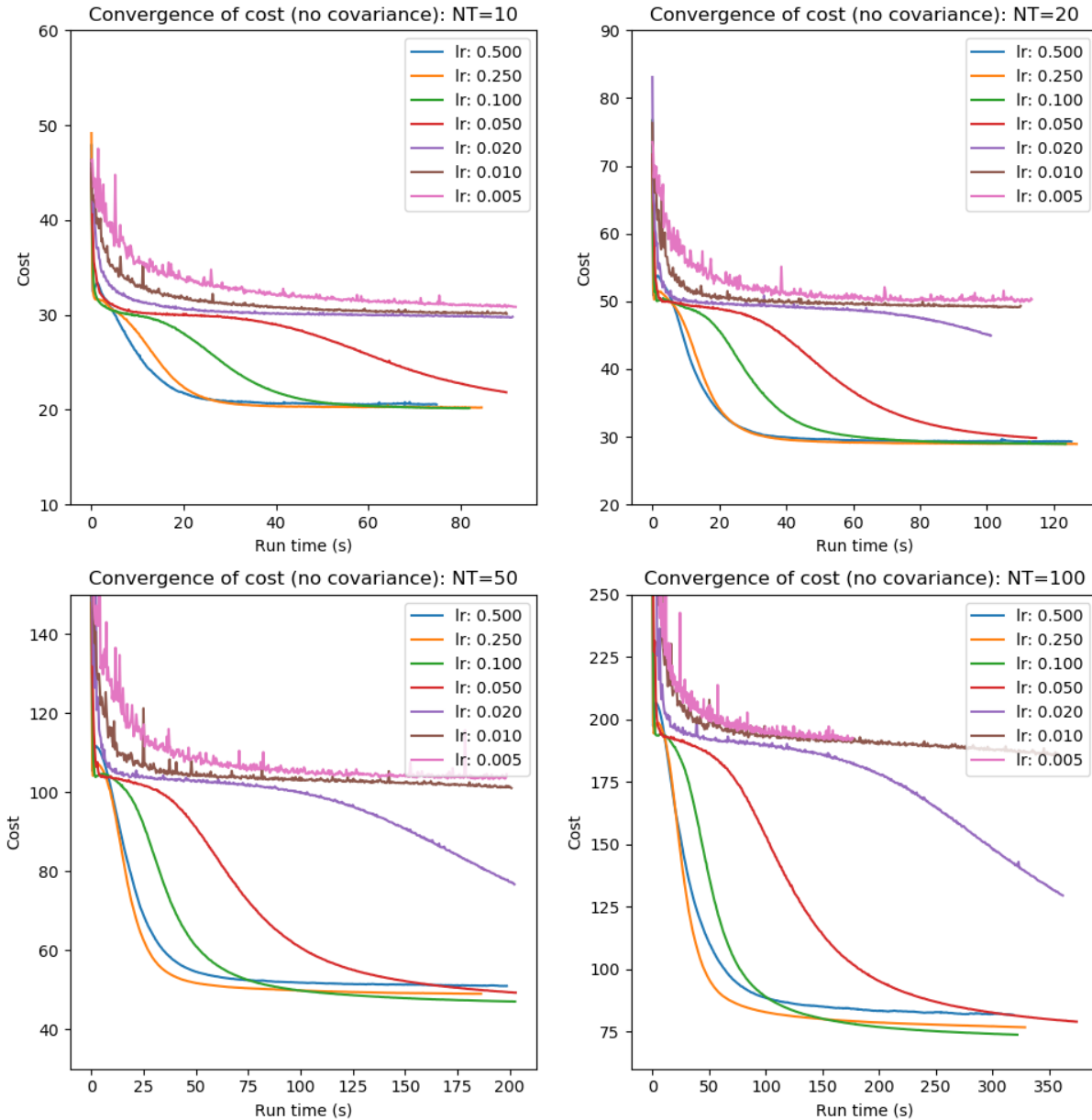
We also consider per-voxel speed of convergence, defined for each voxel as the epoch at which it first came within 5% of its best cost. This definition is only useful when convergence was eventually achieved.

3.3 Effect of learning rate

The learning rate determines the size of optimization steps made by the gradient optimizer and can be a difficult variable to select. Too high and the optimizer may repeatedly overshoot the minima and never actually converge, too low and convergence may simply be too slow. In many machine learning problems the learning rate is determined by trial and error however in our case we do not have this luxury as we need to be able to converge the model fitting on any unseen data without user intervention.

The convergence of the mean cost is shown below by learning rate and number of time points. In these tests mini-batch processing was not used, the analytic calculation of the KL divergence was used and the posterior sample size was 200.



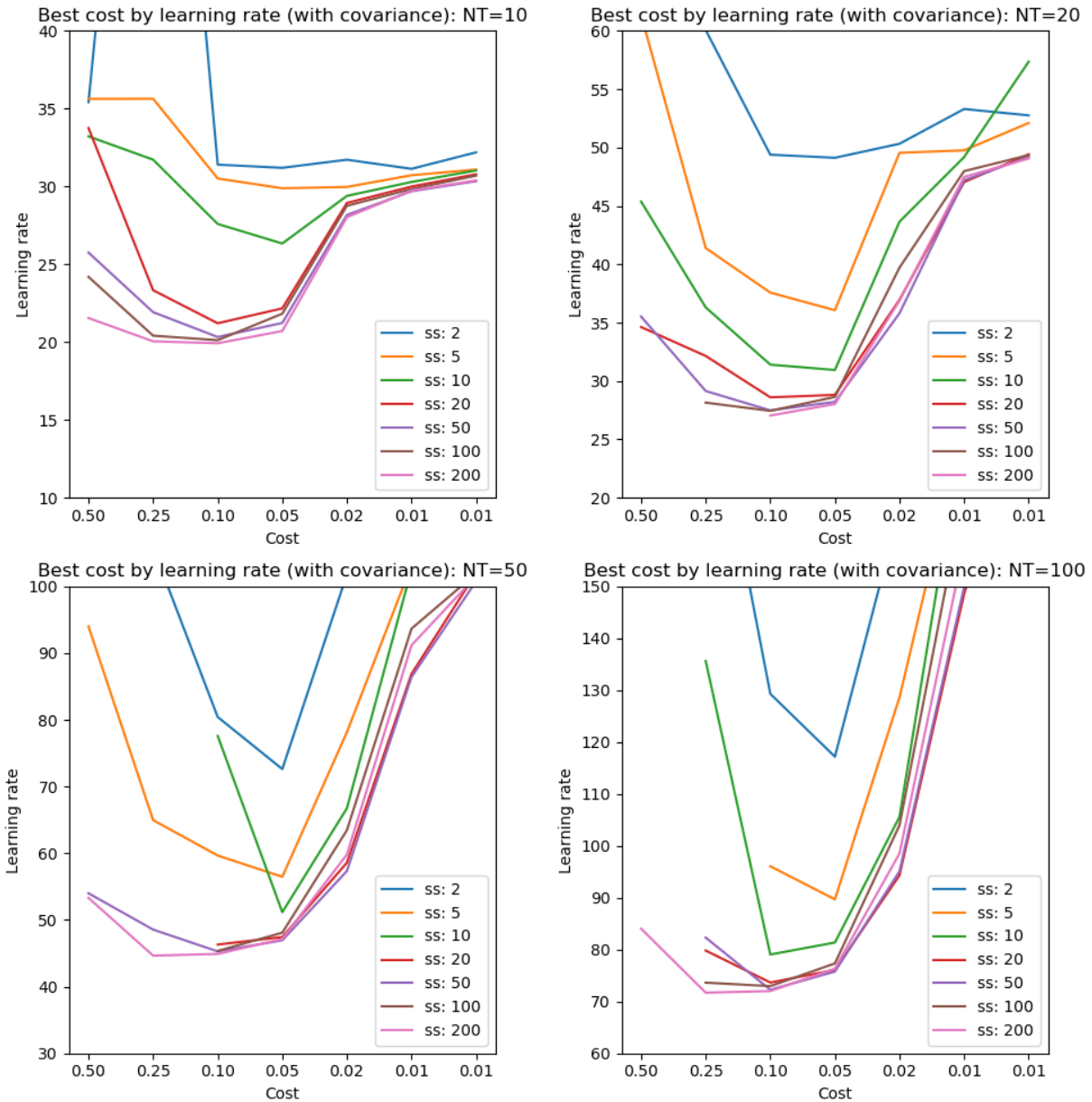


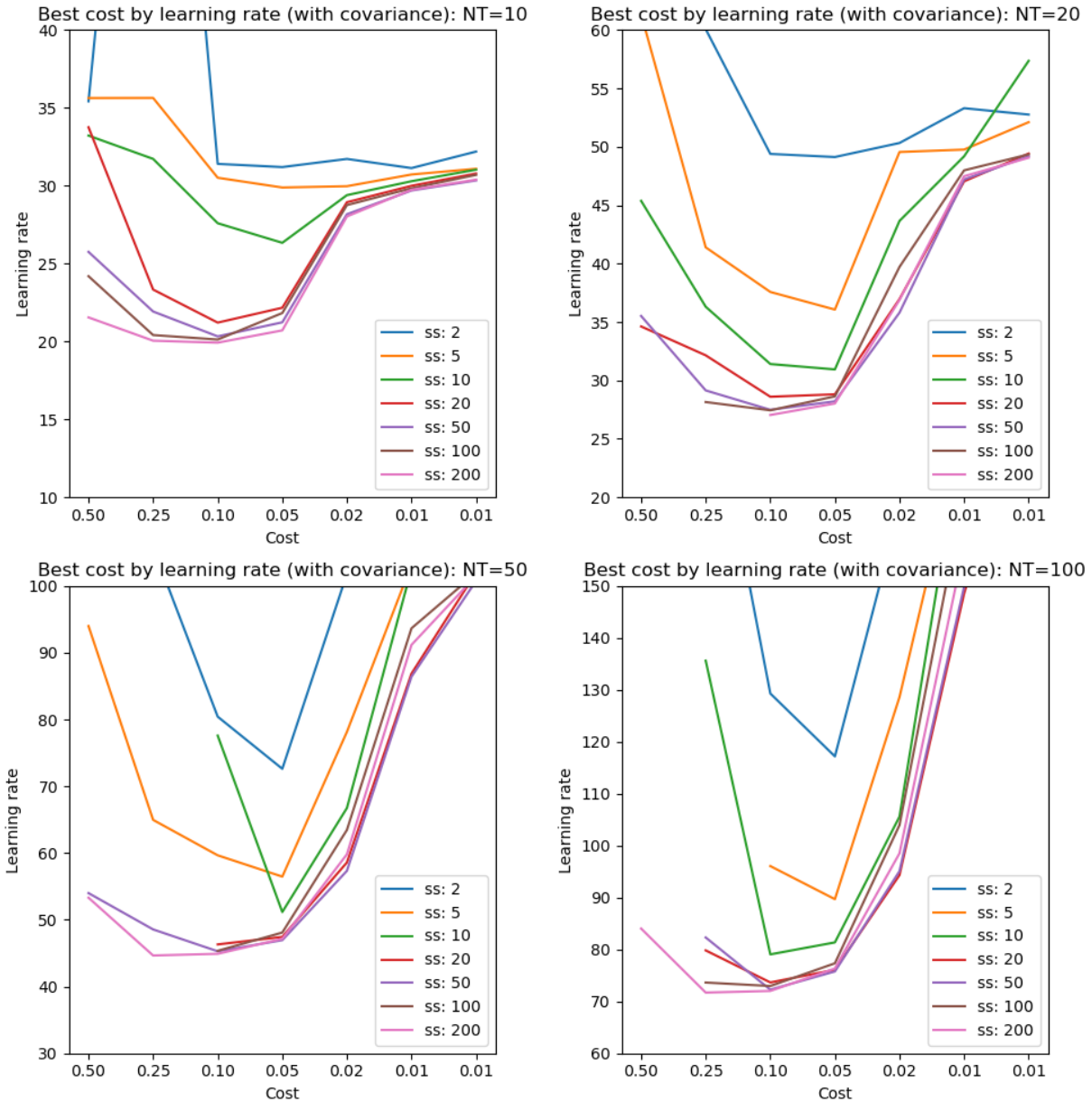
Although the picture is rather messy some observations can be made:

- Excessively high learning rates are unstable and do not achieve the best cost across the data sets (a learning rate of 1.0 was also tested but not plotted as the instability made the plots difficult to read).
- Very low learning rates (0.02 or lower) converge too slowly to be useful
- Even some learning rates which appear to show good smooth convergence do not achieve the minimum cost (e.g. LR=0.25, the amber line on some plots)
- Convergence with covariance is much more challenging as would be expected since the total number of fitted parameters rises from 10 to 20 per instance. In this high-dimensional space finding the overall cost minimum is likely to be more difficult.
- A learning rate of 0.1 gives the fastest reliable convergence. We will use this learning rate in subsequent tests where a single learning rate is required.

- Nevertheless initial convergence can be faster at a higher learning rate (0.25 or 0.5) suggesting use of ‘quenching’ where the learning rate is decreased during the optimization.

We can also examine the best cost achieved at various learning rates including variation in the posterior sample size:





These plots reinforce that a learning rate of 0.1 seems optimal for attaining best cost across a range of tests although there may be slight benefit to a higher rate when including covariance.

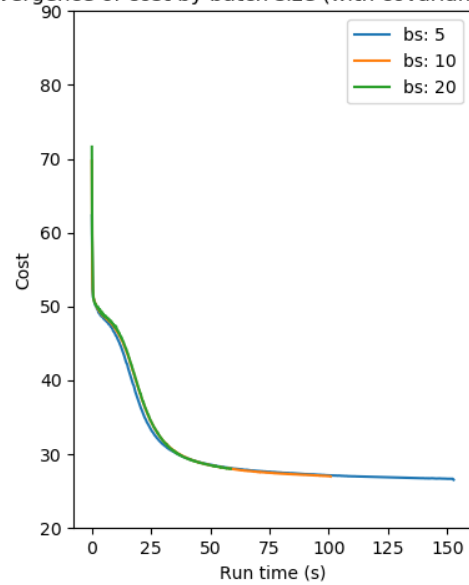
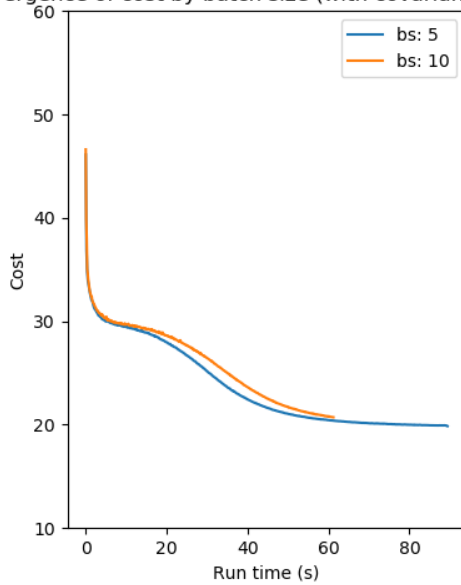
Increasing the posterior sample size leads to a gradual lowering of the best cost with little improvement beyond a size of 50. Small sample sizes combined with high learning rates are problematic - at low learning rates the sample size matters less. We will consider the sample size in more detail in a later section.

3.4 Effect of batch size and learning rate on best cost achieved

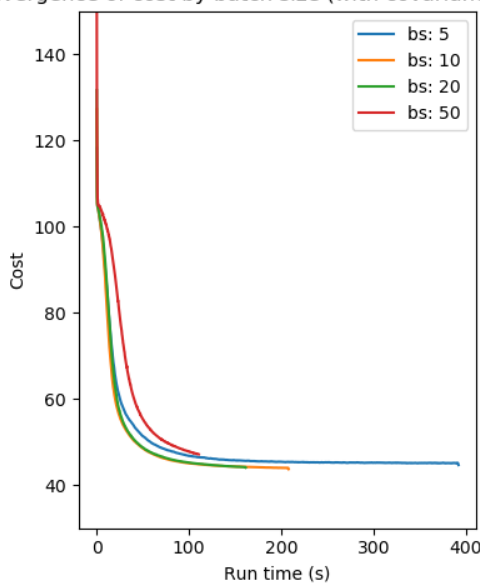
Optimization of the cost function proceeds by 'epochs' which consists of a single pass through all of the data. Batch processing consists of dividing the data into smaller batches and performing multiple iterations of the optimization - one for each batch - during an epoch. Processing the data in batch is a commonly used method to accelerate convergence and works because updates to the parameters occurs multiple times during each epoch. The optimization

steps are ‘noisier’ because they are based on less training samples and this helps to avoid converging onto local minima. Of course if the batch size is too small the optimization may become so noisy that convergence does not occur at all.

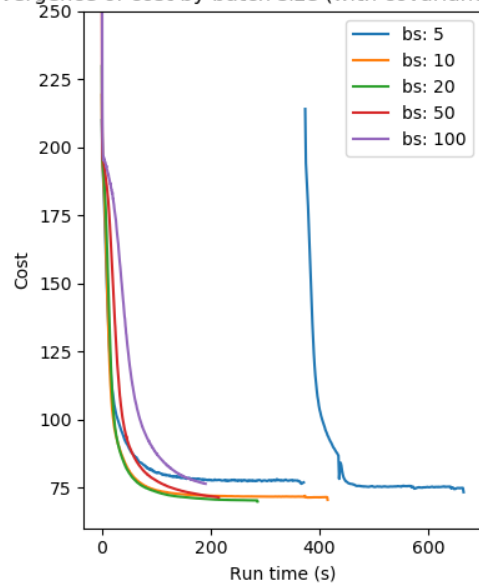
Convergence of cost by batch size (with covariance): NT=10



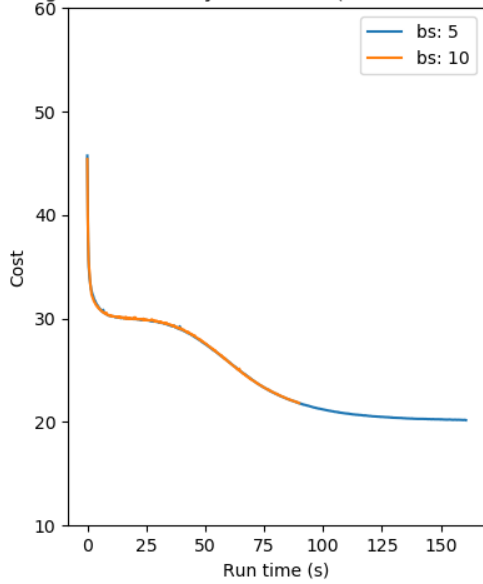
Convergence of cost by batch size (with covariance): NT=50



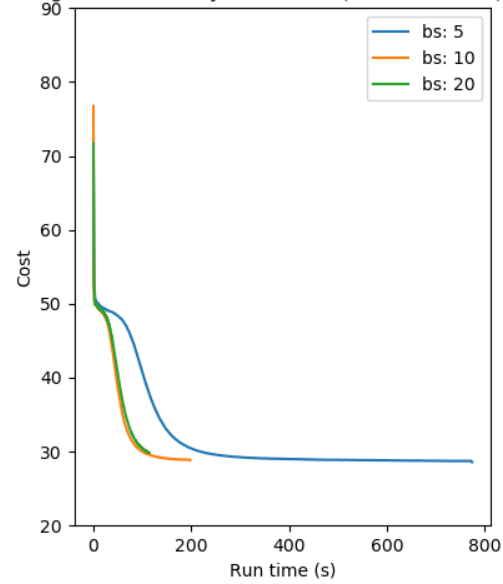
Convergence of cost by batch size (with covariance): NT=100



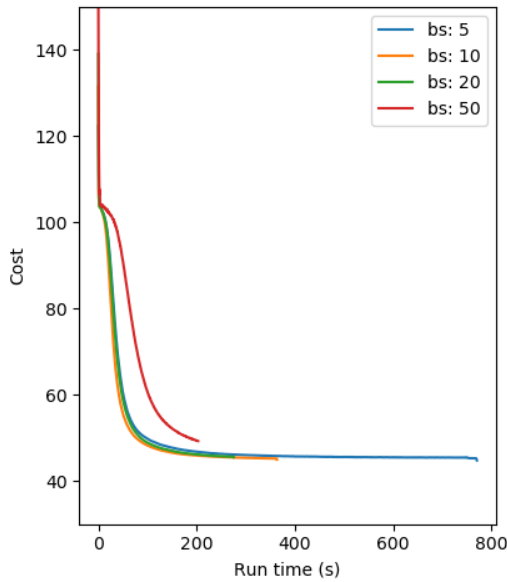
Convergence of cost by batch size (no covariance): NT=10



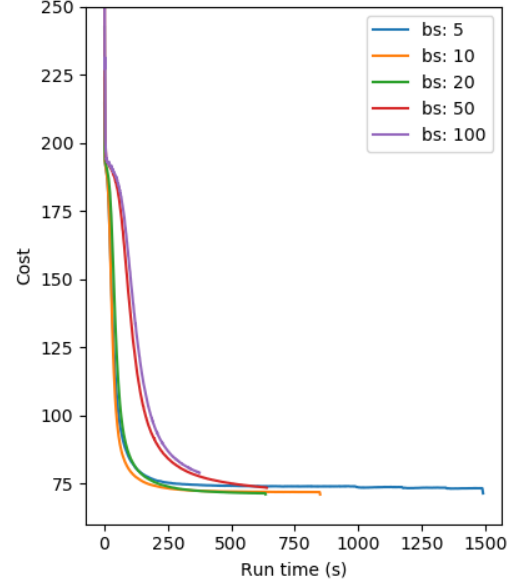
Convergence of cost by batch size (no covariance): NT=20



