BayHunter v2.1

A Python tool for McMC transdimensional Bayesian inversion of receiver functions (RF) and surface wave dispersion (SWD)

Documentation and Tutorial

Jennifer Dreiling
Frederik Tilmann

August 10, 2020

Table of contents

Overview

Abstract

BayHunter is an open source Python tool to perform an McMC transdimensional Bayesian inversion of surface wave dispersion and/or receiver functions. The algorithm follows a data-driven strategy and solves for the velocity-depth structure, the number of layers, V_P/V_S ratio and noise parameters, i.e., data noise correlation and amplitude. The forward modeling codes are provided within the package, but are easily replaceable with own codes. It is also possible to add (completely different) data sets.

The BayWatch module can be used to live-stream the inversion while it is running: this makes it easy to see how each chain is exploring the parameter space, how the data fits and models change and in which direction the inversion progresses.

Available on GitHub: `https://github.com/jenndrei/BayHunter`

Citation

Dreiling, Jennifer; Tilmann, Frederik (2019): BayHunter – McMC transdimensional Bayesian inversion of receiver functions and surface wave dispersion. GFZ Data Services. `https://doi.org/10.5880/GFZ.2.4.2019.001`.

Application examples

- Dreiling et al. (2020): Crustal structure of Sri Lanka derived from joint inversion of surface wave dispersion and receiver functions using a Bayesian approach. Journal of Geophysical Research: Solid Earth. `https://doi.org/10.1029/2019JB018688`.

- Green et al. (2020): Magmatic and sedimentary structure beneath the Klyuchevskoy Volcanic Group, Kamchatka, from ambient noise tomography. Journal of Geophysical Research: Solid Earth. `https://doi.org/10.1029/2019JB018900`.

- Mauerberger et al. (n.a.): The multifaceted Scandinavian lithosphere imaged by surface waves and ambient noise. In preparation.

Comments and Feedback

BayHunter is ready to use. It is quick and efficient and I am happy with the performance. Still, there are always things that can be improved to make it even faster and more efficient, and user friendlier.

Although we tested the software with a variety of synthetic and real data, each data set is still unique and shows own characteristics. If you observe any unforeseen behavior, please share it with me to wipe out possible problems we haven't considered.

I am happy to share my experience with you and also if you share your thoughts with me. I am looking forward to your feedback.

Quickstart

Requirements

- matplotlib
- numpy
- pyPdf
- configobj
- zmq
- Cython

Installation

(compatible with Python 2 and 3)

```
git clone https://github.com/jenndrei/BayHunter.git
cd BayHunter
sudo python setup.py install
```

References

- SWD forward modeling is based on `SURF96` from `CPS` (Herrmann and Ammon, 2002). BayHunter uses a Python wrapper using `pysurf96`[1] and `SurfTomo`[2].
- RF forward modeling using `rfmini` from Joachim Saul, GFZ.
- Most influence offered the work from Bodin et al. (2012).

[1] https://github.com/miili/pysurf96
[2] https://github.com/caiweicaiwei/SurfTomo

1 Introduction

Bayesian inversion is becoming more and more popular for several years and it has many advantages compared to conventional optimization approaches. While other methods often are more constrained and favor one best model based on the least misfit, an inversion after Bayes theorem is based on the model's likelihood and results in probability distributions for each parameter of the model. The inversion result is represented by a collection of models (posterior distribution) that are consistent with the data and with the selected model priors. They image the solution space and disclose uncertainties and trade-offs of the model parameters.

No open-source tools were available that suited our purpose of a joint inversion of SWD and RF after Bayes using a Markov chain Monte Carlo (McMC) sampling algorithm, and solve for the velocity-depth structure, the number of layers, the noise parameters, and the average crustal V_P/V_S ratio. So it was natural to take care of this. We developed BayHunter to serve that purpose, and BayWatch, a module that allows to live-stream the inversion while it is running.

Contents. Chapter 1 describes Bayes theorem and the McMC inversion approaches. They set the foundation for BayHunter, an inversion framework for transdimensional Bayesian inversion, described in chapter 2. We test the inversion code using synthetic data and provide a tutorial in chapter 3. The appendix provides a minimalistic working example of the code.

Bayes theorem

Bayes theorem (Bayes, 1763) is based on the relationship between inverse conditional probabilities. Assuming observed data d_{obs} and a model m; the probability that we observe d_{obs} given m is $p(d_{obs}|m)$, and the probability for m given d_{obs} is $p(m|d_{obs})$. Both occurrences are also dependent on the probability that m or d_{obs} is given, i.e., $p(m)$ and $p(d_{obs})$.

The inverse conditional probability that both events occur, is given by:

$$p(m|d_{obs})p(d_{obs}) = p(d_{obs}|m)p(m) \tag{1.1}$$

As d_{obs} is known as the evidence, i.e., the measurements, Bayes theorem can be rewritten to:

$$p(m|d_{obs}) \propto p(d_{obs}|m)p(m) \tag{1.2}$$

$p(m|d_{obs})$ is the posterior distribution, $p(d_{obs}|m)$ is called the likelihood, and $p(m)$ is the prior probability distribution.