Convergence of cost by batch size (no covariance): NT=50

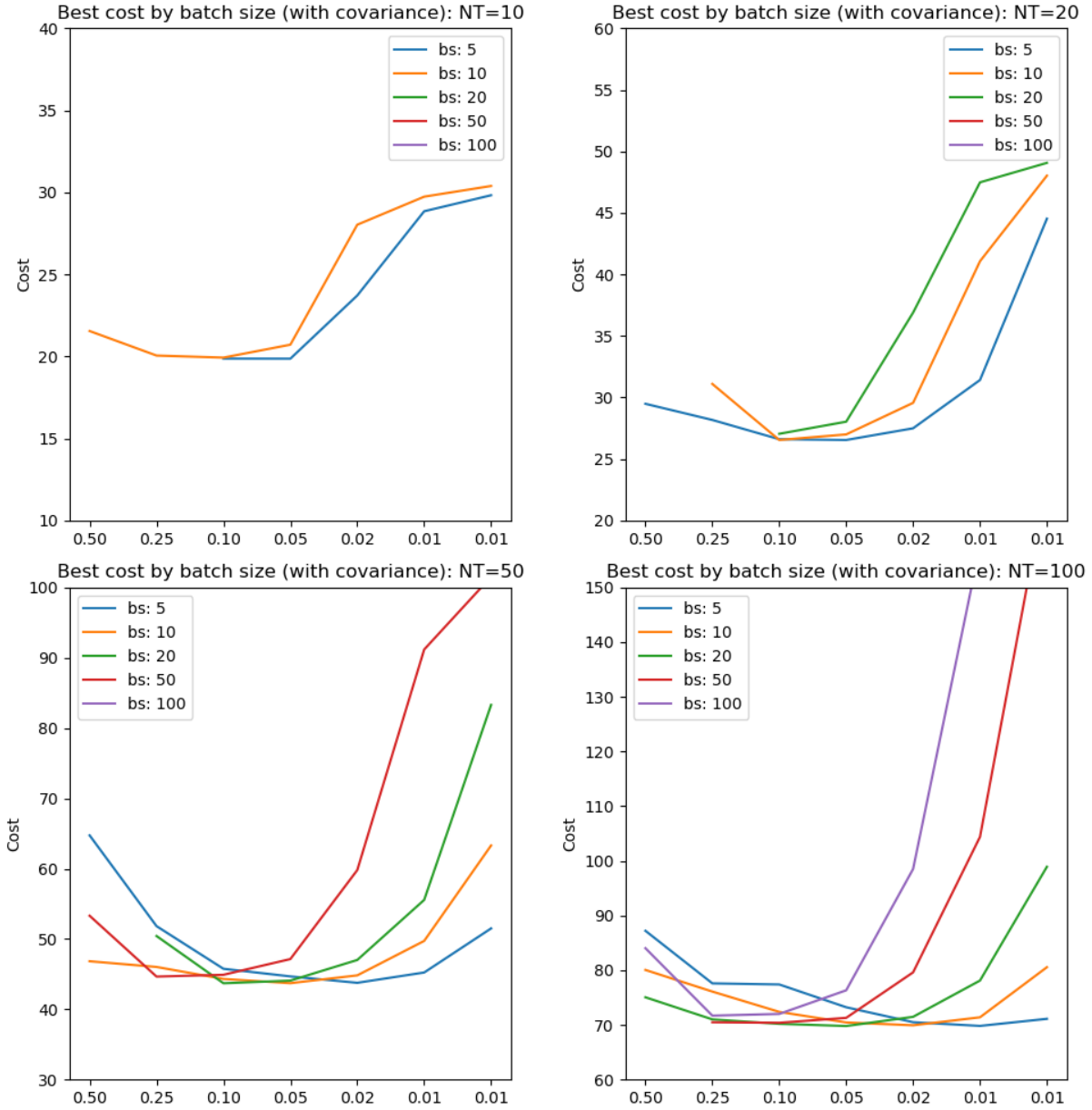


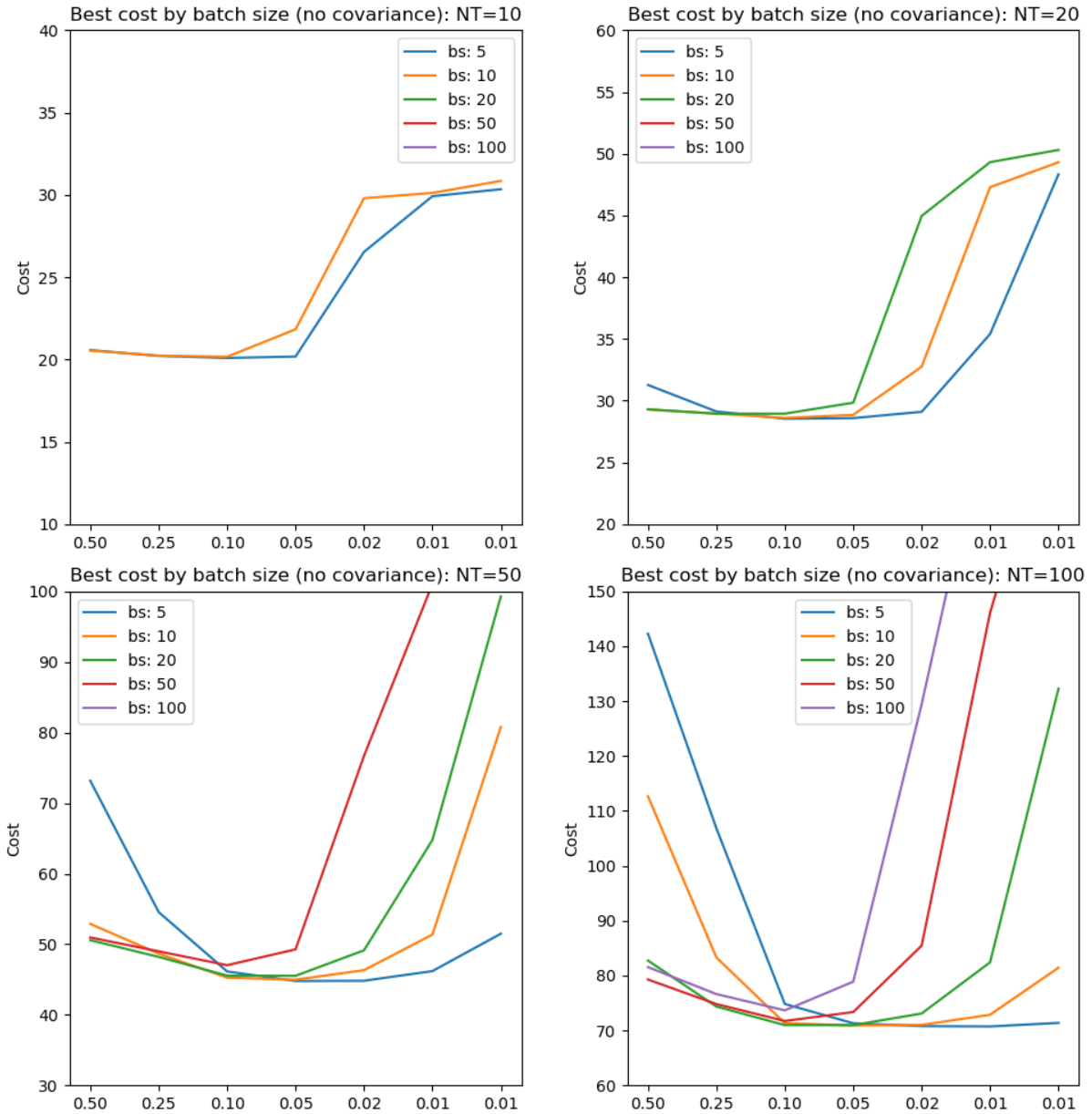
Convergence of cost by batch size (no covariance): NT=100



These plots show that mini-batch processing does indeed accelerate convergence especially where the number of data points is high. Batch sizes of 10 and 20 produce consistently fast convergence compared to using the entire data set at each epoch.

Since mini-batch processing increases gradient noise we might expect it to interact with the learning rate which we can investigate by looking at the best cost achieved by learning rate at different batch sizes:





These results confirm the use of learning rates between 0.1 and 0.05 as optimal across batch sizes. In general small batch sizes can be used with lower learning rates. Large batch sizes can reach a lower cost at higher learning rates, although sometimes they are not able to converge at all. This is in line with expectations since high learning rates and low batch sizes both imply a ‘noisier’ optimization and both excessively high or low noise in the optimization can be problematic.

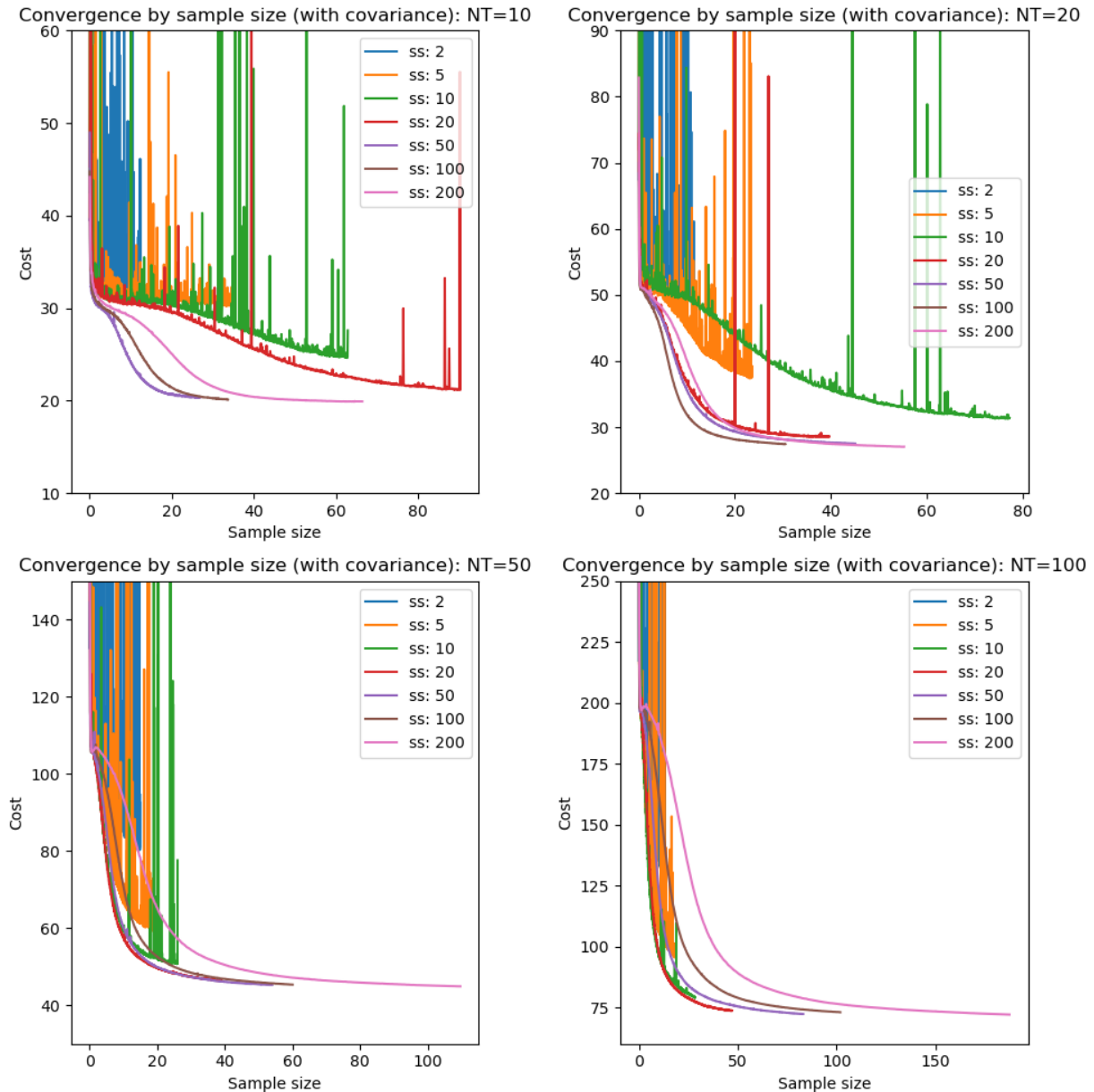
It is noticeable that batch sizes smaller than the number of points in the data only give faster convergence for larger numbers of time points (50 or 100). However there is still an advantage to mini-batch processing in that the best cost curves are ‘flatter’, i.e. more tolerant of variation in the learning rate.

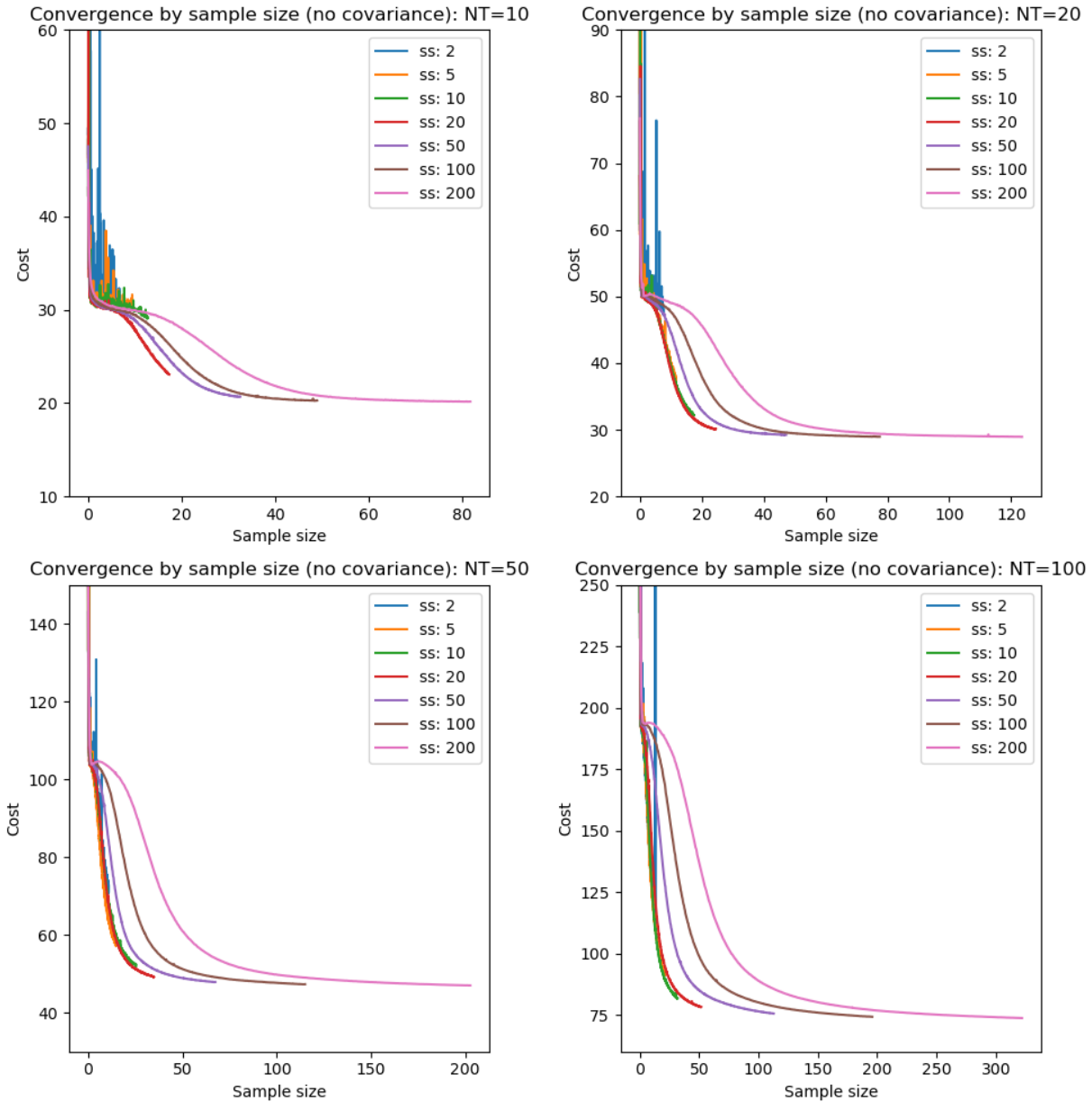
Where batch size is fixed in subsequent tests we use a value of 10.

3.5 Effect of posterior sample size

The sample size is used to estimate the integrals in the calculation of the cost function, so we would expect that a certain minimum size would be required for a good result. The smaller the sample, the more the resulting cost gradients are affected by the random sample selection which may lead to a noisier optimisation process that may not converge at all. On the other hand, larger sample sizes will take longer to calculate the mean cost giving potentially slower real-time convergence.

Here we vary the sample size with a fixed learning rate of 0.1 and initially without mini-batch processing:

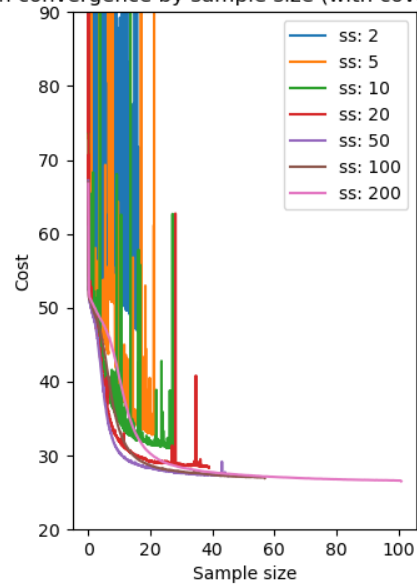
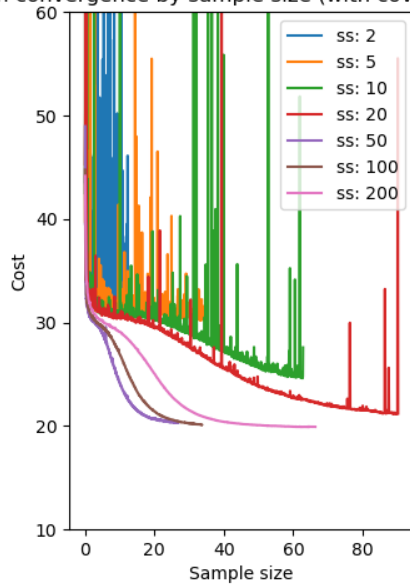




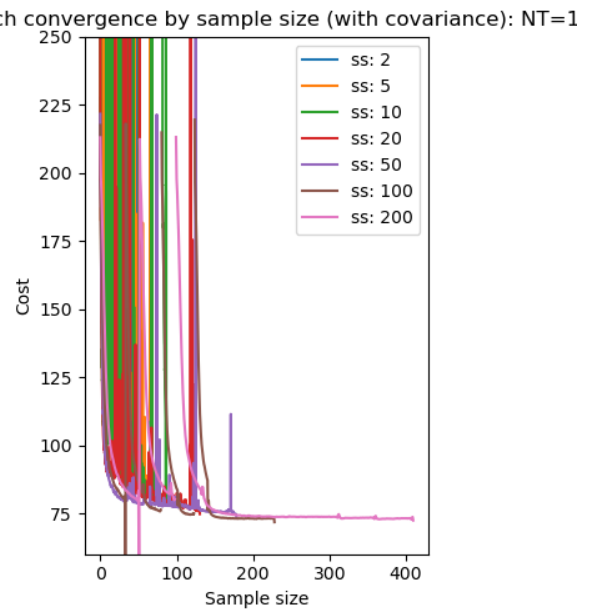
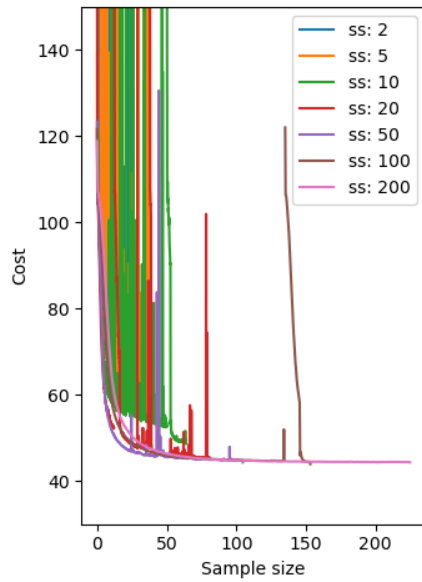
This illustrates that very small sample sizes do indeed result in a noisy potentially non-convergent optimization, and also that larger sample sizes can produce overall slower convergence. The picture is mixed, however the optimal sample size is around 50 when inferring covariance but only 20 without covariance.

We can also look at the equivalent convergence when using mini-batch processing with a batch size of 10:

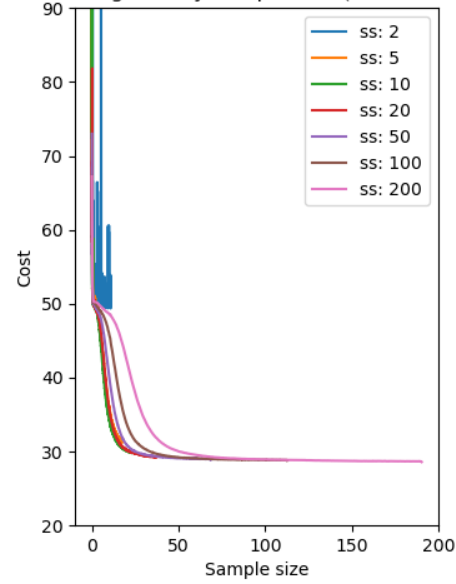
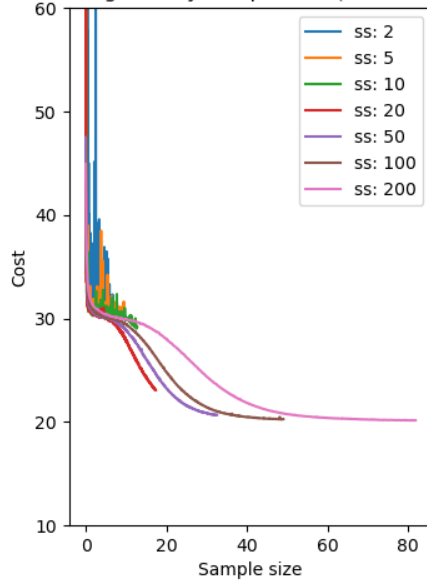
ni-batch convergence by sample size (with covariance): NT=10



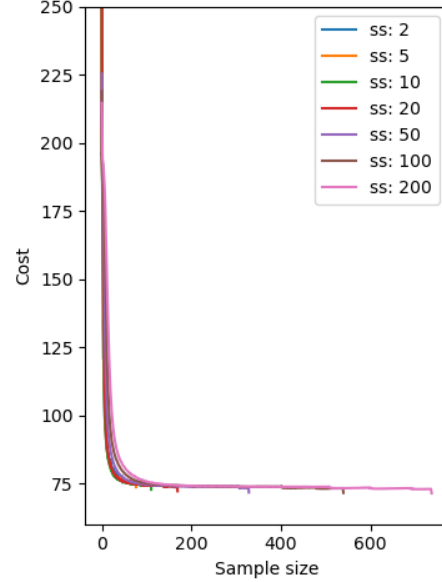
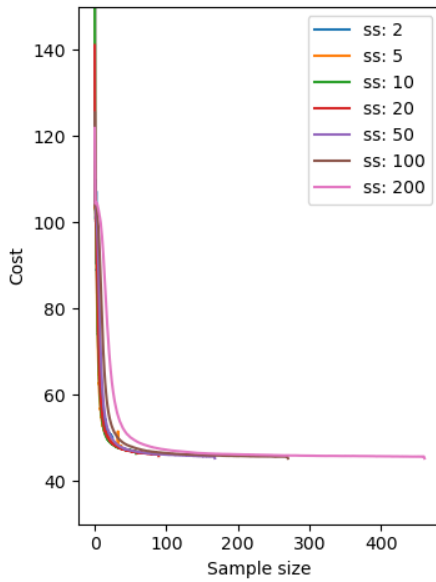
ni-batch convergence by sample size (with covariance): NT=50



ini-batch convergence by sample size (no covariance): NT=10 Mini-batch convergence by sample size (no covariance): NT=2



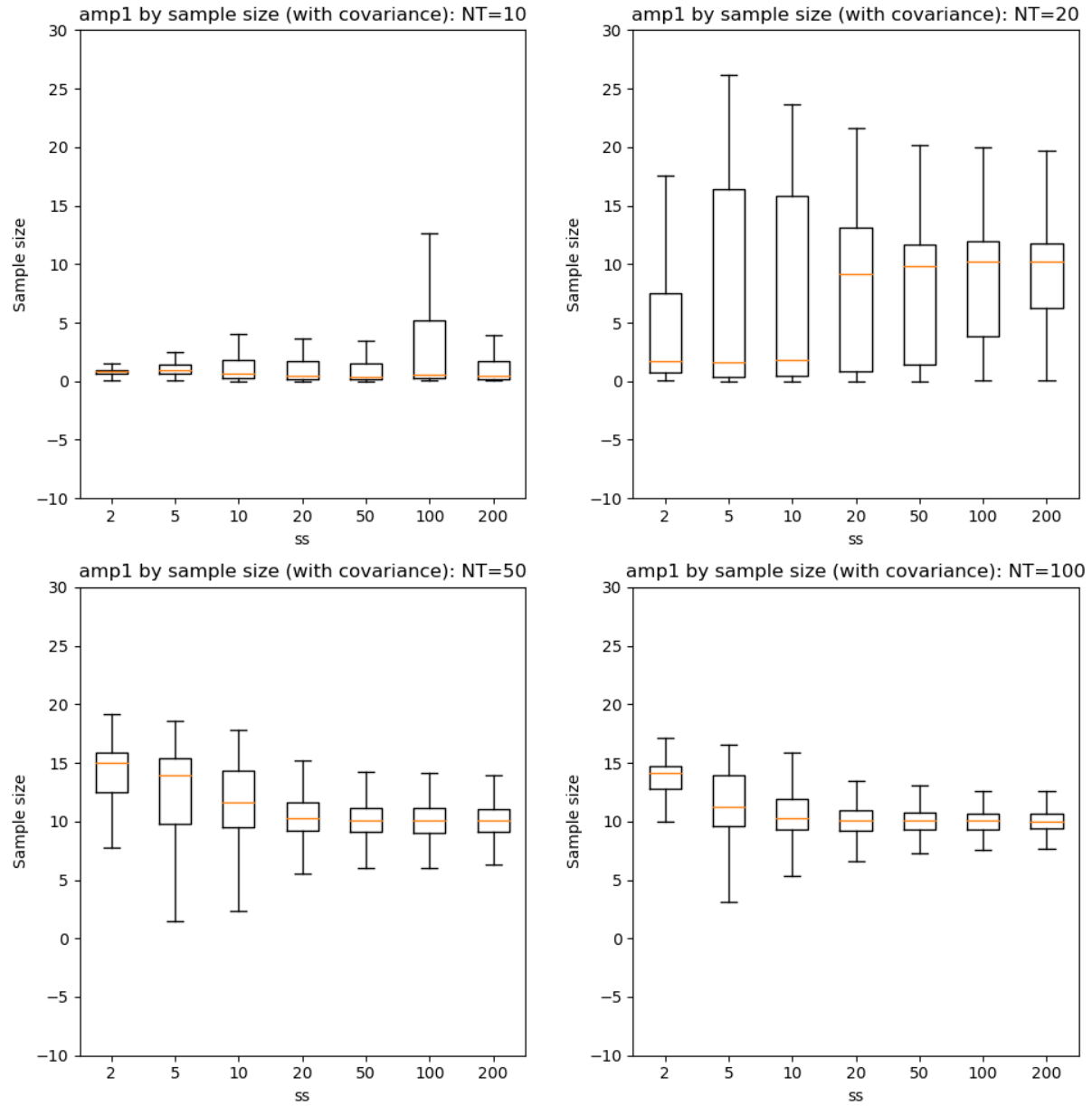
ini-batch convergence by sample size (no covariance): NT=50 Mini-batch convergence by sample size (no covariance): NT=100

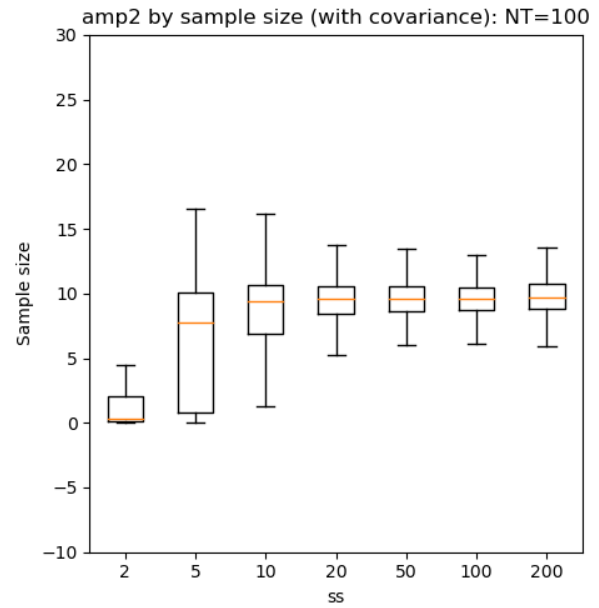
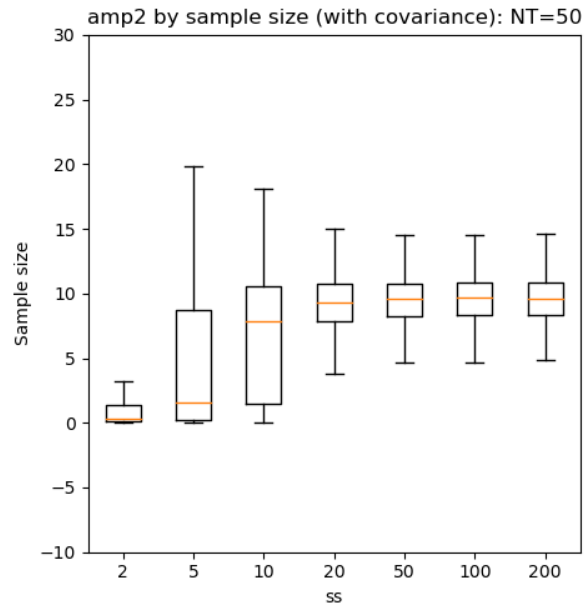
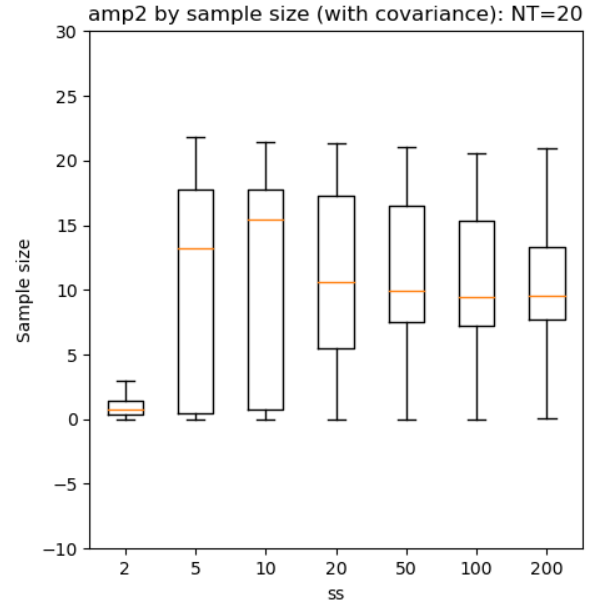
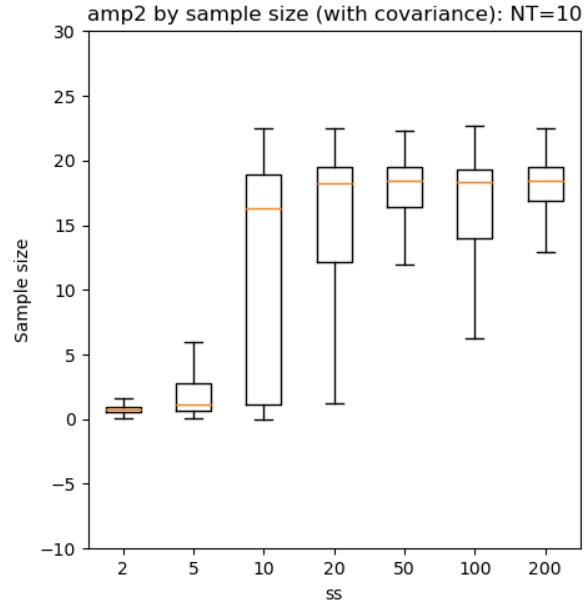


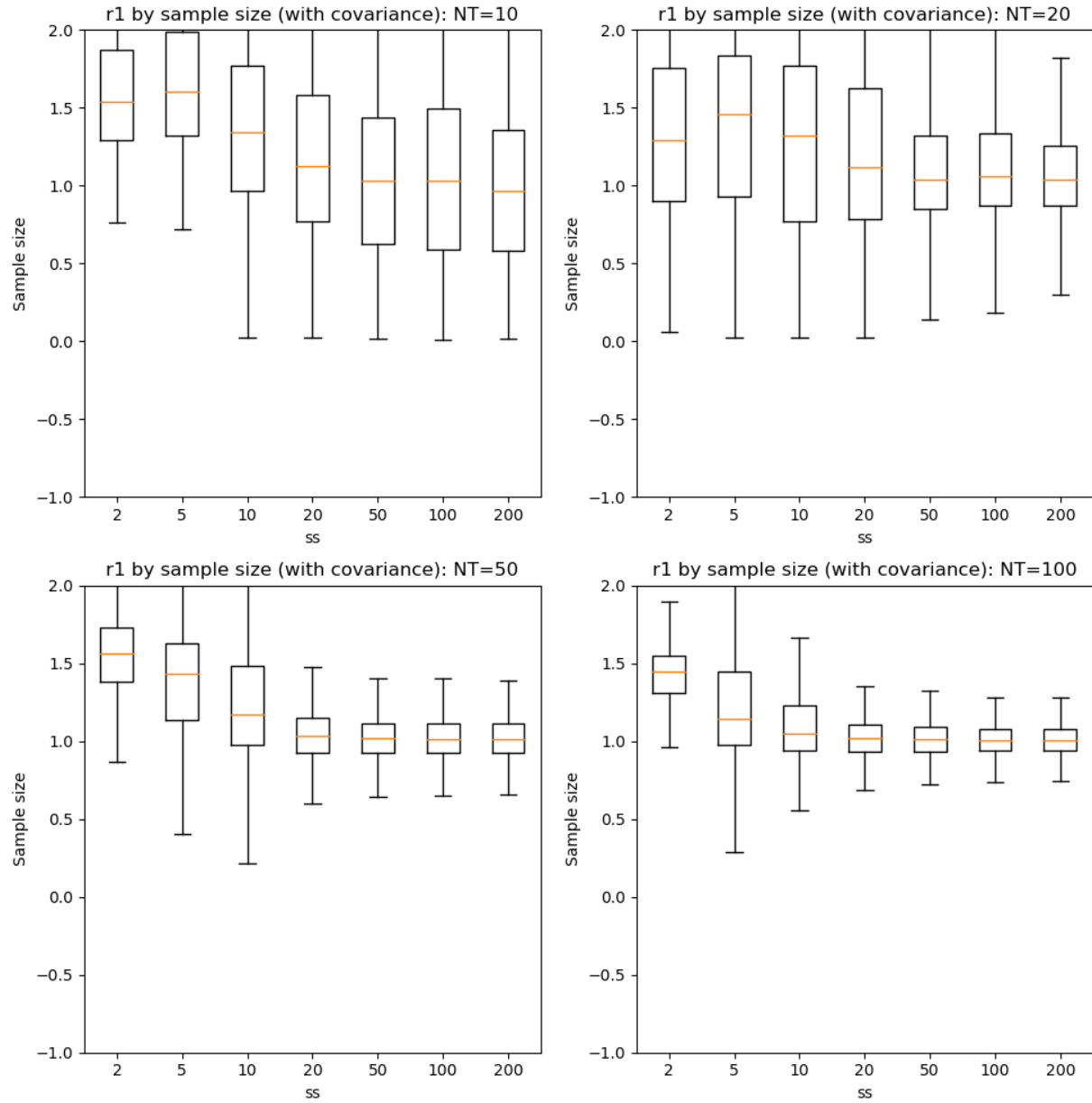
The results are essentially the same however the optimization becomes extremely unstable at small sample sizes when combined with mini-batch processing.

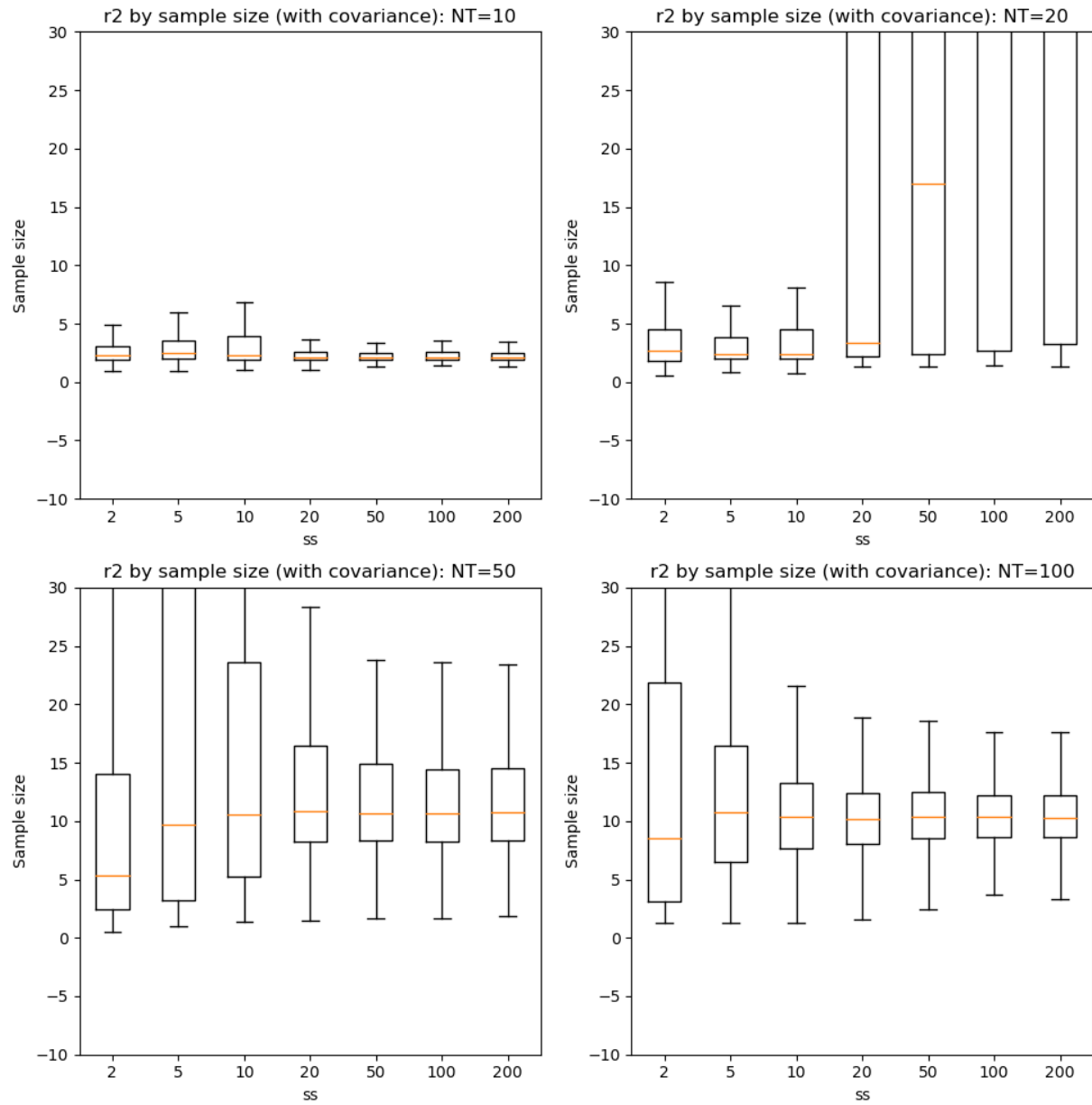
Note also that it is possible that a lower sample size may constrain the free energy systematically (analogously to the way in which numerical integration techniques may systematically under or over estimate depending on whether the function is convex). So the higher free energy of smaller sample sizes does not necessarily mean that the posterior is actually further from the best variational solution.

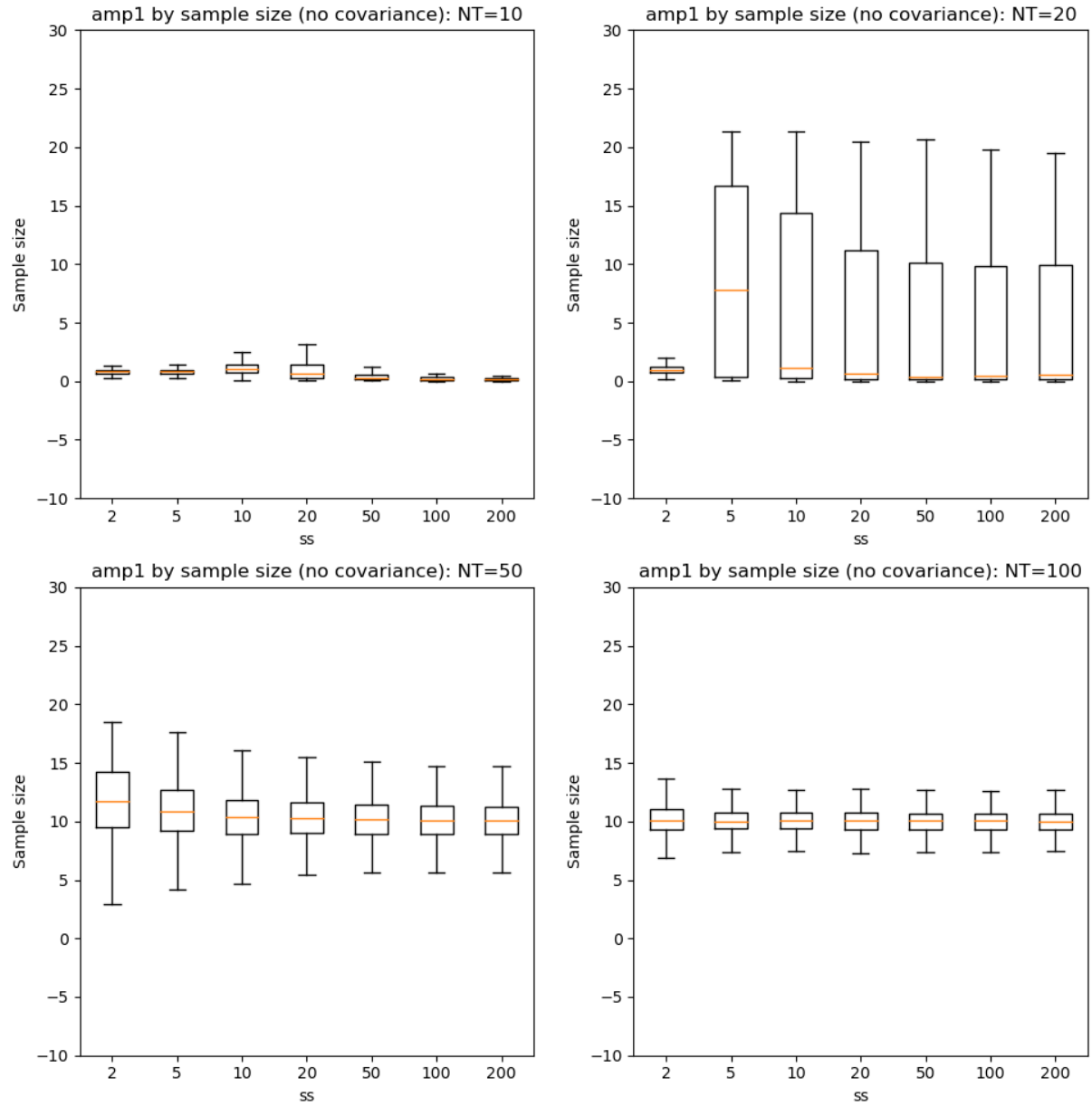
With this in mind it is useful to look at convergence in parameter values (using mini-batch processing as above):

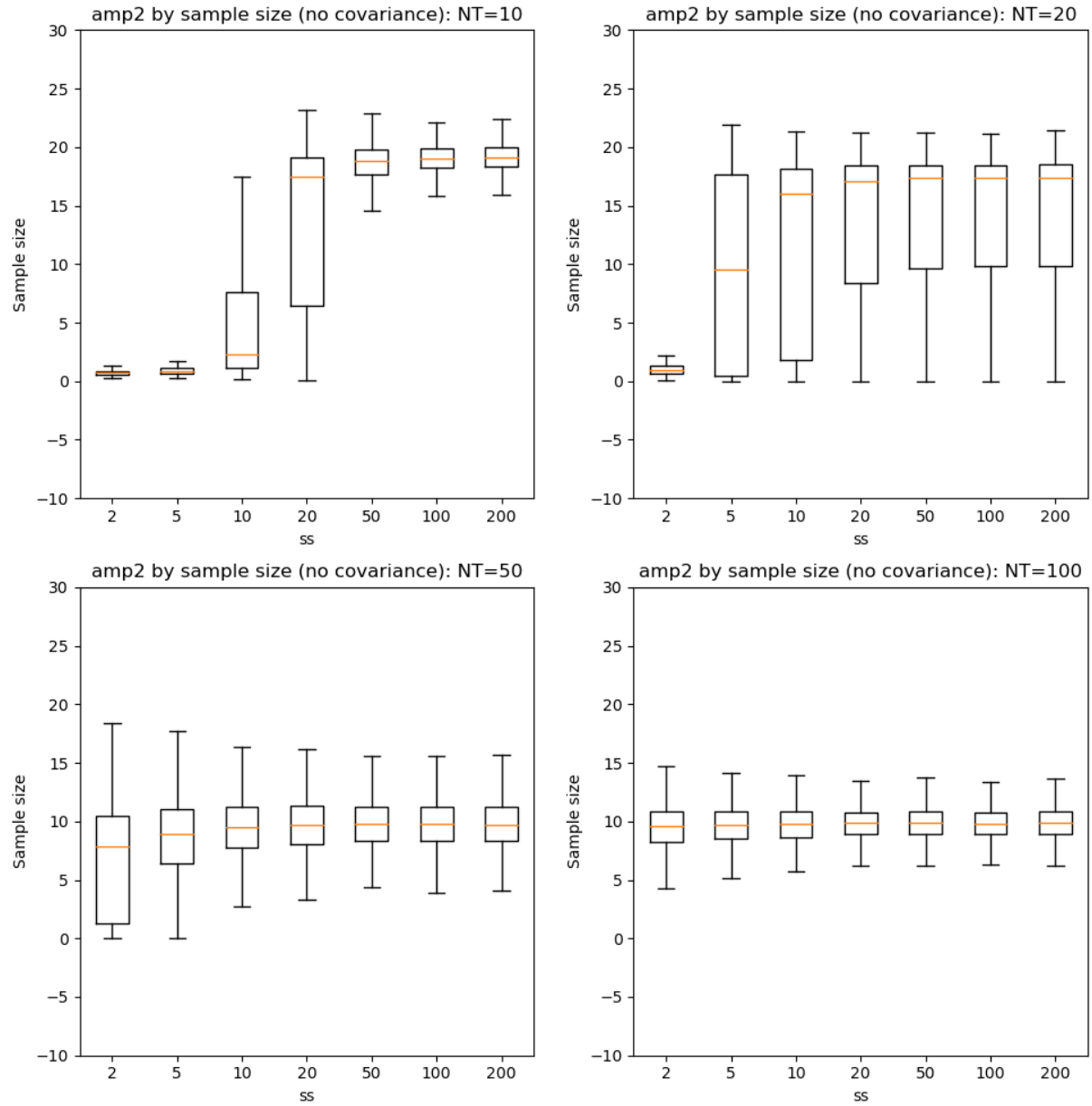


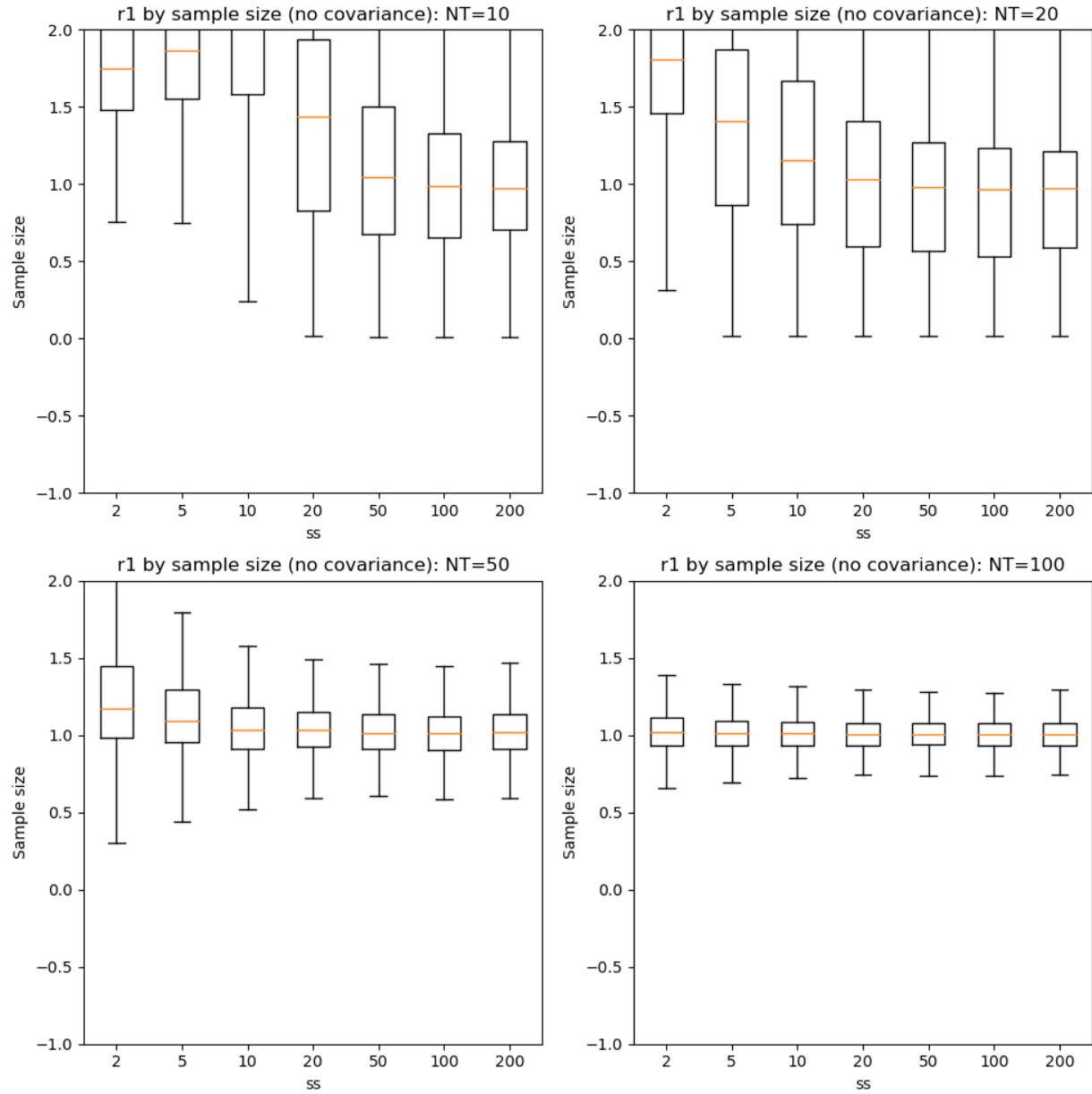


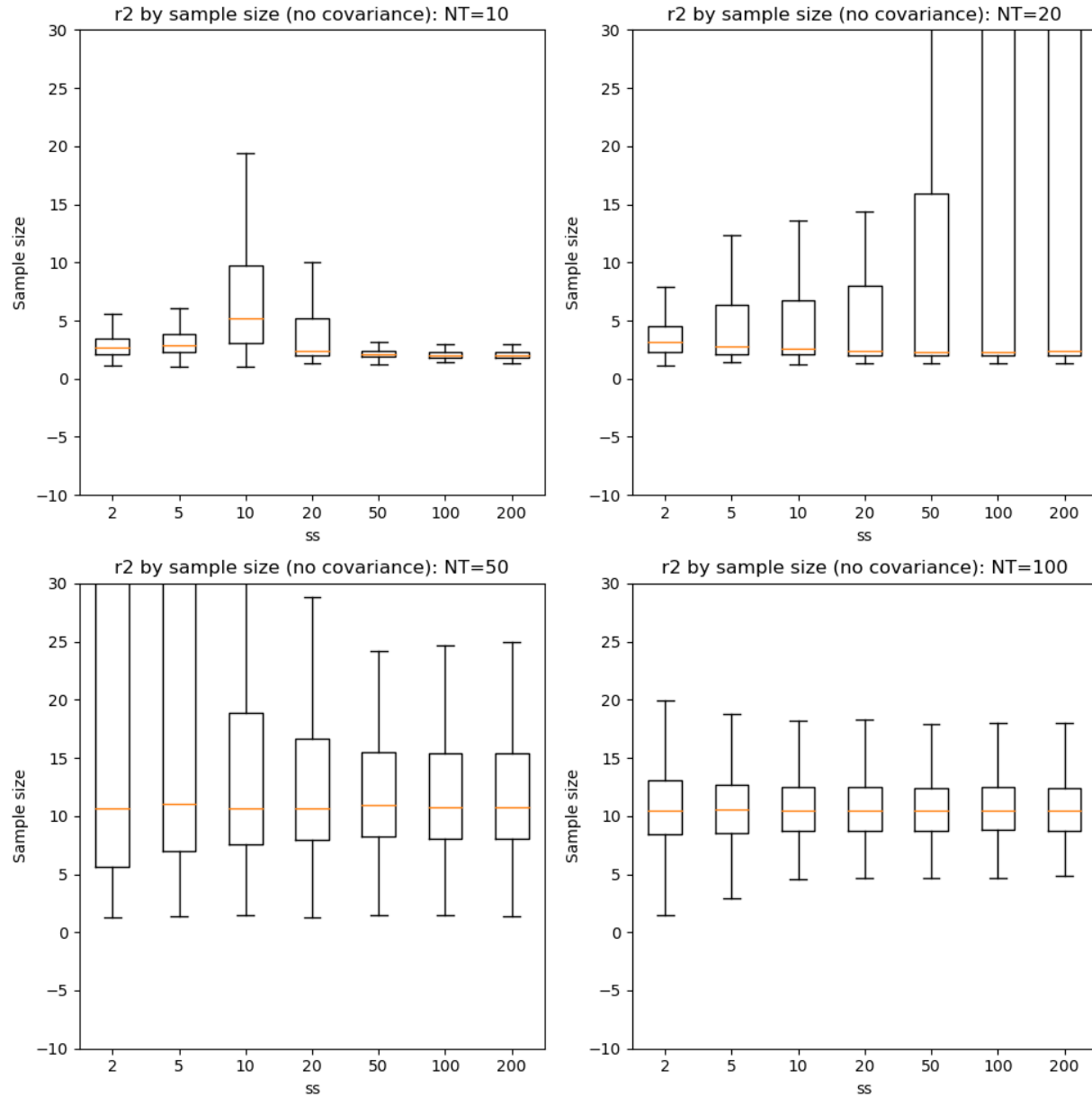












Here we can see that firstly, with fewer data points the optimization tends to favour a single-exponential solution and does not recover the biexponential property for most voxels until we have at NT=50.