Markov chain Monte Carlo

Markov chain Monte Carlo describes a sampling algorithm for sampling from a probability distribution. This algorithm is a combination of Monte Carlo, a random sampling method, and

Markov chains, assuming a dependency between the current and the previous sample. The exact implementation of the algorithm is described in chapter 2.

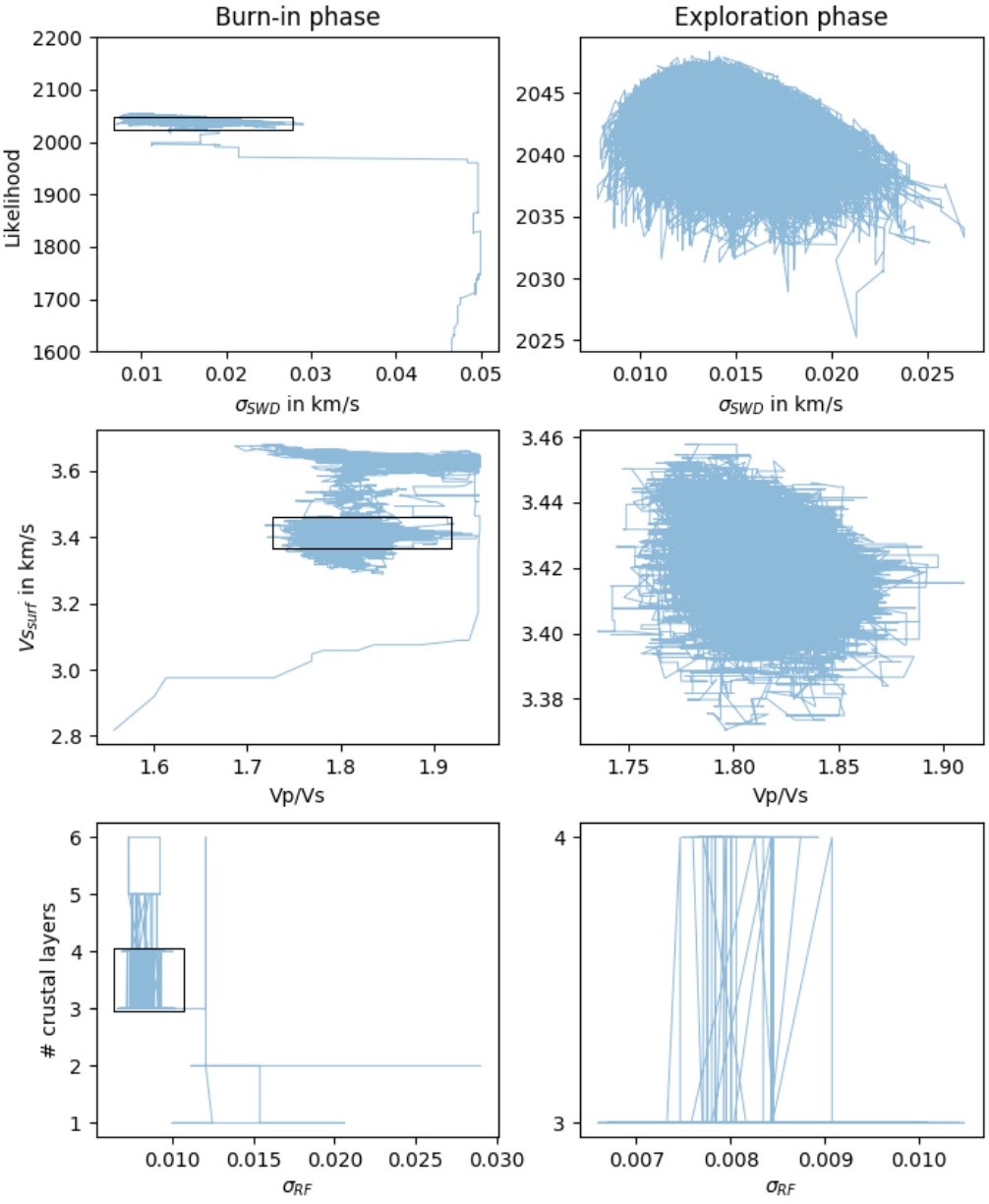

Figure 1.1: McMC sampling scheme for one chain progressing through the parameter space. Left and right columns show iterations of the burn-in (first phase) and exploration phase (second phase), respectively, with the black box framing the exploration region as shown in the right column. Illustrated parameters are the likelihood and noise amplitude σ_{SWD}, surface V_S and V_P/V_S, and number of crustal layers and noise amplitude σ_{RF}. The top panels reflect the optimization process based on the likelihood, while the lower panels show the parameter trade-off. Example taken from station SL10 in Sri Lanka.