In general there is little benefit to sample sizes above 50, and 20 gives very similar results for NT=50 and NT=100.

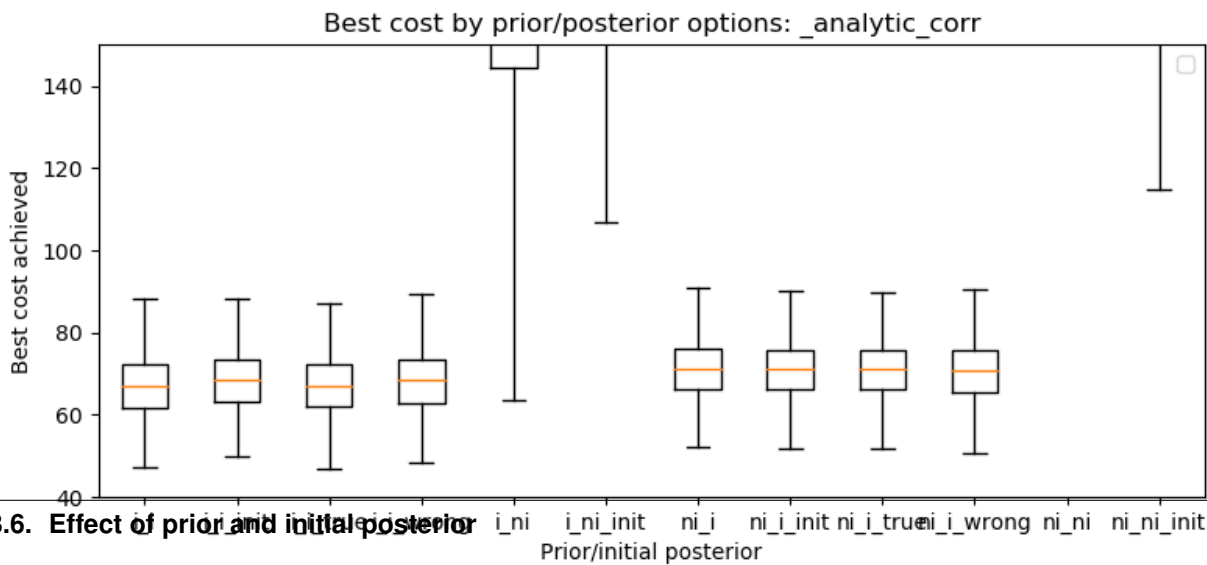
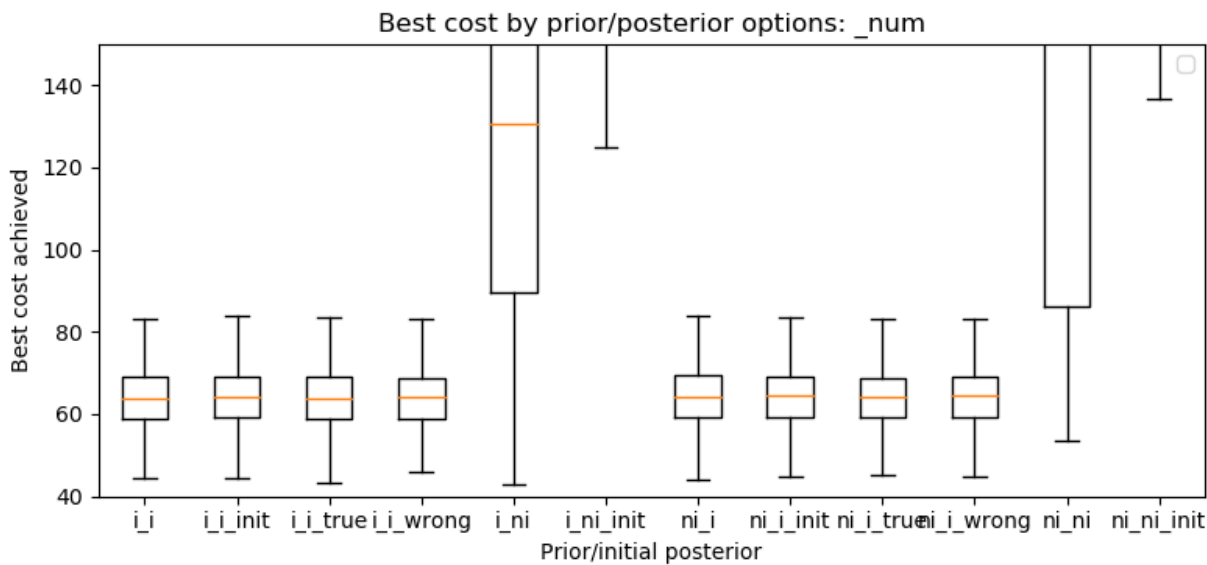
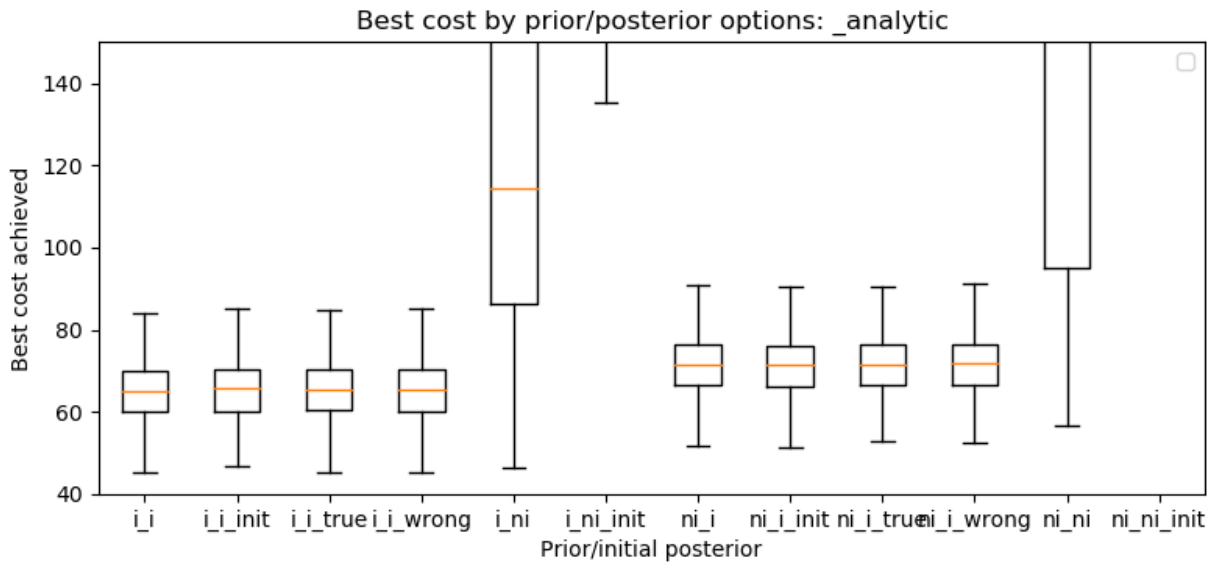
3.6 Effect of prior and initial posterior

The following combinations of prior and posterior were used. An informative prior was set with a mean equal to the true parameter value and a standard deviation of 2.0. Non-informative priors were set with a mean of 1 and a standard deviation of $1e6$ for all parameters.

Non-informative initial posteriors were set equal to the non-informative prior. Informative posteriors were set with a standard deviation of 2.0 and a mean which either matched or did not match the true parameter value as described

below. In addition, an option in the model enabled the initial posterior mean for the amplitude parameters to be initialised from the data.

Code	Description
i_i	Informative prior, informative posterior initialised with mean values equal to 1.0 for all parameters
i_i_init	Informative prior, informative posterior initialised with true values of the decay rates and with amplitude initialised from the data
i_i_true	Informative prior, informative posterior initialised with true values
i_i_wrong	Informative prior, informative posterior initialised with mean values of 1.0 for the decay rate and 100.0 for the amplitudes (i.e. very far from the true values)
i_ni	Informative prior, non-informative posterior
i_ni_init	Informative prior, non-informative posterior with amplitude initialised from the data
ni_i	Non-informative prior, informative posterior initialised with mean values equal to 1.0 for all parameters
ni_i_init	Non-informative prior, informative posterior initialised with true values of the decay rates and with amplitude initialised from the data
ni_i_true	Non-informative prior, informative posterior initialised with true values
ni_i_wrong	Non-informative prior, informative posterior initialised with mean values of 1.0 for the decay rate and 100.0 for the amplitudes (i.e. very far from the true values)
ni_ni	Non-informative prior, non-informative posterior
ni_ni_init	Non-informative prior, non-informative posterior with amplitude initialised from the data



These results show that in terms of absolute convergence there is no significant difference between the choice of prior and posterior. Note that the absolute cost achieved can be different between the informative and non-informative priors as expected. The exception is the cases where a *non-informative* initial posterior is used - these cases do not achieve convergence.

The explanation for this lies in the fact that components of the cost are dependent on a sample drawn from the posterior. In the case of a non-informative posterior samples of realistic sizes cannot be large enough to be representative and different samples may contain widely varying contents. Such samples cannot reliably direct the optimisation to minimise the cost function because the calculated cost (and its gradients) are dominated by random variation in the values contained within the sample.

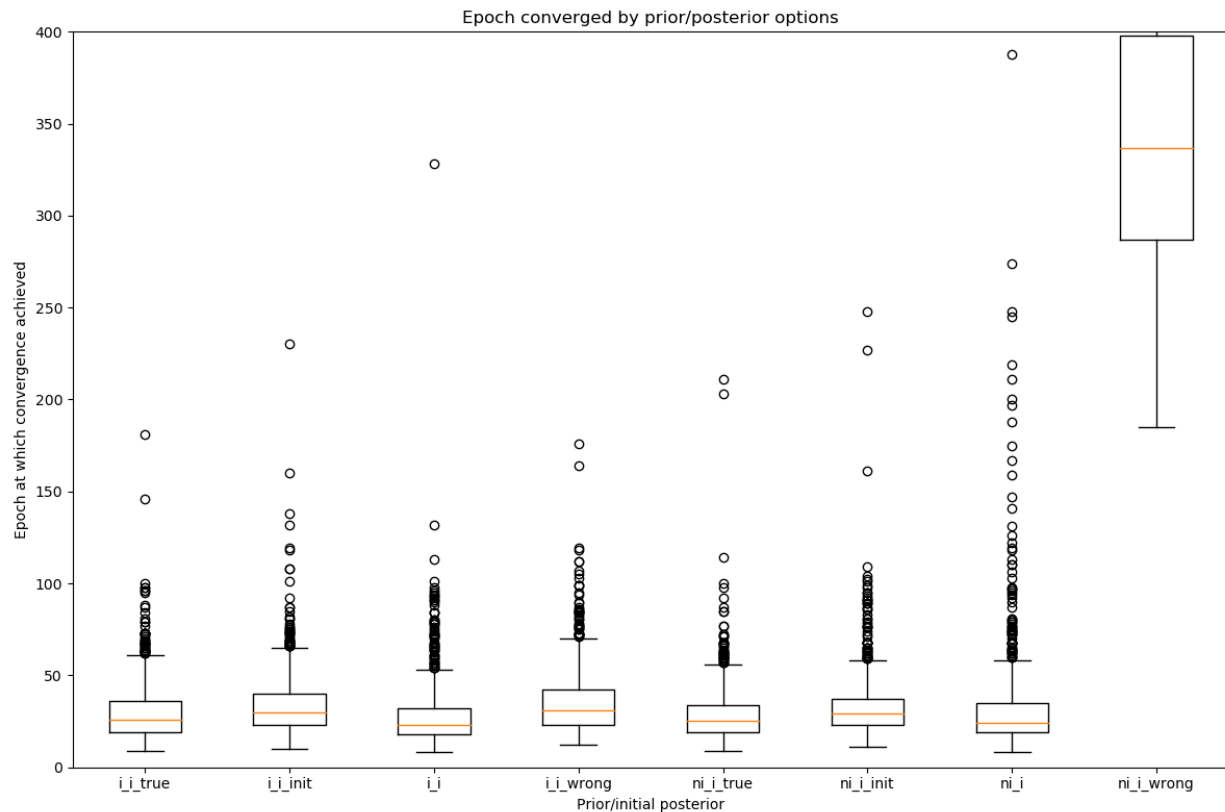
By contrast if the posterior is informative - even if it is far from the best solution - different moderately-size random samples are all likely to provide a reasonable representation of that distribution. The optimisation will therefore be directed to minimise the cost more reliably since it is less dependent on the particular values that happened to be included in the sample.

We conclude that the initial posterior must be informative even if it is a long way from the true solution.

The `_analytic` and `_num` plots are identical apart from using the analytic or the numerical solution to the KL divergence between two MVNs. The similarity between these results suggests that the numerical solution should be sufficient in cases where the prior and posterior cannot be represented as two MVN distributions.

The `_corr` and `__nocorr` plots were generated with and without a full posterior covariance matrix. In this case we see little difference between the two.

It is reassuring that the cost can converge under a wide variety of prior and posterior assumptions, however it is also useful to consider the effect of these variables on speed of convergence. The results below illustrate this:

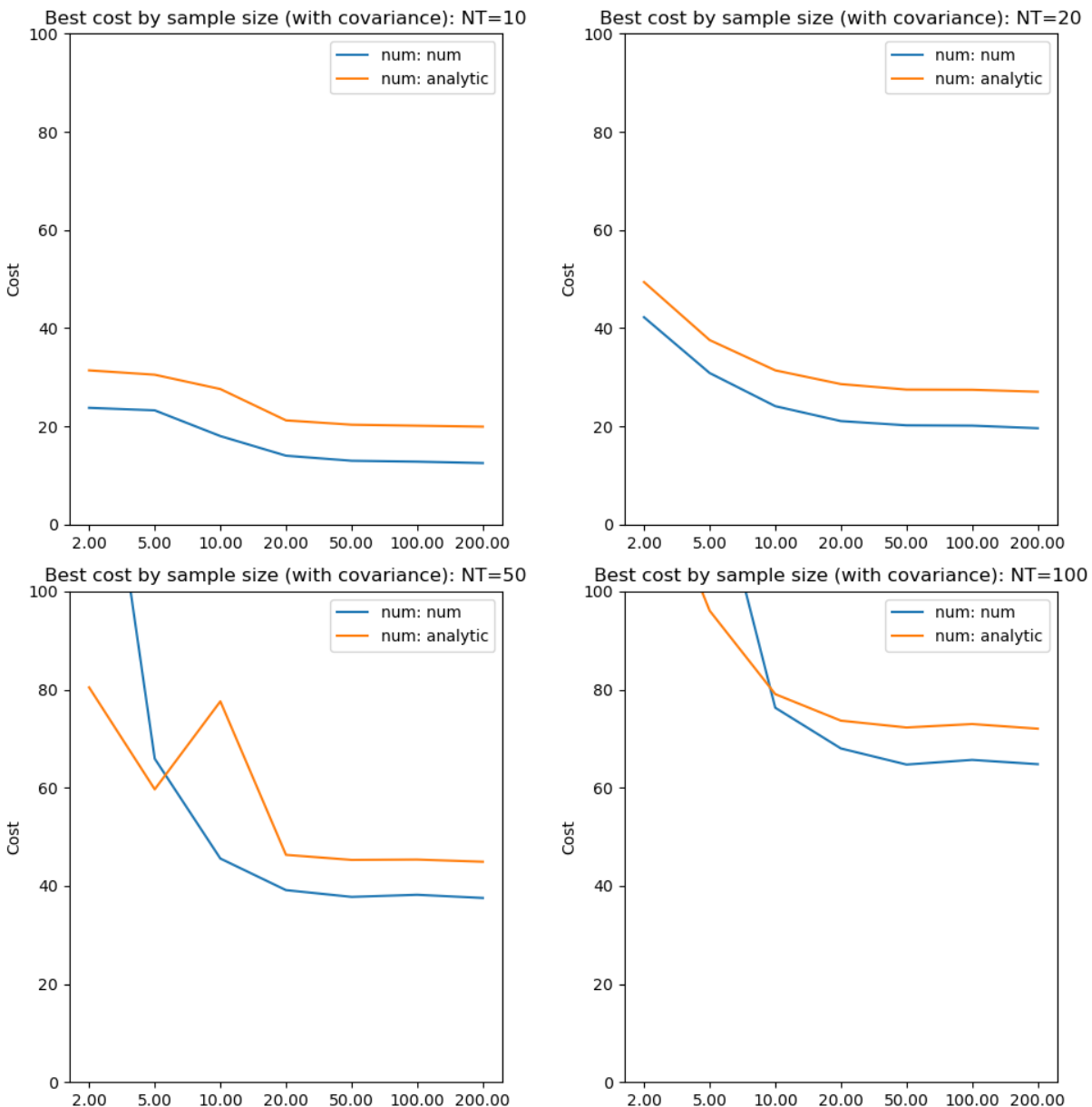


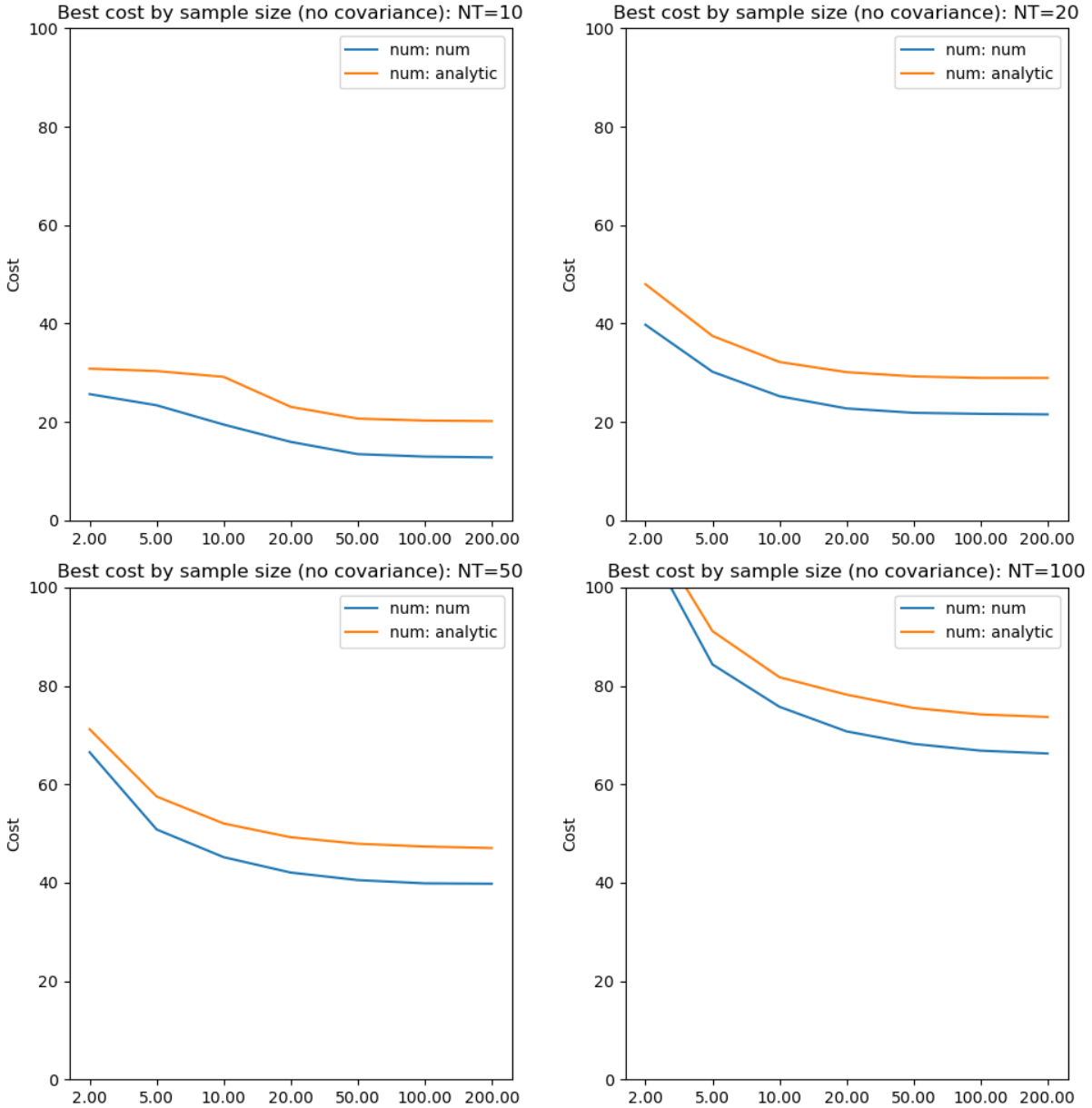
This plot shows the epoch at which each voxel converged (to within 5% of its final values). The box plot shows the median and IQR, while the circles show slow-converging outliers. For the reasons given above, non-informative posterior test cases were excluded from this plot.

It is clear that the main impact on convergence speed is the initial posterior. Where it is far from the true values (`i_wrong`) convergence is slowest. However this problem is much less obvious when the priors are informative as in this case the ‘wrong’ posterior values generate high latent cost as they are far from the ‘true’ prior values. This quickly guides the optimisation to the correct solution. Initialisation of the posterior from the data (where there is a reasonable method for doing this) is therefore recommended to improve convergence speed.

3.7 Numerical vs analytic evaluation of the KL divergence

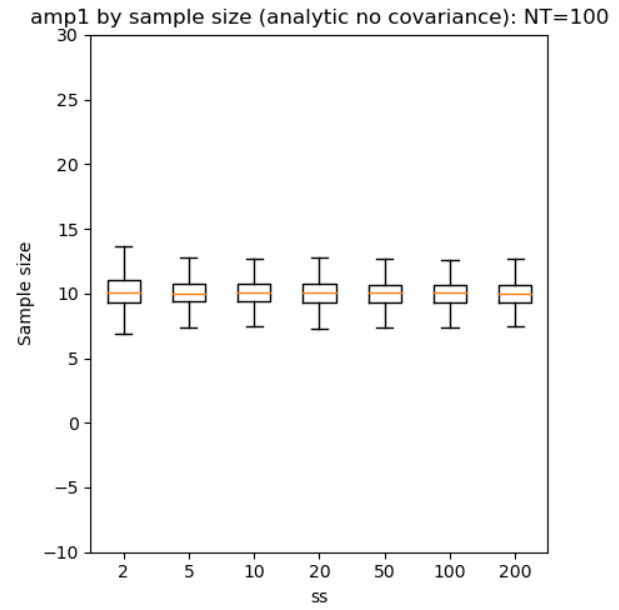
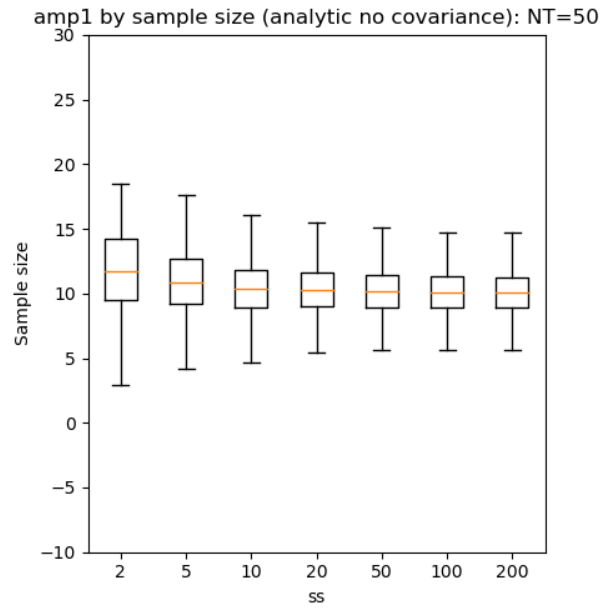
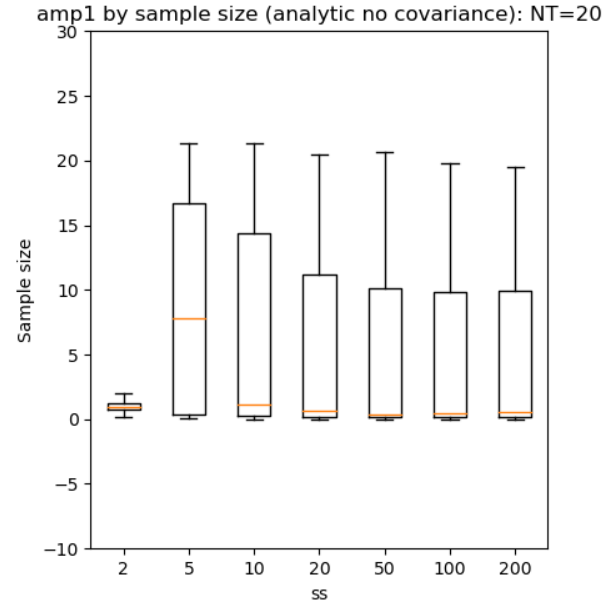
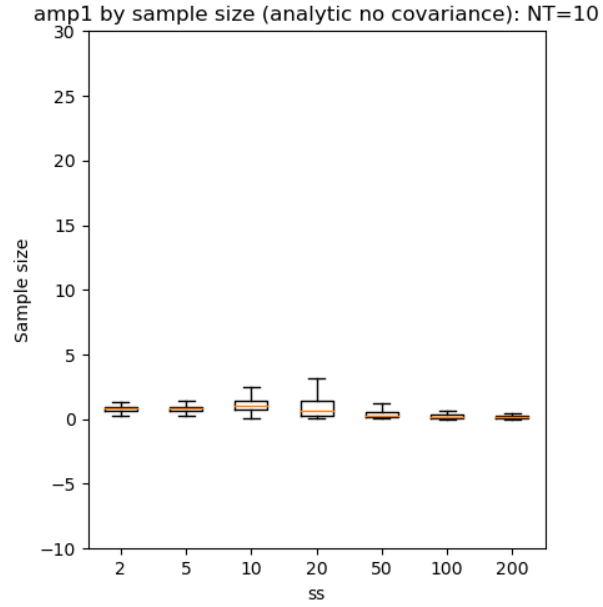
In the results above we have used the analytic result for the KL divergence of two multivariate Gaussian distributions. In general where the posterior is not constrained to this distribution we need to use a numerical evaluation which involves the posterior sample. So it is useful to assess the effect of forcing the numerical method in this case, particularly in combination with variation in the sample size.



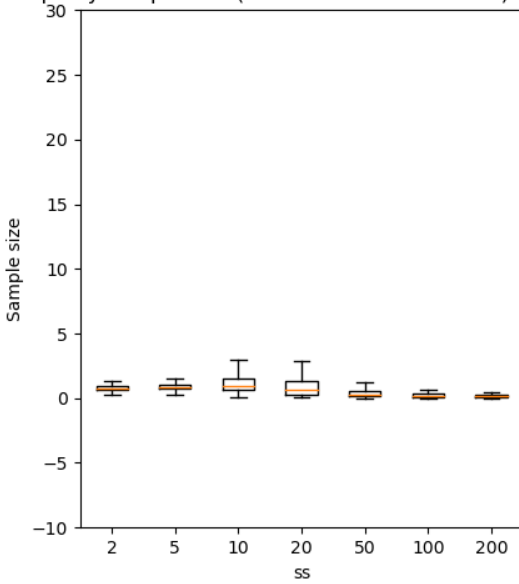


The absolute values of the free energy cannot be compared directly since some constant terms in the analytic solution are dropped from the calculation. The convergence properties with sample size, however, are closely similar even though part of the cost is independent of sample size in the analytic case.

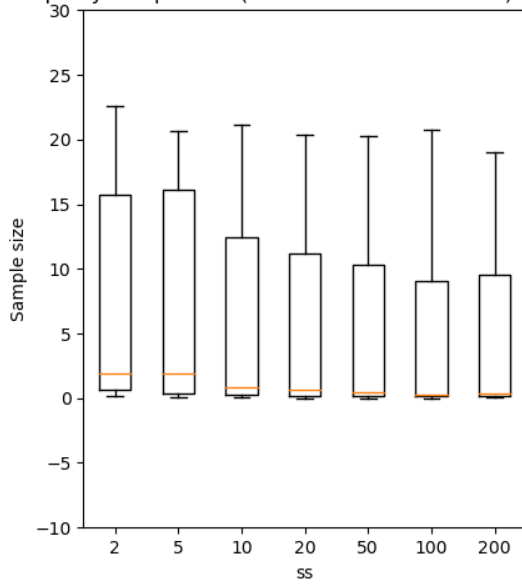
We can also compare parameter convergence with sample size:



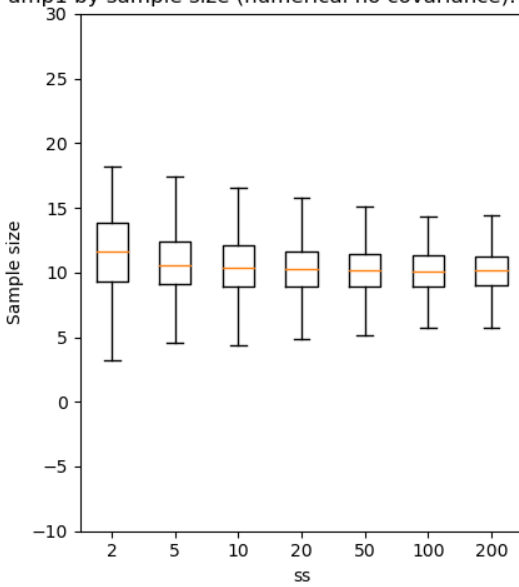
amp1 by sample size (numerical no covariance): NT=10



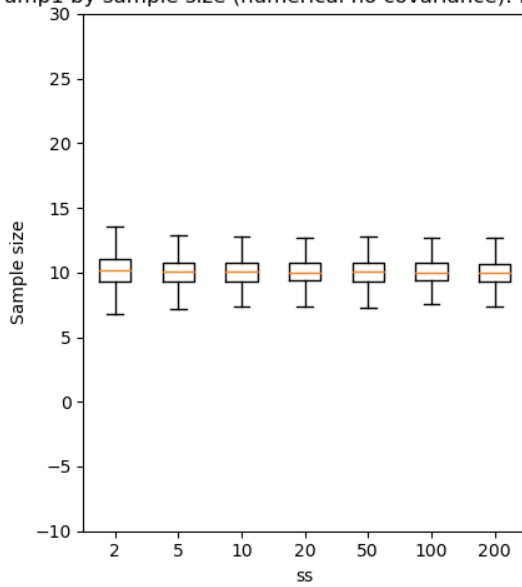
amp1 by sample size (numerical no covariance): NT=20



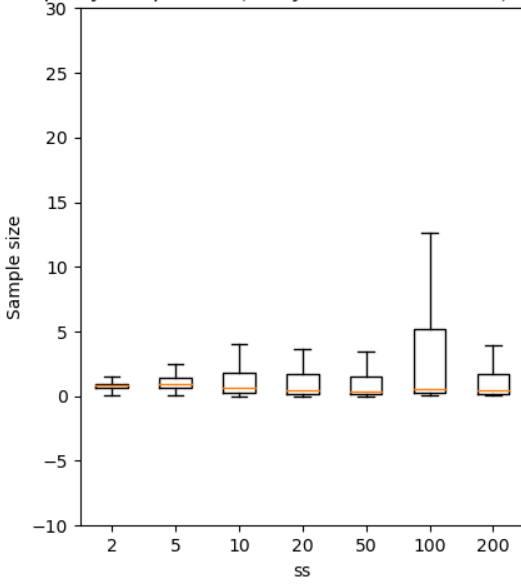
amp1 by sample size (numerical no covariance): NT=50



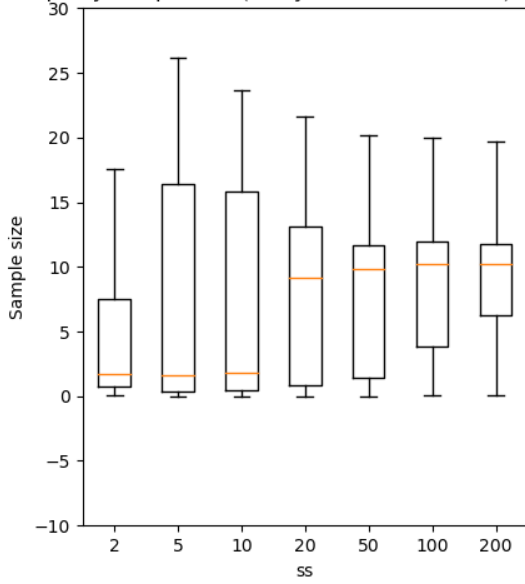
amp1 by sample size (numerical no covariance): NT=100



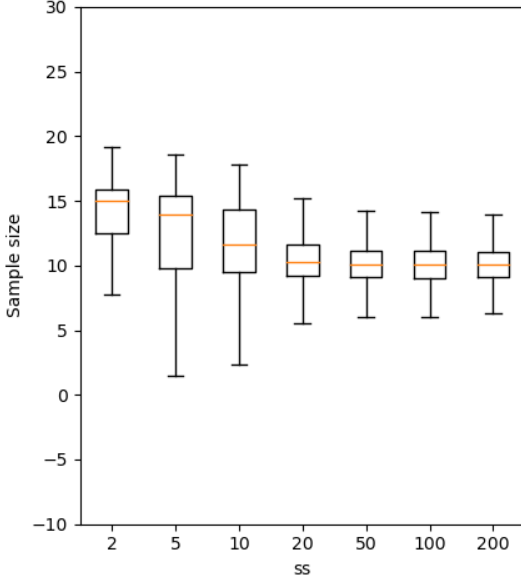
amp1 by sample size (analytic with covariance): NT=10



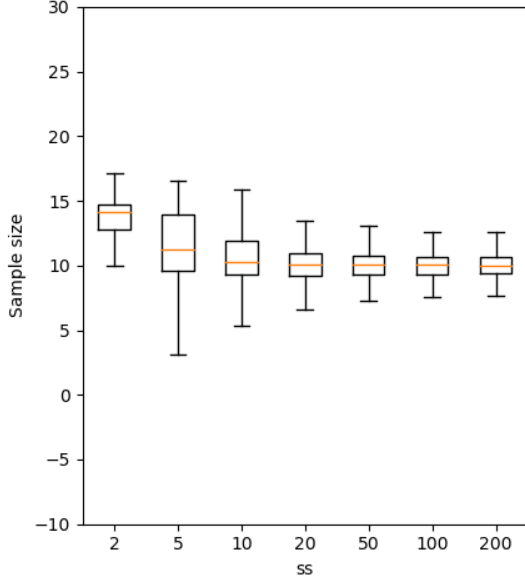
amp1 by sample size (analytic with covariance): NT=20



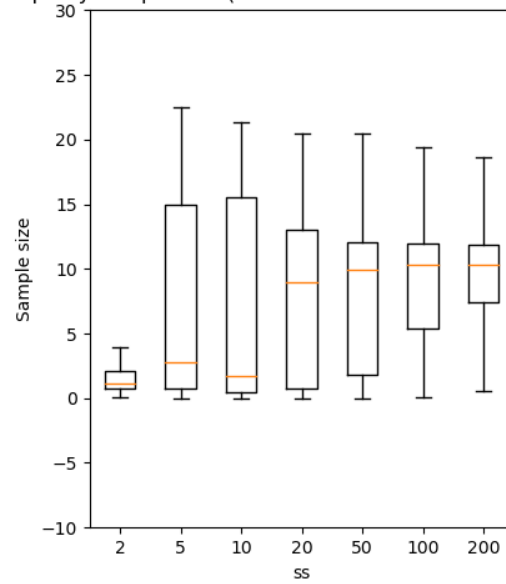
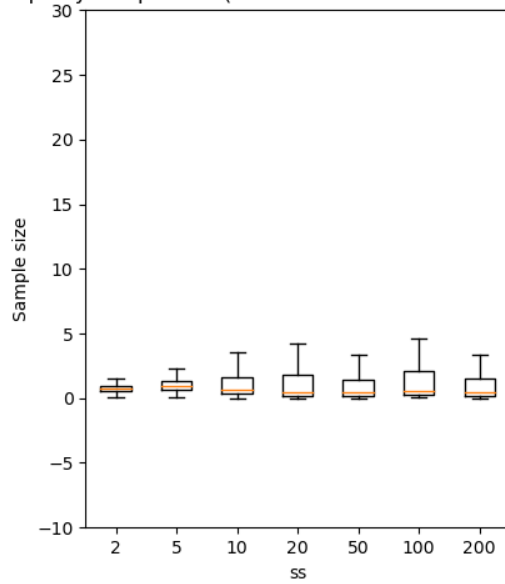
amp1 by sample size (analytic with covariance): NT=50



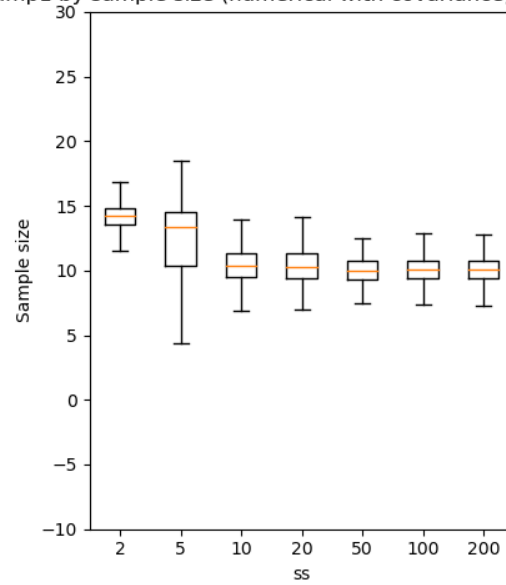
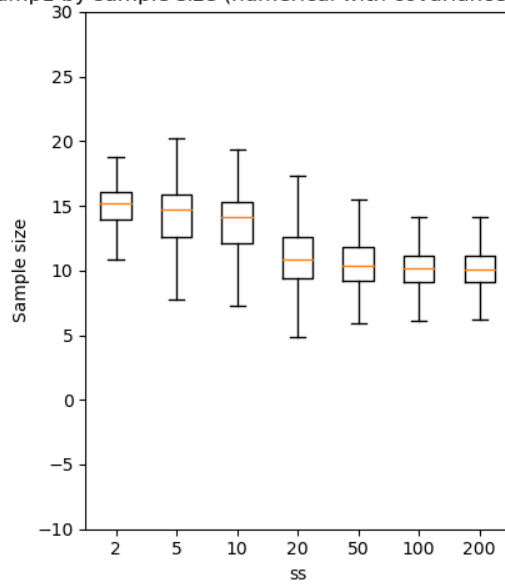
amp1 by sample size (analytic with covariance): NT=100

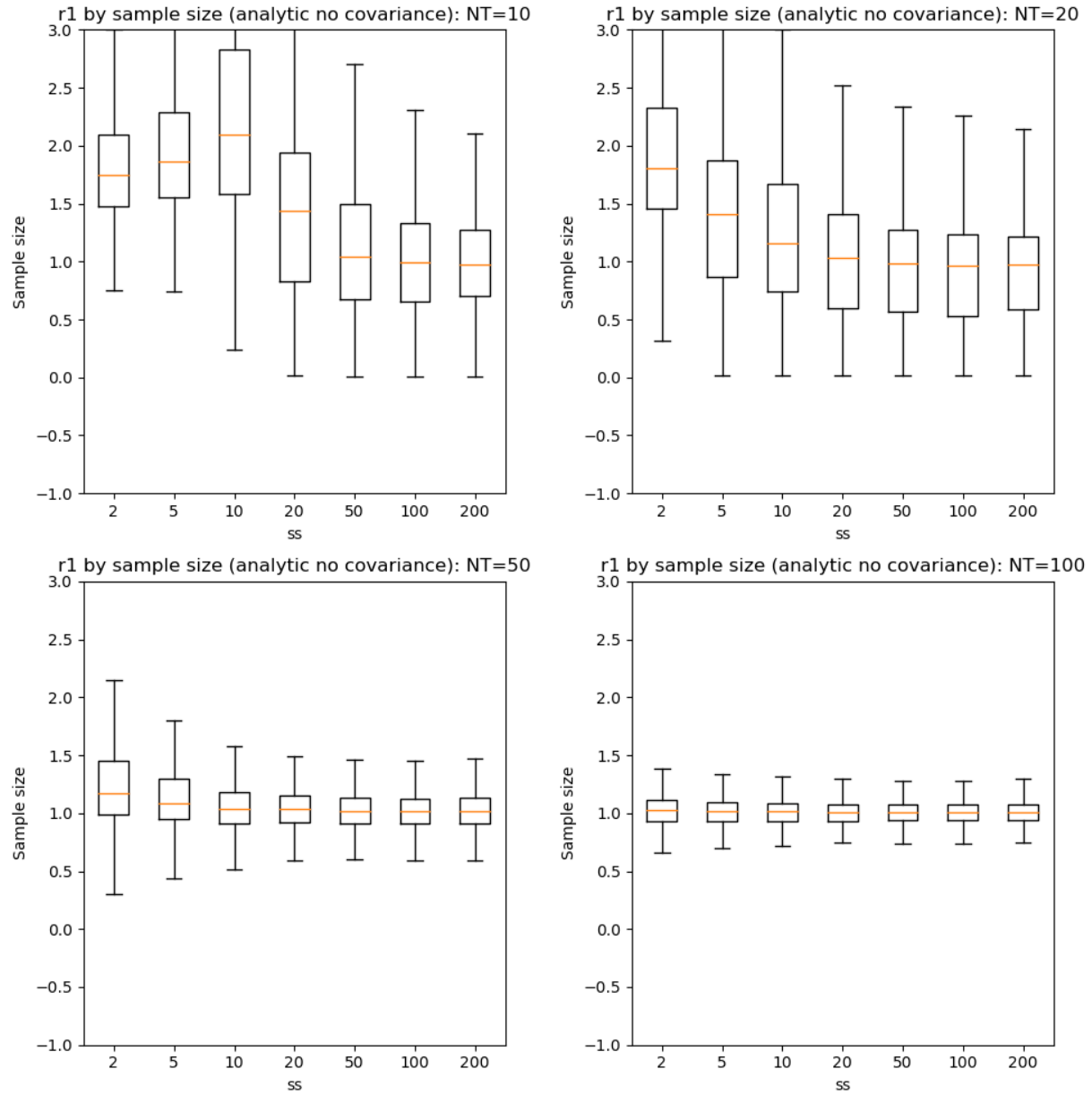


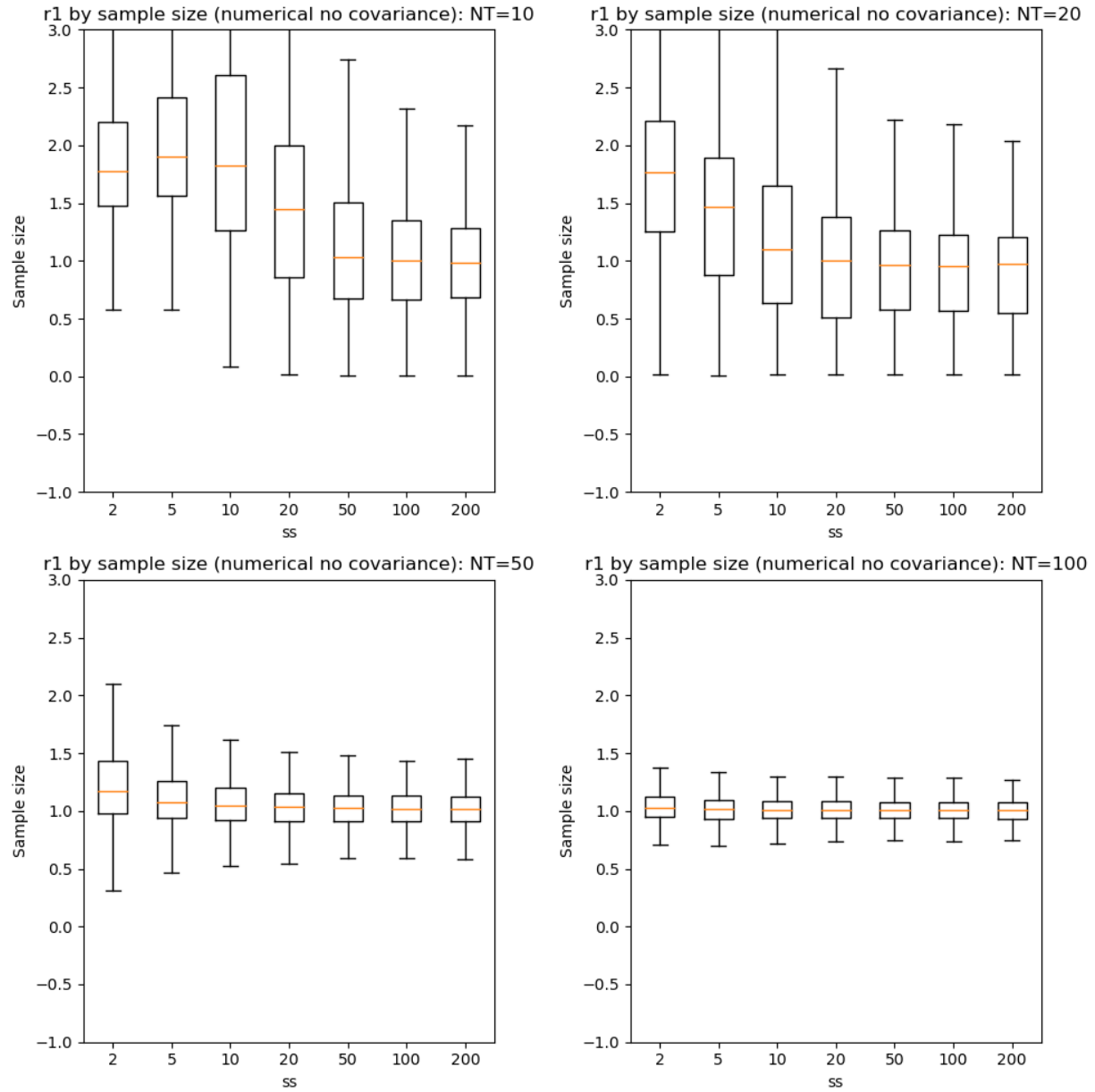
amp1 by sample size (numerical with covariance): NT=10 amp1 by sample size (numerical with covariance): NT=20

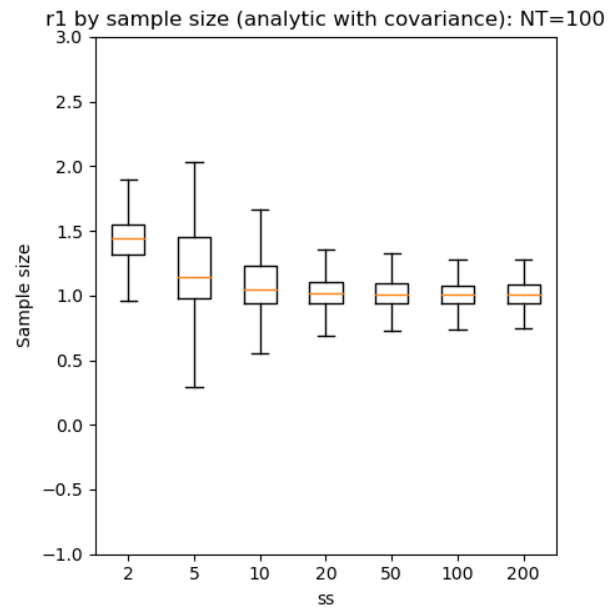
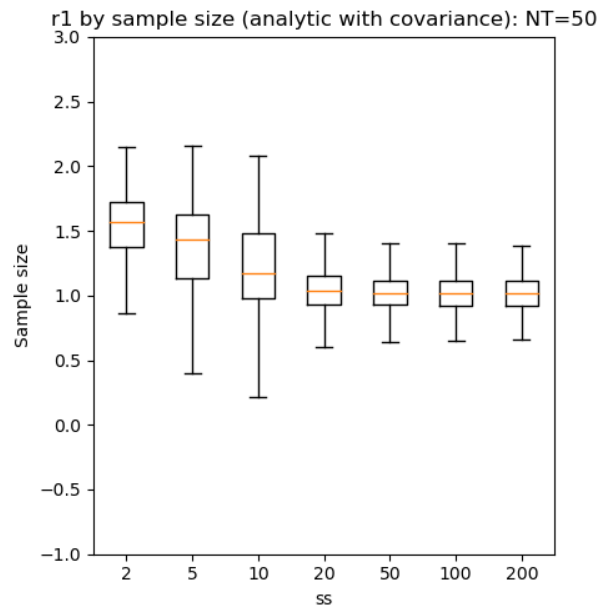
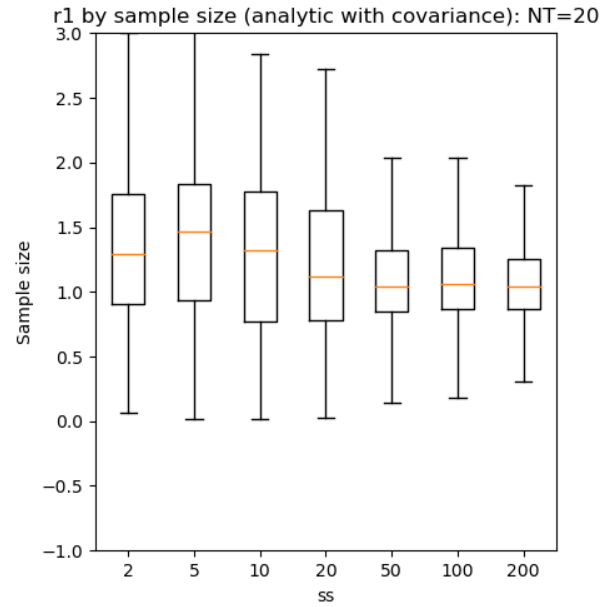
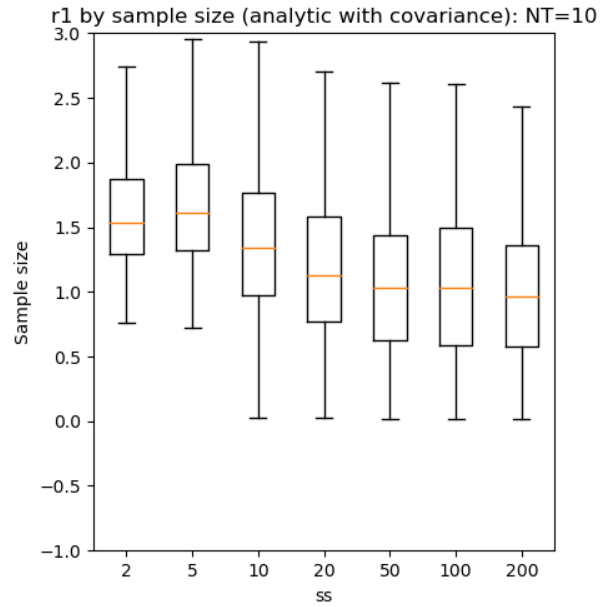


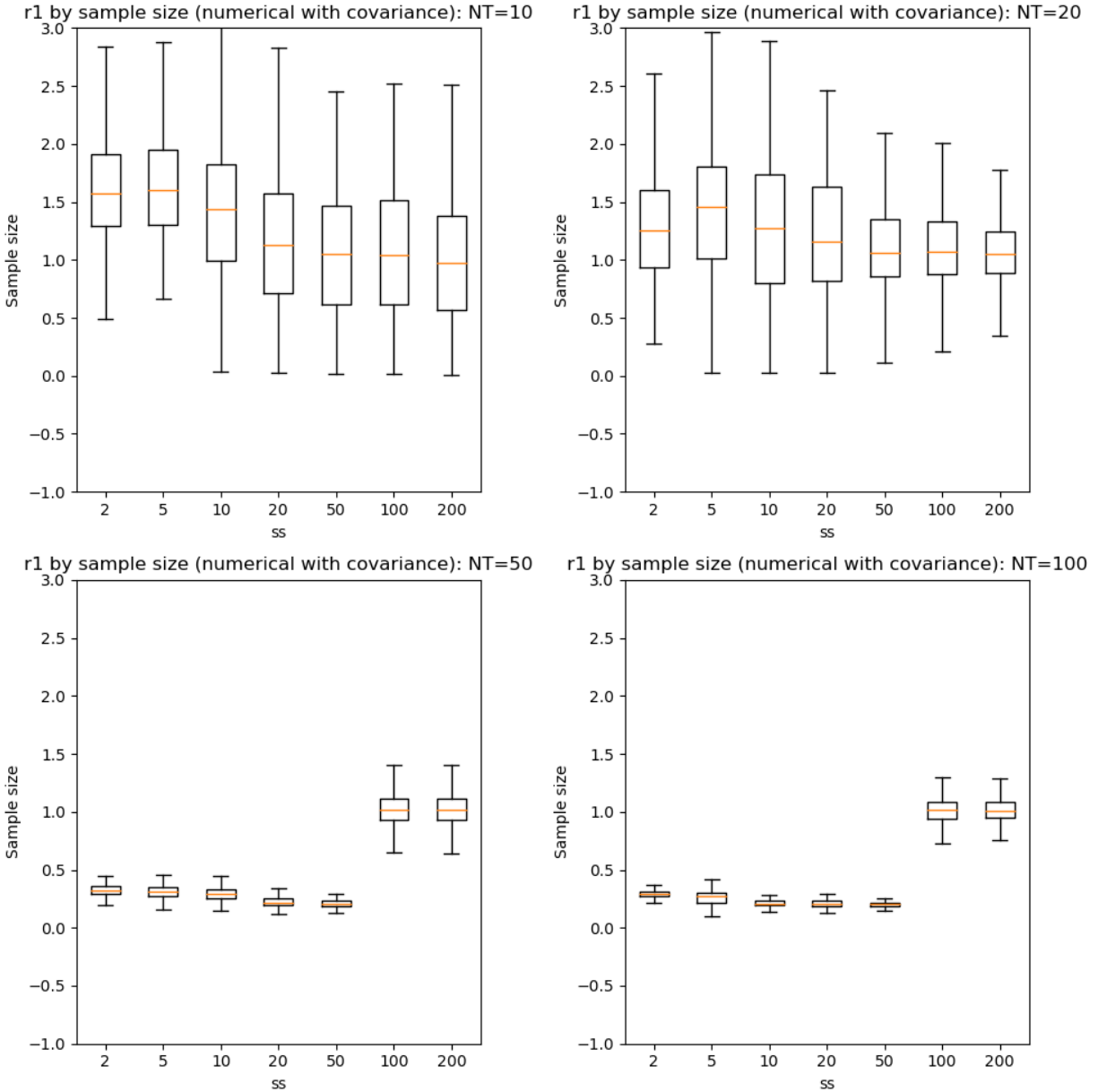
amp1 by sample size (numerical with covariance): NT=50 amp1 by sample size (numerical with covariance): NT=100











In most cases the numerical and analytic solutions seem very similar, however in the case of the rate parameter we do not appear to get a converged result at NT=50 or 100 until we have a sample size of 100 when inferring covariance. *This requires additional investigation since it is out of step with the remainder of the results.*

3.8 Inference of covariance

The effect of inferring covariance or not has been shown throughout these tests. In general the effect is that convergence is more challenging with covariance as would be expected with the increased parameter space, and instabilities caused by small batch or sample sizes, or large learning rates, are exacerbated by the inclusion of covariance. It's worth mentioning that the symmetry of the biexponential model would expect to generate significant parameter covariances.

A strategy of initially optimizing without covariance, and then restarting the optimization with the covariance parameters included is an obvious way to address this.

Tests using Arterial Spin Labelling model

This model implements a basic resting-state ASL kinetic model for PASL and pCASL acquisitions. The model parameters are f_{tiss} , the relative perfusion and δt the transit time of the blood from the labelling plane to the voxel.

Time points are divided into two categories:

During bolus is defined as $\delta t < t \leq \tau + \delta t$

Post bolus is defined as $t > \tau + \delta t$

Here τ is the bolus duration. The model output is zero for pre-bolus time points.

The following rate constant is defined:

$$\frac{1}{T_{1app}} = \frac{1}{(1/T_1 + f_{calib}/\lambda)}$$

λ is the tissue/blood partition coefficient of water which we take to be 0.9. f_{calib} is the calibrated CBF which typically we do not have access to (since we are inferring relative CBF) so we use a typical value of 0.01 s^{-1} .

4.1 CASL model

4.1.1 During bolus

$$M(t) = 2f_{tiss}T_{1app} \exp\left(\frac{-\delta t}{T_{1b}}\right) \left(1 - \exp\left(-\frac{(t-\delta t)}{T_{1app}}\right)\right)$$

4.1.2 Post bolus

$$M(t) = 2f_{tiss}T_{1app} \exp\left(-\frac{\delta t}{T_{1b}}\right) \exp\left(-\frac{(t-\tau-\delta t)}{T_{1app}}\right) \left(1 - \exp\left(-\frac{\tau}{T_{1app}}\right)\right)$$

4.2 PASL model

$$r = \frac{1}{T_{1app}} - \frac{1}{T_{1b}}$$

$$f = 2 \exp\left(-\frac{t}{T_{1app}}\right)$$

4.2.1 During bolus

$$M(t) = f_{tiss} \frac{f}{r} (\exp(rt) - \exp(r\delta t))$$

4.2.2 Post bolus

$$M(t) = f_{tiss} \frac{f}{r} (\exp(r(\delta t + \tau)) - \exp(r\delta t))$$

The time points in evaluating an ASL model are the T_i values, which may be expressed as the sum of the bolus duration τ and a post-labelling delay time. For 2D acquisitions they may be further modified by the additional time delay in acquiring each slice.

4.3 Test data

The test data used is a pCASL acquisition with $\tau = 1.8s$ and six post-labelling delays of 0.25, 0.5, 0.75, 1.0, 1.25 and 1.5s. The acquisition was 2D with an additional time delay of 0.0452s per slice. 8 repeats of the full set of PLDs was obtained.

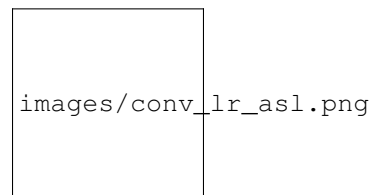
The test data was fitted in two ways. One method was to average over the repeats and fit the model to the repeat-free data. The other is to fit the model to the whole data including repeats. Naturally this involves a larger data size and hence a mini-batch approach to the optimization.

4.3.1 Mean data tests

For these tests we have only 6 time points and therefore we do not use a mini-batch approach, instead using a fixed batch size of 6 (all data points).

Convergence by learning rate

The convergence of mean cost by learning rate is shown below:



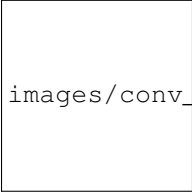

The pattern is closely similar to that obtained using a biexponential model although the convergence here is generally 'cleaner'. Learning rates between 0.05 and 0.1 attain the lowest cost within the given number of epochs, with 0.1 converging faster. Higher learning rates are less stable and do not appear to be likely to converge, while lower learning rates converge slowly.

The best cost achieved in 500 epochs is shown below, reinforcing the optimum learning rate range 0.1 - 0.05


`images/best_cost_lr_asl.png`

4.3.2 Full data tests

For these tests we have 8 repeats of the 6 PLDs giving 48 data points. This raises the possibility of a mini-batch approach. Intuitively the obvious choice of batch size is 6, arranged so that each optimization iteration considers one repeat of all 6 PLDs. However we experiment with varying the batch size to see if there is any actual advantage in this structure.


`images/conv_lr_asl_rpts.png`
`images/best_cost_lr_asl_rpts.png`

The patterns with convergence and batch size are very similar to those obtained from the biexponential model. In particular there is no visible effect of aligning the batch size with the ASL repeats. Again we find a general optimum learning rate of 0.1 - 0.05 associated with a batch size around 10, although it is noticeable that the best cost achieved at lower learning rates is a bit better with smaller batch sizes.

Sample size inflation tests

These tests are designed to explore the idea of increasing the posterior sample size during the course of the fitting. The theory is that we can start out with a small sample size which is very fast and will get us close to the optimal cost, then the size is increased to get a more accurate sample from the posterior and refine the optimization to be more accurate.

In principle this should be faster than simply using the larger sample size from the beginning and may also help to avoid local minima by encouraging a ‘noisier’ initial optimization.

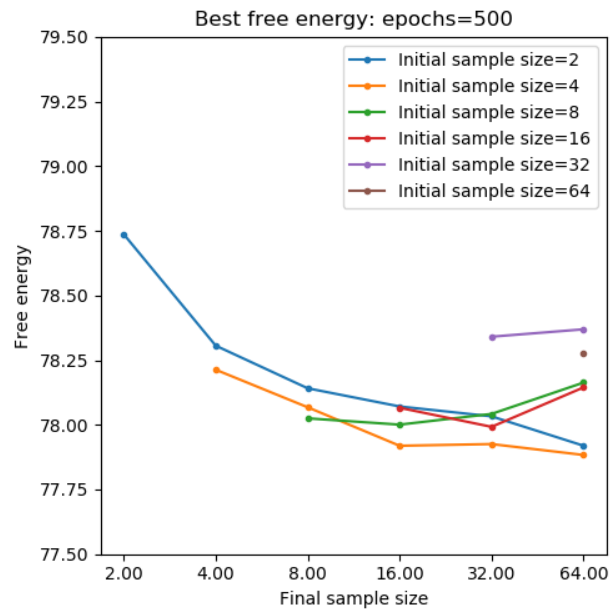
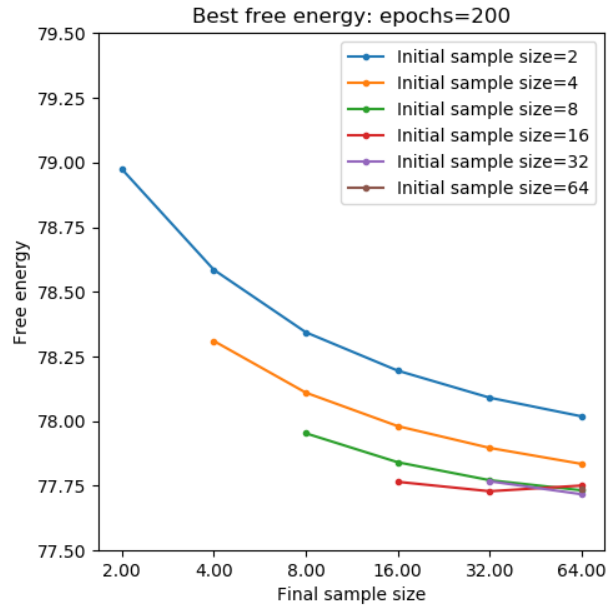
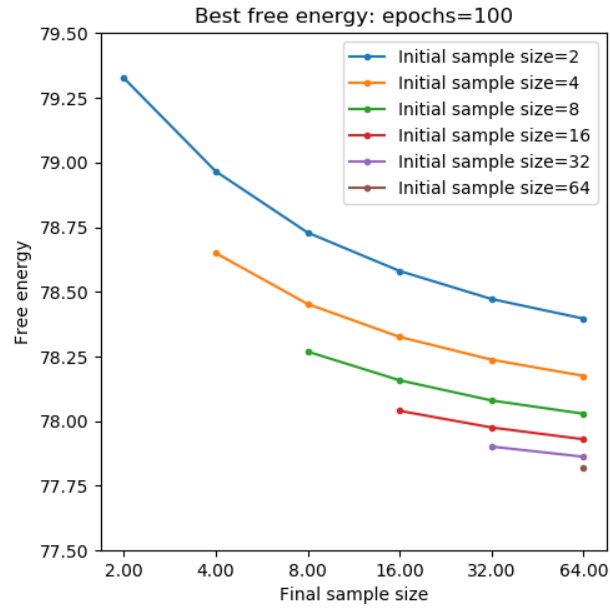
There are two key questions we need to answer when exploring this strategy:

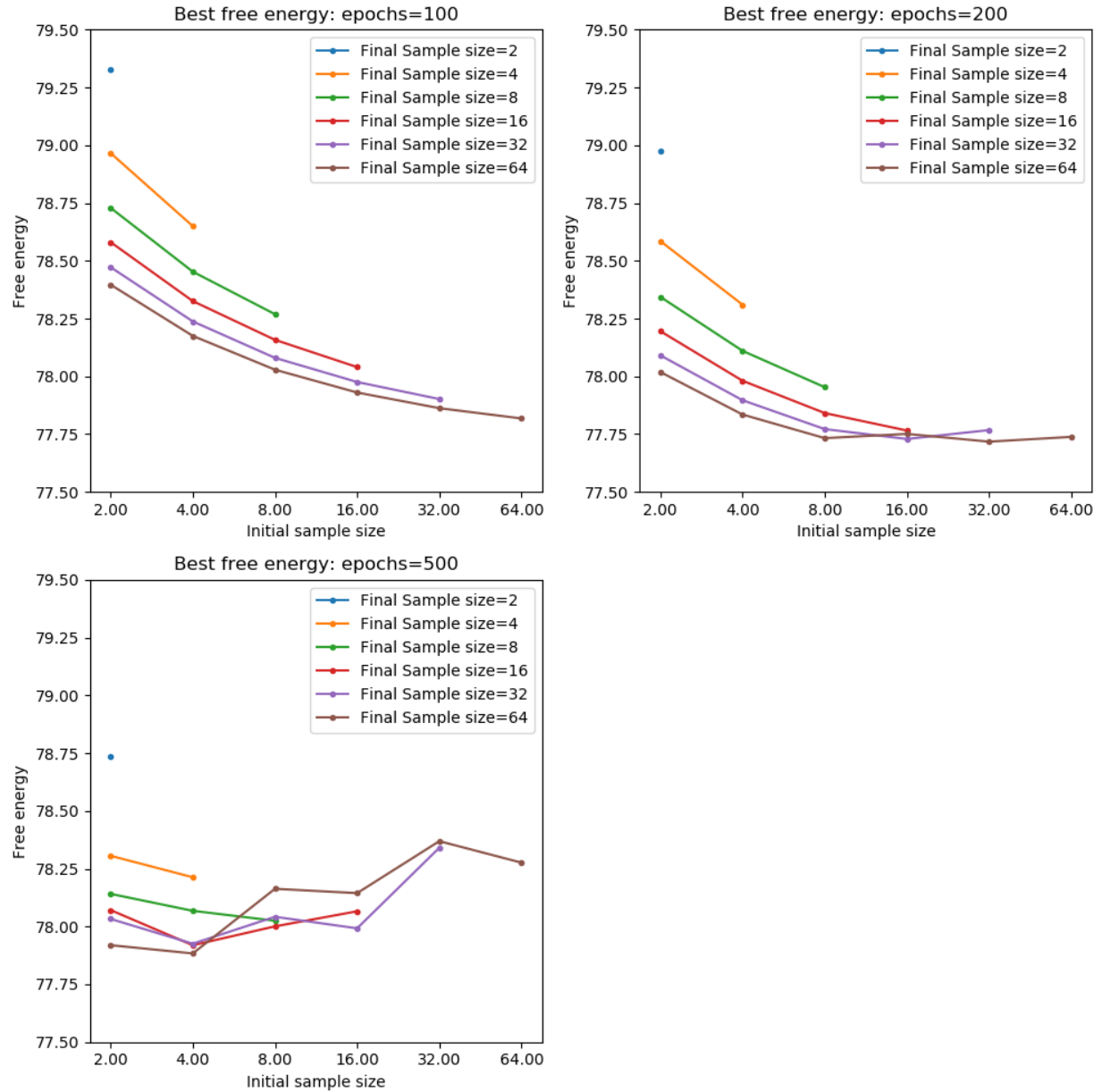
- What combination of initial sample size, final sample size, and number of training epochs is required to ensure we achieve convergence sufficiently close to the actual minimum cost for the data?
- For combinations which achieve within some given tolerance of this cost, which get there in the shortest time?

5.1 Tests using ASL data

These tests were performed using the multi-repeat ASL data described [here](#). We compared initial sample sizes of 2, 4, 8, 16, 32 and 64 growing to final sample sizes of 2, 4, 8, 16, 32 and 64 over 100, 200 and 500 epochs respectively.

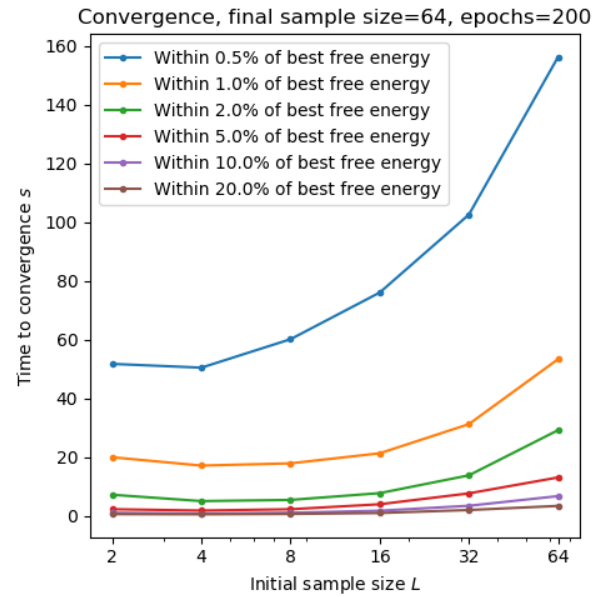
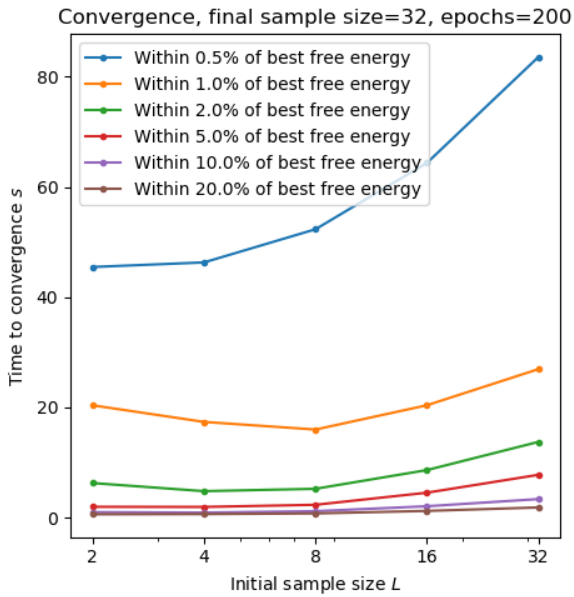
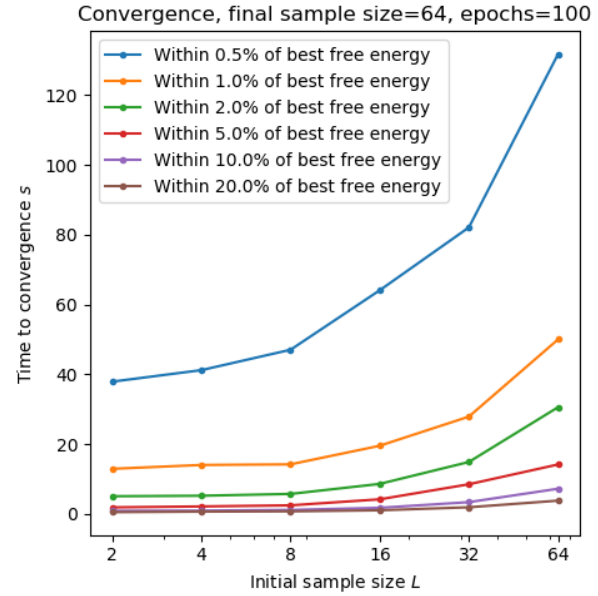
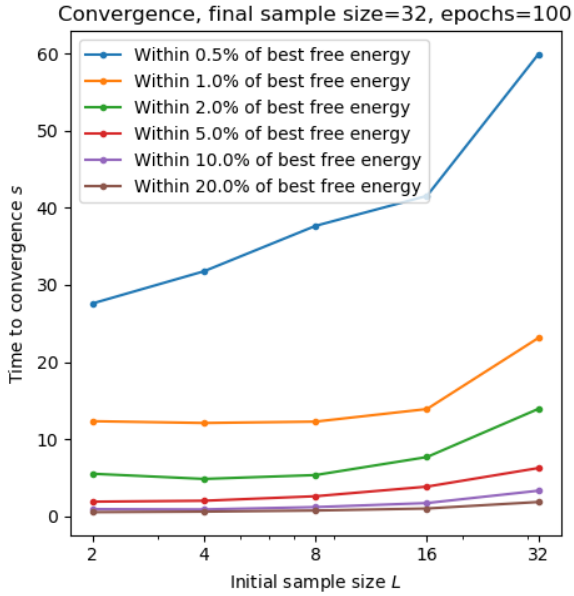
The following plots show the minimum cost achieved for each combination of initial and final sample size at each number of epochs (the two plots show the same data but one is focused on comparing initial sample sizes for a given final sample sizes and the other is focused on comparing final sample sizes for a given initial sample size):

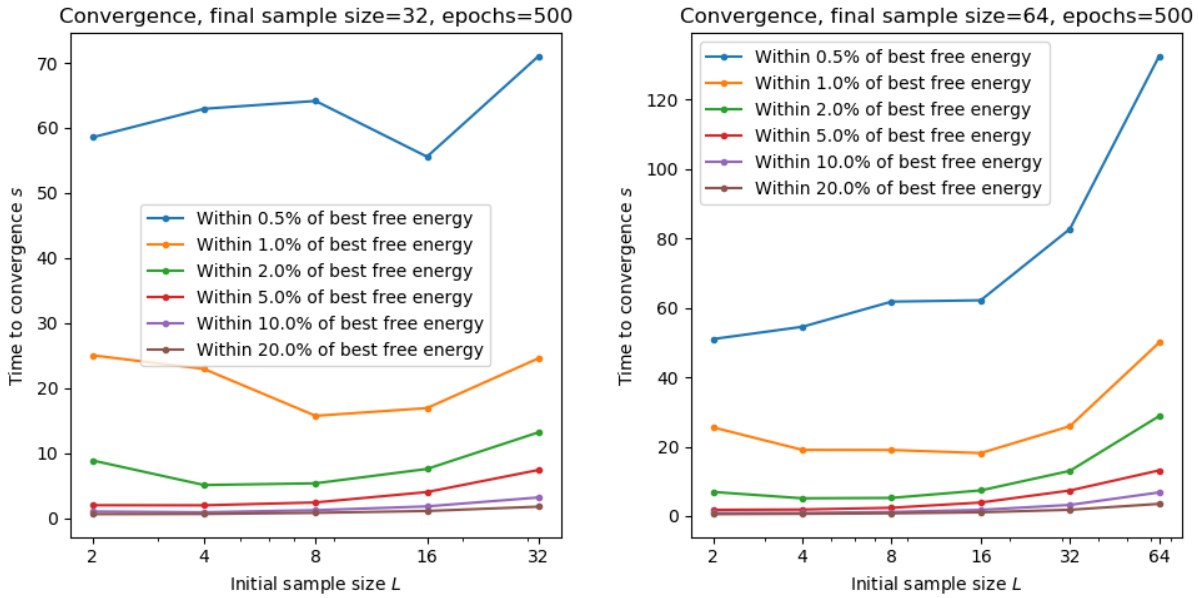




From these plots we can see that with only 100 training epochs we are not yet at absolute convergence even using the maximum sample size throughout. For 200 epochs, we achieve close to the optimal cost when the final sample size is 64 or 32 and the initial sample size is at least 8.

The following plots show the time taken to reach within a given tolerance of the best free energy for combinations of initial and final sample sizes. We only consider final sample sizes of 64 and 32 based on the previous results:





Here we see that starting with a smaller sample size is generally associated with faster overall convergence. For this data we would recommend an initial sample size of 8 and a final sample size of 64.

Learning rate quenching tests

These tests are designed to explore the idea of decreasing the learning rate during the course of the fitting. The theory is that we can start out with a high learning rate which rapidly converges close to the optimum, but then reduce it over time to get as close as possible and avoid the problem of large steps overshooting the minimum.

In principle this should be faster than simply using the lower learning rate throughout since the initial move towards the neighbourhood of the minimum should require fewer epochs. In addition if the initial learning rate is very low then the optimization may fail to escape from local cost minima that exist close to the initial values.

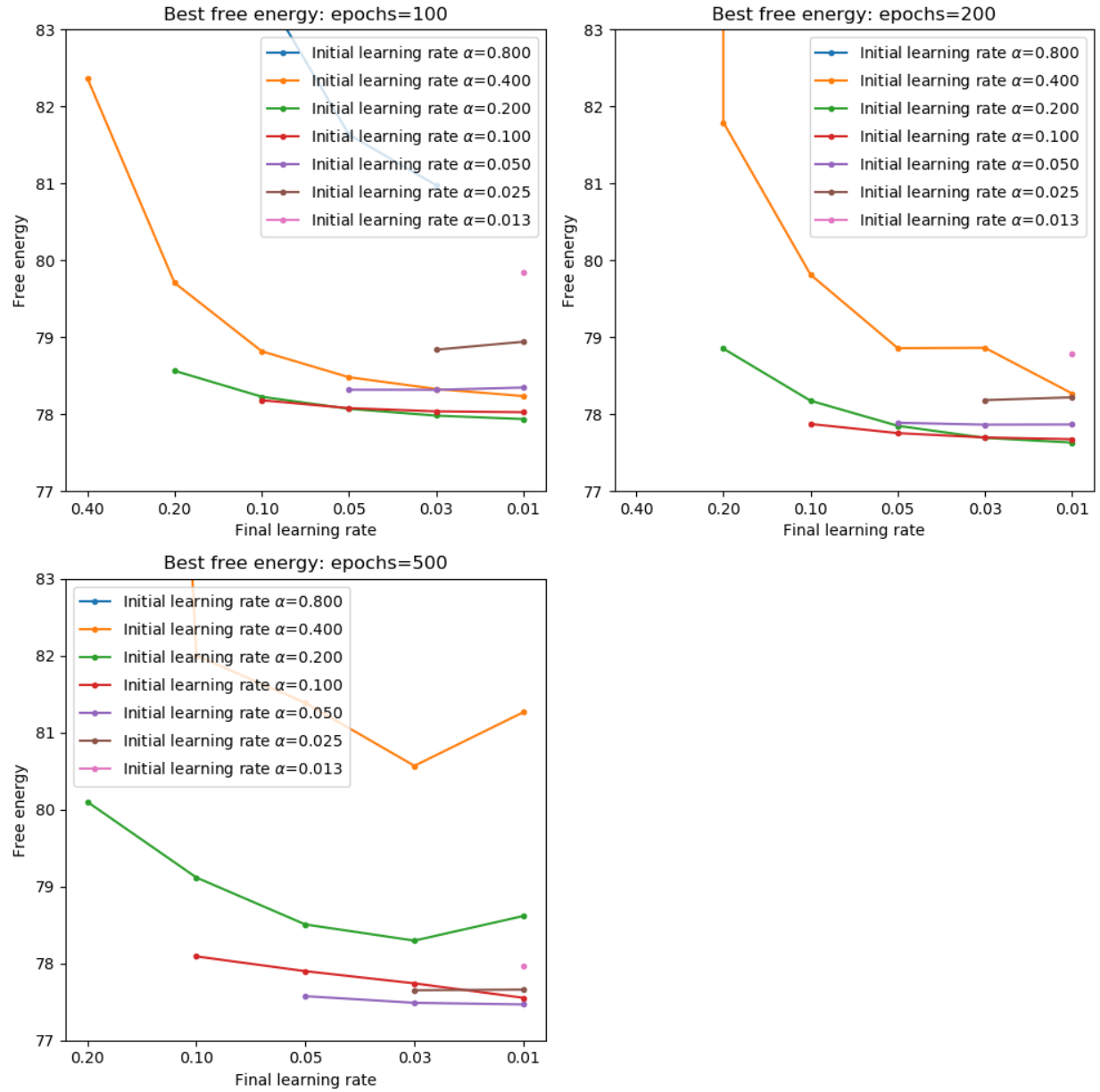
There are two key questions we need to answer when exploring this strategy:

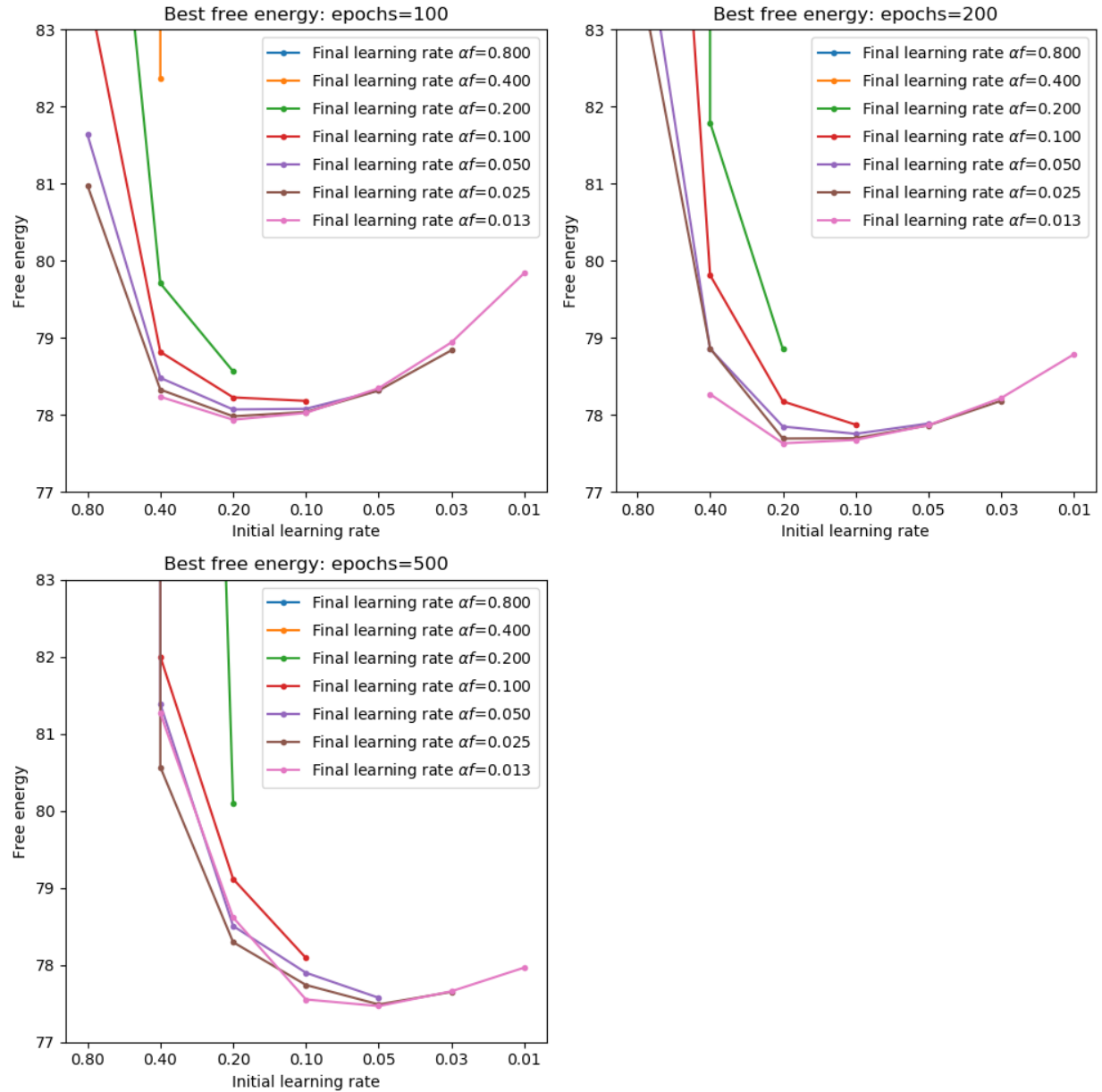
- What combination of initial learning rate, final learning rate, and number of training epochs is required to ensure we achieve convergence sufficiently close to the actual minimum cost for the data?
- For combinations which achieve within some given tolerance of this cost, which get there in the shortest time?

6.1 Tests using ASL data

These tests were performed using the multi-repeat ASL data described [here](#). We compared initial learning rates of 0.8, 0.4, 0.2, 0.1, 0.05, 0.025 and 0.0125 with reduction over the training cycle to final learning rates from the same set (but only running examples where the final learning rate was less than the initial. The training cycle was performed over 100, 200 and 500 epochs.

The following plot show the minimum cost achieved for each combination of initial and final learning rate at each number of epochs (the two plots show the same data but one is focused on comparing initial learning rates for a given final learning rates and the other is focused on comparing final learning rates for a given initial learning rate):





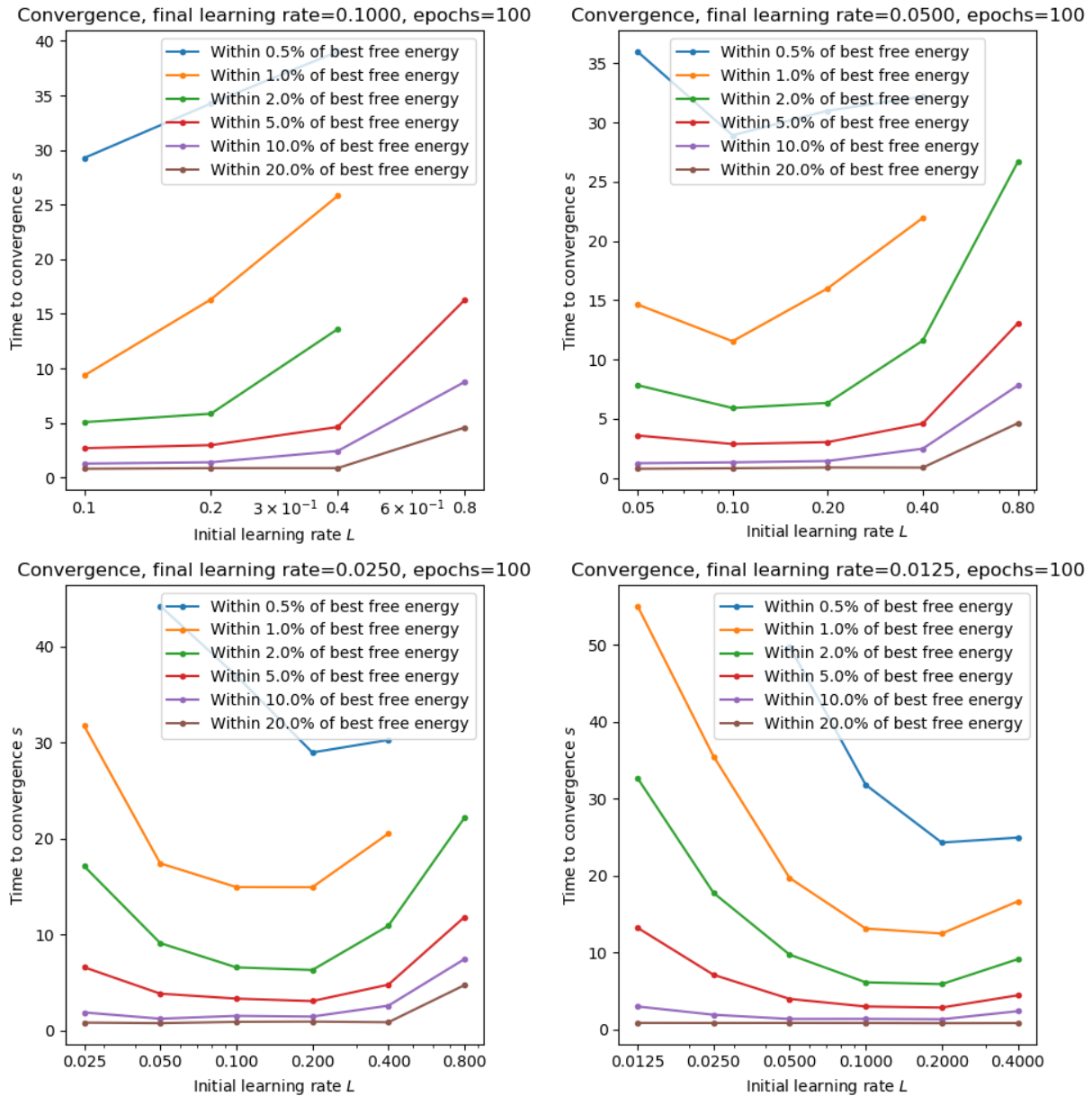
From these plots we can see that with firstly the minimum cost is achieved only when the final learning rate is sufficiently low - this confirms that a low learning rate is necessary to accurately home in on the minimum without overshooting.

Furthermore, a better cost is achieved by starting at a higher learning rate - 0.2 to 0.1. This is the case even with 500 epochs of training. So the general strategy of starting out with rapid learning and quenching to a very low value seems to be a good one.

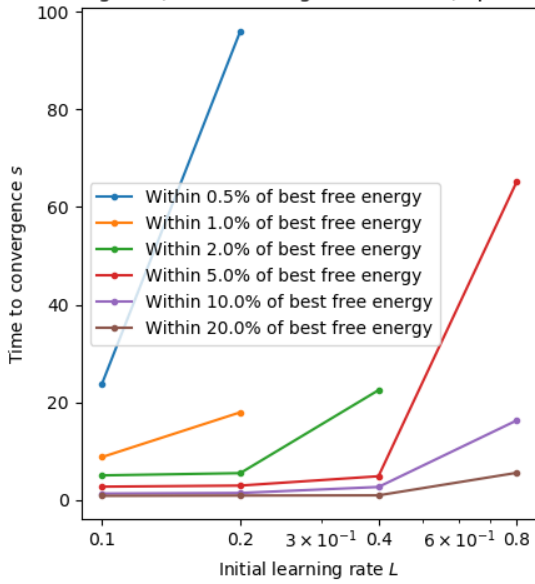
It is noticeable that we obtain *worse* cost over 500 epochs than over 200 epochs when the initial learning rate is high. It may be that maintaining a high learning rate for too many epochs leads the optimization far away from the optimum. This is confirmed by the actual runtime free energy which starts out by reducing but rapidly begins to oscillate between high and low values if the high learning rate is continued.

Best cost over 100 epochs was 77.9 (initial 0.2 -> final 0.0125). Over 200 epochs the best cost was 77.6 (same combination) and over 500 epochs the best was 77.5 (initial 0.05 -> final 0.0125).

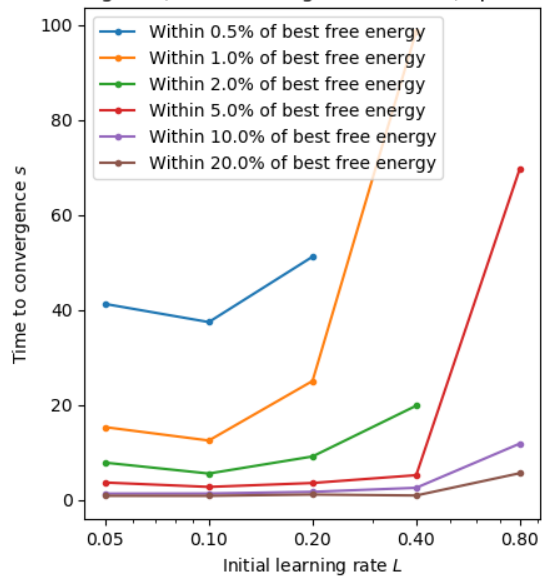
The following plots show the time taken to reach within a given tolerance of the best free energy for combinations of initial and final learning rates. We only consider final learning rates of 0.1 or lower as previous plots show that we are not close to convergence when then final learning rate is higher:



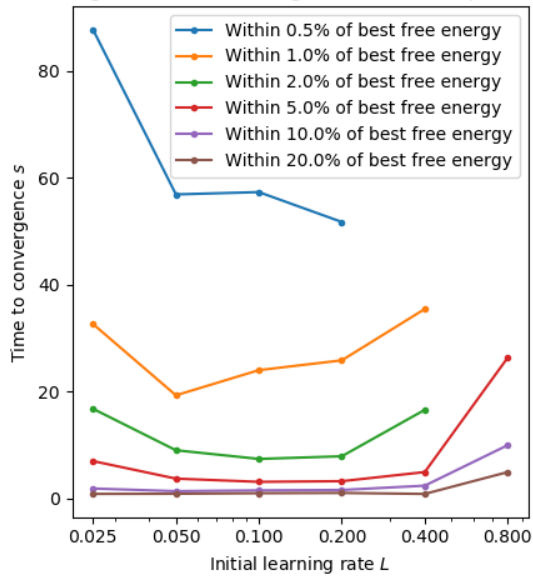
Convergence, final learning rate=0.1000, epochs=200



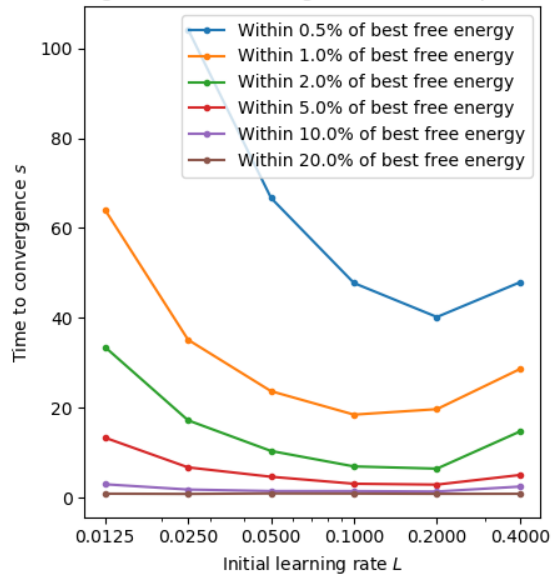
Convergence, final learning rate=0.0500, epochs=200

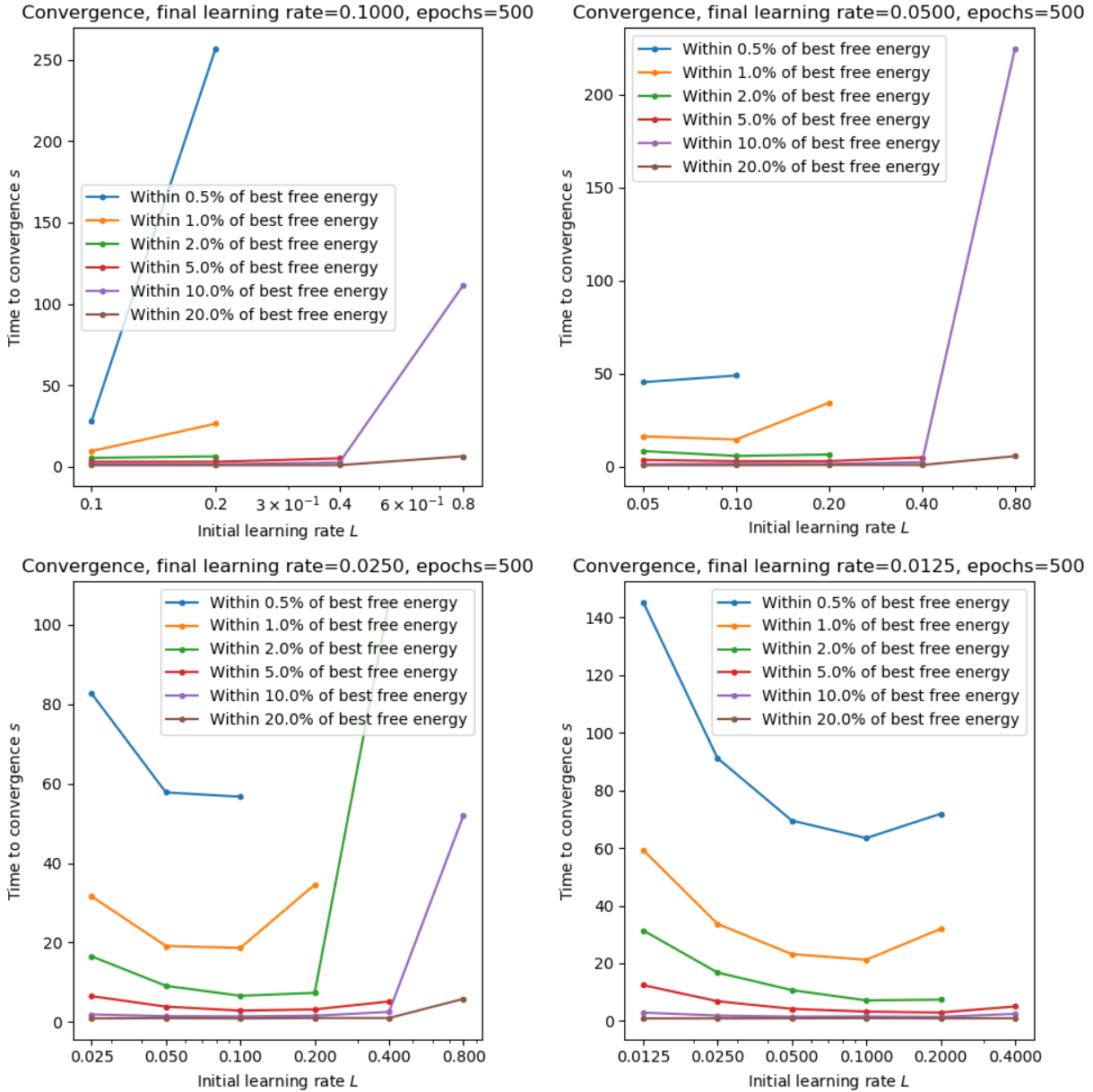


Convergence, final learning rate=0.0250, epochs=200



Convergence, final learning rate=0.0125, epochs=200





These plots show that starting with a high learning rate can indeed accelerate convergence provided it is quenched rapidly. Slow quenching (e.g. from 0.2 to 0.1) seems to leave too many training epochs at the higher learning rate and can reduce the convergence speed.

It's worth noting that the computational time per epoch is largely independent of learning rate, so these measures are essentially measures of how many epochs were needed for convergence (unlike with sample size inflation where the training time increases as we increase the sample size). So slower convergence can be a result of too much time initially at a low learning rate (where the optimizer slowly 'inches' its way towards the minimum) or alternatively too much time initially at a high learning rate, where the optimizer repeatedly overshoots the minimum until the quenching process lowers it sufficiently to converge.

For this data and appropriate combination seems to be 200 epochs starting at 0.2 and reducing by a factor of 16 to 0.0125. This gives a cost very close to the minimum and also optimizes the convergence rate measure.

Command line usage

Our implementation of Stochastic Variational Bayes includes a command line application designed to be similar to [Fabber](#) (our implementation of analytic Variational Bayes). The command line program is simply named `svb`

7.1 Examples

To fit the ASL data given in the FSL course we would use the following command line:

```
svb --data=mpld_asltc.nii.gz --casl --plds=0.25,0.5,0.75,1.0,1.25,1.5 --slicedt=0.  
↪0452 \  
  --tau=1.8 --repeats=8 \  
  --mask=mpld_asltc_mask.nii.gz \  
  --model=aslrest \  
  --output=mpld_asltc_out
```

8.1 Model module

Base class for a forward model whose parameters are to be fitted

class `svb.model.Model` (*data_model*, ***options*)

A forward model

Attr `params` Sequence of `Parameter` objects

Attr `nparams` Number of model parameters

evaluate (*params*, *tpts*)

Evaluate the model

Parameters *t* – Time values to evaluate the model at, supplied as a tensor of shape $[1 \times 1 \times B]$ (if time values at each voxel are identical) or $[V \times 1 \times B]$ otherwise.

:param `params` Sequence of parameter values arrays, one for each parameter. Each array is $W \times S \times 1$ tensor where W is the number of parameter vertices and S is the number of samples per parameter. This may be supplied as a $P \times V \times S \times 1$ tensor where P is the number of parameters.

Returns $[V \times S \times B]$ tensor containing model output at the specified time values for each voxel, and each sample (set of parameter values).

ievaluate (*params*, *tpts*)

Evaluate the model outside of a TensorFlow session

Same as `evaluate()` but will run the evaluation within a session and return the evaluated output tensor

log_config (*log=None*)

Write model configuration to a log stream

Param `log` Optional logger to use - defaults to class instance logger

nparams

Number of parameters in the model

param_idx (*name*)

Returns the index of a named parameter

test_data (*tpts, params_map*)

Generate test data by evaluating the model on known parameter values with optional added noise

FIXME this is non-functional at present.

Parameters

- **tpts** – 1xN or MxN tensor of time values (possibly varying by voxel)
- **params_map** – Mapping from parameter name either a single parameter value or a sequence of M parameter values. The special key `noise_sd`, if present, should containing the standard deviation of Gaussian noise to add to the output.

:return If noise is present, a tuple of two MxN Numpy arrays. The first contains the ‘clean’ output data without noise, the second contains the noisy data. If noise is not present, only a single array is returned.

tpts ()

Get the full set of timeseries time values

Parameters

- **n_tpts** – Number of time points required for the data to be fitted
- **shape** – Shape of source data which may affect the times assigned

By default this is a linear space using the attributes `t0` and `dt`. Some models may have time values fixed by some other configuration. If the number of time points is fixed by the model it must match the supplied value `n_tpts`.

Returns Either a Numpy array of shape `[n_tpts]` or a Numpy array of shape `shape + [n_tpts]` for voxelwise timepoints

`svb.model.get_model_class` (*model_name*)

Get a model class by name

8.2 Parameter module

SVB - Model parameters

This module defines a set of classes of model parameters.

The factory methods which create priors/posteriors can make use of the instance class to create the appropriate type of vertexwise prior/posterior

class `svb.parameter.Parameter` (*name, **kwargs*)

A standard model parameter

`svb.parameter.get_parameter` (*name, **kwargs*)

Factory method to create an instance of a parameter

8.3 Posterior module

Definition of the posterior distribution

```

class svb.posterior.FactorisedPosterior(posts, **kwargs)
    Posterior distribution for a set of parameters with no covariance

    entropy(_samples=None)

        Parameters samples – A tensor of shape [W, P, S] where W is the number of parameter
            vertices, P is the number of parameters in the prior (possibly 1) and S is the number of
            samples. This parameter may or may not be used in the calculation. If it is required, the
            implementation class must check that it is provided

        :return Tensor of shape [W] containing vertexwise distribution entropy

    latent_loss(prior)
        Analytic expression for latent loss which can be used when posterior and prior are Gaussian

        https://en.wikipedia.org/wiki/Multivariate\_normal\_distribution#Kullback%E2%80%93Leibler\_
        divergence

        Parameters prior – Vertexwise Prior instance which defines the mean and cov vertices at-
            tributes

    log_det_cov()
        Determinant of diagonal matrix is product of diagonal entries

    sample(nsamples)

        Parameters nsamples – Number of samples to return per parameter vertex / parameter

        Returns A tensor of shape [W, P, S] where W is the number of parameter vertices, P is the
            number of parameters in the distribution (possibly 1) and S is the number of samples

    set_state(state)

        Parameters state – State of variables in this posterior, as returned by previous call to state()

        :return Sequence of tf.Operation objects containing which will set the variables in this posterior to
            the specified state

    state()
        :return Sequence of tf.Tensor objects containing the state of all variables in this posterior. The tensors
            returned will be evaluated to create a savable state which may then be passed back into set_state()

class svb.posterior.GaussianGlobalPosterior(idx, mean, var, **kwargs)
    Posterior which has the same value at every parameter vertex

    entropy(_samples=None)

        Parameters samples – A tensor of shape [W, P, S] where W is the number of parameter
            vertices, P is the number of parameters in the prior (possibly 1) and S is the number of
            samples. This parameter may or may not be used in the calculation. If it is required, the
            implementation class must check that it is provided

        :return Tensor of shape [W] containing vertexwise distribution entropy

    sample(nsamples)
        FIXME should each parameter vertex get the same sample? Currently YES

    set_state(state)

        Parameters state – State of variables in this posterior, as returned by previous call to state()

        :return Sequence of tf.Operation objects containing which will set the variables in this posterior to
            the specified state

```

```

state()
    :return Sequence of tf.Tensor objects containing the state of all variables in this posterior. The tensors
        returned will be evaluated to create a savable state which may then be passed back into set_state()

class svb.posterior.MVNPosterior(posts, **kwargs)
    Multivariate Normal posterior distribution

    entropy(_samples=None)
        Parameters samples – A tensor of shape [W, P, S] where W is the number of parameter
            vertices, P is the number of parameters in the prior (possibly 1) and S is the number of
            samples. This parameter may or may not be used in the calculation. If it is required, the
            implementation class must check that it is provided

        :return Tensor of shape [W] containing vertexwise distribution entropy

    log_det_cov()
        Determinant of a matrix can be calculated from the Cholesky decomposition which may be faster and more
        stable than tf.matrix_determinant

    sample(nsamples)
        Parameters nsamples – Number of samples to return per parameter vertex / parameter

        Returns A tensor of shape [W, P, S] where W is the number of parameter vertices, P is the
            number of parameters in the distribution (possibly 1) and S is the number of samples

    set_state(state)
        Parameters state – State of variables in this posterior, as returned by previous call to state()

        :return Sequence of tf.Operation objects containing which will set the variables in this posterior to
            the specified state

state()
    :return Sequence of tf.Tensor objects containing the state of all variables in this posterior. The tensors
        returned will be evaluated to create a savable state which may then be passed back into set_state()

class svb.posterior.NormalPosterior(idx, mean, var, **kwargs)
    Posterior distribution for a single vertexwise parameter with a normal distribution

    entropy(_samples=None)
        Parameters samples – A tensor of shape [W, P, S] where W is the number of parameter
            vertices, P is the number of parameters in the prior (possibly 1) and S is the number of
            samples. This parameter may or may not be used in the calculation. If it is required, the
            implementation class must check that it is provided

        :return Tensor of shape [W] containing vertexwise distribution entropy

    sample(nsamples)
        Parameters nsamples – Number of samples to return per parameter vertex / parameter

        Returns A tensor of shape [W, P, S] where W is the number of parameter vertices, P is the
            number of parameters in the distribution (possibly 1) and S is the number of samples

    set_state(state)
        Parameters state – State of variables in this posterior, as returned by previous call to state()

        :return Sequence of tf.Operation objects containing which will set the variables in this posterior to
            the specified state

```

```

state ()
    :return Sequence of tf.Tensor objects containing the state of all variables in this posterior. The tensors
    returned will be evaluated to create a savable state which may then be passed back into set_state()

class svb.posterior.Posterior (idx, **kwargs)
    Posterior distribution

    entropy (samples=None)

        Parameters samples – A tensor of shape [W, P, S] where W is the number of parameter
        vertices, P is the number of parameters in the prior (possibly 1) and S is the number of
        samples. This parameter may or may not be used in the calculation. If it is required, the
        implementation class must check that it is provided

        :return Tensor of shape [W] containing vertexwise distribution entropy

    sample (nsamples)

        Parameters nsamples – Number of samples to return per parameter vertex / parameter

        Returns A tensor of shape [W, P, S] where W is the number of parameter vertices, P is the
        number of parameters in the distribution (possibly 1) and S is the number of samples

    set_state (state)

        Parameters state – State of variables in this posterior, as returned by previous call to state()

        :return Sequence of tf.Operation objects containing which will set the variables in this posterior to
        the specified state

    state ()

        :return Sequence of tf.Tensor objects containing the state of all variables in this posterior. The ten-
        sors returned will be evaluated to create a savable state which may then be passed back into set_state()

svb.posterior.get_posterior (idx, param, t, data_model, **kwargs)
    Factory method to return a posterior

        Parameters param – svb.parameter.Parameter instance

    :

```

8.4 Prior module

Definition of prior distribution

```

class svb.prior.ARDPrior (nvertices, mean, var, **kwargs)
    Automatic Relevance Determination prior

class svb.prior.ConstantMRFSpatialPrior (nvertices, mean, var, idx=None, nn=None,
                                         n2=None, **kwargs)
    Prior which performs adaptive spatial regularization based on the contents of neighbouring vertices using the
    Markov Random Field method

    This is equivalent to the Fabber ‘M’ type spatial prior

class svb.prior.FabberMRFSpatialPrior (nvertices, mean, var, idx=None, post=None,
                                         nn=None, n2=None, **kwargs)
    Prior designed to mimic the ‘M’ type spatial prior in Fabber.

    Note that this uses update equations for ak which is not in the spirit of the stochastic method. ‘Native’ SVB
    MRF spatial priors are also defined which simply treat the spatial precision parameter as an inference variable.

```

This code has been verified to generate the same ak estimate given the same input as Fabber, however in practice it does not optimize to the same value. We don't yet know why.

```
class svb.prior.FactorisedPrior (priors, **kwargs)
```

Prior for a collection of parameters where there is no prior covariance

In this case the mean log PDF can be summed from the contributions of each parameter

```
log_det_cov ()
```

Determinant of diagonal matrix is product of diagonal entries

```
mean_log_pdf (samples)
```

Parameters *samples* – A tensor of shape [W, P, S] where W is the number of parameter vertices, P is the number of parameters in the prior (possibly 1) and S is the number of samples

Returns A tensor of shape [W] where W is the number of parameter vertices containing the mean log PDF of the parameter samples provided

```
class svb.prior.MRF2SpatialPrior (nvertices, mean, var, idx=None, post=None, nn=None, n2=None, **kwargs)
```

Prior which performs adaptive spatial regularization based on the contents of neighbouring vertices using the Markov Random Field method

This uses the same formalism as the Fabber 'M' type spatial prior but treats the ak as a parameter of the optimization. It differs from MRFSpatialPrior by using the PDF formulation of the PDF rather than the matrix formulation (the two are equivalent but currently we keep both around for checking that they really are!)

FIXME currently this does not work unless sample size=1

```
mean_log_pdf (samples)
```

Parameters *samples* – A tensor of shape [W, P, S] where W is the number of parameter vertices, P is the number of parameters in the prior (possibly 1) and S is the number of samples

Returns A tensor of shape [W] where W is the number of parameter vertices containing the mean log PDF of the parameter samples provided

```
class svb.prior.MRFSpatialPrior (nvertices, mean, var, idx=None, post=None, nn=None, n2=None, **kwargs)
```

Prior which performs adaptive spatial regularization based on the contents of neighbouring vertices using the Markov Random Field method

This uses the same formalism as the Fabber 'M' type spatial prior but treats the ak as a parameter of the optimization.

```
mean_log_pdf (samples)
```

mean log PDF for the MRF spatial prior.

This is calculating:

$$\log P = \frac{1}{2} \log \phi - \frac{\phi}{2} \underline{x}^T D \underline{x}$$

```
class svb.prior.NormalPrior (nvertices, mean, var, **kwargs)
```

Prior based on a vertexwise univariate normal distribution

```
mean_log_pdf (samples)
```

Mean log PDF for normal distribution

Note that `term1` is a constant offset when the prior variance is fixed and hence in earlier versions of the code this was neglected, along with other constant offsets such as factors of pi. However when this code is inherited by spatial priors and ARD the variance is no longer fixed and this term must be included.

```
class svb.prior.Prior (**kwargs)
    Base class for a prior, defining methods that must be implemented

    mean_log_pdf (samples)

        Parameters samples – A tensor of shape [W, P, S] where W is the number of parameter
            vertices, P is the number of parameters in the prior (possibly 1) and S is the number of
            samples

        Returns A tensor of shape [W] where W is the number of parameter vertices containing the
            mean log PDF of the parameter samples provided

svb.prior.get_prior (param, data_model, **kwargs)
    Factory method to return a vertexwise prior
```

8.5 SVB module

Stochastic Bayesian inference of a nonlinear model

Infers:

- Posterior mean values of model parameters
- A posterior covariance matrix (which may be diagonal or a full positive-definite matrix)

The general order for tensor dimensions is:

- Voxel indexing (V=number of voxels / W=number of parameter vertices)
- Parameter indexing (P=number of parameters)
- Sample indexing (S=number of samples)
- Data point indexing (B=batch size, i.e. number of time points being trained on, in some cases T=total number of time points in full data)

This ordering is chosen to allow the use of TensorFlow batch matrix operations. However it is inconvenient for the model which would like to be able to index input by parameter. For this reason we transpose when calling the model's `evaluate` function to put the P dimension first.

The parameter vertices, W, are the set of points on which parameters are defined and will be output. They may be voxel centres, or surface element vertices. The data voxels, V, on the other hand are the points on which the data to be fitted to is defined. Typically this will be volumetric voxels as that is what most imaging experiments output as raw data.

In many cases, W will be the same as V since we are inferring volumetric parameter maps from volumetric data. However we might alternatively want to infer surface based parameter maps but keep the comparison to the measured volumetric data. In this case V and W will be different. The key point at which this difference is handled is the model evaluation which takes parameters defined on W and outputs a prediction defined on V.

V and W are currently identical but may not be in the future. For example we may want to estimate parameters on a surface (W=number of surface vertices) using data defined on a volume (V=number of voxels).

Ideas for per voxel/vertex convergence:

- Maintain `vertex_mask` as member. Initially all ones
- Mask vertices when generating samples and evaluating model. The latent cost will be over unmasked vertices only.
- **PROBLEM:** need reconstruction cost defined over full voxel set hence need to project model evaluation onto all voxels. So masked vertices still need to keep their previous model evaluation output

- Define criteria for masking vertices after each epoch
- **PROBLEM:** spatial interactions make per-voxel convergence difficult. Maybe only do full set convergence in this case (like Fabber)

class `svb.svb.SvbFit` (*data_model*, *fwd_model*, ***kwargs*)
 Stochastic Bayesian model fitting

Variables

- ***model*** – Model instance to be fitted to some data
- ***prior*** – `svb.prior.Prior` instance defining the prior parameter distribution
- ***post*** – `svb.posterior.Posterior` instance defining the posterior parameter distribution
- ***params*** – Sequence of `Parameter` instances of parameters to infer. This includes the model parameters and the noise parameter(s)

evaluate (**tensors*)

Evaluate tensor values

Parameters ***tensors*** – Sequence of tensors or names of tensors

Returns If single tensor requested, it's value as Numpy array. Otherwise tuple of Numpy arrays

fit_batch ()

Train model based on mini-batch of input data.

Returns Tuple of total cost of mini-batch, latent cost and reconstruction cost

set_state (*state*)

Set the state of the optimization

Parameters ***state*** – State as returned by the `state()` method

state ()

Get the current state of the optimization.

This can be used to restart from a previous state if a numerical error occurs

train (*tpts*, *data*, *batch_size=None*, *sequential_batches=False*, *epochs=100*, *fit_only_epochs=0*, *display_step=1*, *learning_rate=0.1*, *lr_decay_rate=1.0*, *sample_size=None*, *ss_increase_factor=1.0*, *revert_post_trials=50*, *revert_post_final=True*, ***kwargs*)

Train the graph to infer the posterior distribution given timeseries data

Parameters

- ***tpts*** – Time series values. Should have shape [T] or [V, T] depending on whether time-series is constant or varies voxelwise
- ***data*** – Full timeseries data, shape [V, T]

Optional arguments:

Parameters

- ***batch_size*** – Batch size to use when training model. Need not be a factor of T, however if not batches will not all be the same size. If not specified, data size is used (i.e. no mini-batch optimization)
- ***sequential_batches*** – If True, form batches from consecutive time points rather than strides
- ***epochs*** – Number of training epochs

- **fit_only_epochs** – If specified, this number of epochs will be restricted to fitting only and ignore prior information. In practice this means only the reconstruction loss is considered not the latent cost
- **display_step** – How many steps to execute for each display line
- **learning_rate** – Initial learning rate
- **lr_decay_rate** – When adjusting the learning rate, the factor to reduce it by
- **sample_size** – Number of samples to use when estimating expectations over the posterior
- **ss_increase_factor** – Factor to increase the sample size by over the epochs
- **revert_post_trials** – How many epoch to continue for without an improvement in the mean cost before reverting the posterior to the previous best parameters
- **revert_post_final** – If True, revert to the state giving the best cost achieved after the final epoch

8.6 Utils module

General utility functions

class `svb.utils.LogBase` (***kwargs*)

Base class that provides a named log and the ability to log tensors easily

log_tf (*tensor, level=10, **kwargs*)

Log a tensor

Parameters

- **tensor** – tf.Tensor
- **level** – Logging level (default: DEBUG)

Keyword arguments:

Parameters

- **summarize** – Number of entries to include (default 100)
- **force** – If True, always log this tensor regardless of log level
- **shape** – If True, precede tensor with its shape

`svb.utils.ValueList` (*value_type*)

Class used with argparse for options which can be given as a comma separated list

Stochastic Bayesian inference of a nonlinear model

Infers:

- Posterior mean values of model parameters
- A posterior covariance matrix (which may be diagonal or a full positive-definite matrix)

The general order for tensor dimensions is:

- Voxel indexing (V=number of voxels / W=number of parameter vertices)
- Parameter indexing (P=number of parameters)
- Sample indexing (S=number of samples)

- Data point indexing (B=batch size, i.e. number of time points being trained on, in some cases T=total number of time points in full data)

This ordering is chosen to allow the use of TensorFlow batch matrix operations. However it is inconvenient for the model which would like to be able to index input by parameter. For this reason we transpose when calling the model's `evaluate` function to put the P dimension first.

The parameter vertices, W , are the set of points on which parameters are defined and will be output. They may be voxel centres, or surface element vertices. The data voxels, V , on the other hand are the points on which the data to be fitted to is defined. Typically this will be volumetric voxels as that is what most imaging experiments output as raw data.

In many cases, W will be the same as V since we are inferring volumetric parameter maps from volumetric data. However we might alternatively want to infer surface based parameter maps but keep the comparison to the measured volumetric data. In this case V and W will be different. The key point at which this difference is handled is the model evaluation which takes parameters defined on W and outputs a prediction defined on V .

V and W are currently identical but may not be in the future. For example we may want to estimate parameters on a surface (W =number of surface vertices) using data defined on a volume (V =number of voxels).

Ideas for per voxel/vertex convergence:

- Maintain `vertex_mask` as member. Initially all ones
- Mask vertices when generating samples and evaluating model. The latent cost will be over unmasked vertices only.
- PROBLEM: need reconstruction cost defined over full voxel set hence need to project model evaluation onto all voxels. So masked vertices still need to keep their previous model evaluation output
- Define criteria for masking vertices after each epoch
- PROBLEM: spatial interactions make per-voxel convergence difficult. Maybe only do full set convergence in this case (like Fabber)

8.7 Indices and tables

- `genindex`
- `modindex`
- `search`

S

- `svb.model`, [62](#)
- `svb.parameter`, [63](#)
- `svb.posterior`, [63](#)
- `svb.prior`, [66](#)
- `svb.svb`, [68](#)
- `svb.utils`, [70](#)

A

ARDPrior (class in *svb.prior*), 66

C

ConstantMRFSpatialPrior (class in *svb.prior*), 66

E

entropy() (*svb.posterior.FactorisedPosterior* method), 64

entropy() (*svb.posterior.GaussianGlobalPosterior* method), 64

entropy() (*svb.posterior.MVNPosterior* method), 65

entropy() (*svb.posterior.NormalPosterior* method), 65

entropy() (*svb.posterior.Posterior* method), 66

evaluate() (*svb.model.Model* method), 62

evaluate() (*svb.svb.SvbFit* method), 69

F

FabberMRFSpatialPrior (class in *svb.prior*), 66

FactorisedPosterior (class in *svb.posterior*), 63

FactorisedPrior (class in *svb.prior*), 67

fit_batch() (*svb.svb.SvbFit* method), 69

G

GaussianGlobalPosterior (class in *svb.posterior*), 64

get_model_class() (in module *svb.model*), 63

get_parameter() (in module *svb.parameter*), 63

get_posterior() (in module *svb.posterior*), 66

get_prior() (in module *svb.prior*), 68

I

ievaluate() (*svb.model.Model* method), 62

L

latent_loss() (*svb.posterior.FactorisedPosterior* method), 64

log_config() (*svb.model.Model* method), 62

log_det_cov() (*svb.posterior.FactorisedPosterior* method), 64

log_det_cov() (*svb.posterior.MVNPosterior* method), 65

log_det_cov() (*svb.prior.FactorisedPrior* method), 67

log_tf() (*svb.utils.LogBase* method), 70

LogBase (class in *svb.utils*), 70

M

mean_log_pdf() (*svb.prior.FactorisedPrior* method), 67

mean_log_pdf() (*svb.prior.MRF2SpatialPrior* method), 67

mean_log_pdf() (*svb.prior.MRFSpatialPrior* method), 67

mean_log_pdf() (*svb.prior.NormalPrior* method), 67

mean_log_pdf() (*svb.prior.Prior* method), 68

Model (class in *svb.model*), 62

MRF2SpatialPrior (class in *svb.prior*), 67

MRFSpatialPrior (class in *svb.prior*), 67

MVNPosterior (class in *svb.posterior*), 65

N

NormalPosterior (class in *svb.posterior*), 65

NormalPrior (class in *svb.prior*), 67

nparams (*svb.model.Model* attribute), 62

P

param_idx() (*svb.model.Model* method), 62

Parameter (class in *svb.parameter*), 63

Posterior (class in *svb.posterior*), 66

Prior (class in *svb.prior*), 67

S

sample() (*svb.posterior.FactorisedPosterior* method), 64

sample() (*svb.posterior.GaussianGlobalPosterior* method), 64

`sample()` (*svb.posterior.MVNPosterior method*), 65
`sample()` (*svb.posterior.NormalPosterior method*), 65
`sample()` (*svb.posterior.Posterior method*), 66
`set_state()` (*svb.posterior.FactorisedPosterior method*), 64
`set_state()` (*svb.posterior.GaussianGlobalPosterior method*), 64
`set_state()` (*svb.posterior.MVNPosterior method*), 65
`set_state()` (*svb.posterior.NormalPosterior method*), 65
`set_state()` (*svb.posterior.Posterior method*), 66
`set_state()` (*svb.svb.SvbFit method*), 69
`state()` (*svb.posterior.FactorisedPosterior method*), 64
`state()` (*svb.posterior.GaussianGlobalPosterior method*), 64
`state()` (*svb.posterior.MVNPosterior method*), 65
`state()` (*svb.posterior.NormalPosterior method*), 65
`state()` (*svb.posterior.Posterior method*), 66
`state()` (*svb.svb.SvbFit method*), 69
`svb.model` (*module*), 62
`svb.parameter` (*module*), 63
`svb.posterior` (*module*), 63
`svb.prior` (*module*), 66
`svb.svb` (*module*), 68, 70
`svb.utils` (*module*), 70
`SvbFit` (*class in svb.svb*), 69

T

`test_data()` (*svb.model.Model method*), 63
`tpts()` (*svb.model.Model method*), 63
`train()` (*svb.svb.SvbFit method*), 69

V

`ValueList()` (*in module svb.utils*), 70