Figure 1.1 shows a real data example from a station in Sri Lanka (SL10), following the evolution of one chain through the model parameter space. Parameters are shown in couples, but note that the parameter space is multidimensional. In the burn-in phase (left column) the chain starts at a random parameter combination in the solution space and progresses with ongoing

iterations towards an optimum, based on the likelihood. In the second phase (right column) the chain has already reached its exploration region and samples the posterior distribution.

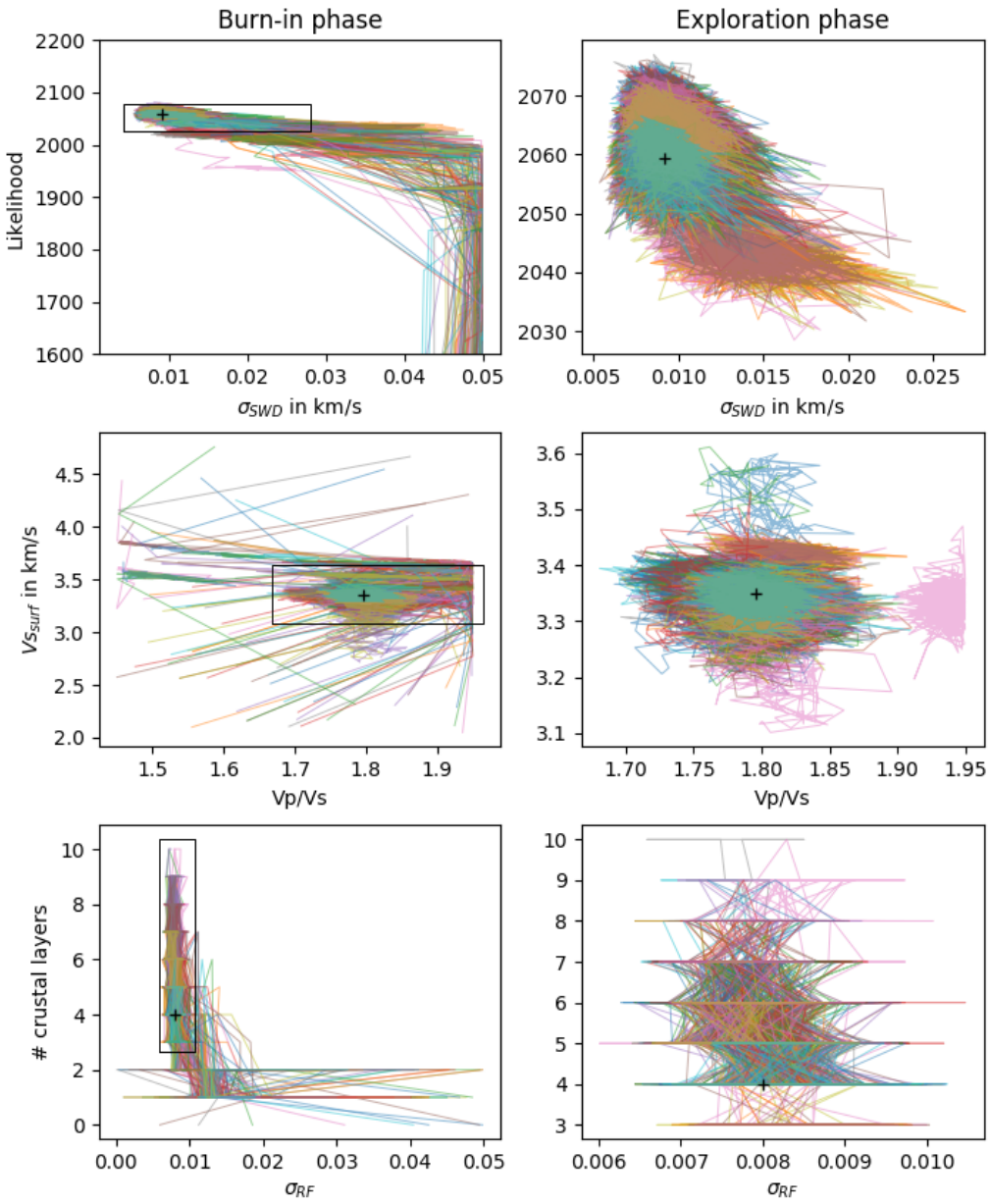

Figure 1.2: McMC sampling scheme for one hundred chains progressing independently through the parameter space. Same parameters and parameter space as illustrated in Figure 1.1. Black cross shows the median from the complete posterior distribution (exploration phase). Example taken from station SL10 in Sri Lanka.

Figure 1.2 shows one hundred independent chains exploring the same parameter space. Each chain starts with a different random model, yet most chains converge to the same exploration region for sampling the posterior distribution. There are still chains (e.g., the two rose colored ones in the middle panel) that have not converged into the optimum zone when entering the exploration phase. If those chains do not converge to that optimum zone within the second phase, they probably represent outlier chains (or secondary minima) and should not be considered for the posterior distribution.

2 Development of BayHunter

BayHunter is a tool to perform McMC transdimensional Bayesian inversion of SWD and RF, solving for the velocity-depth structure, the number of layers, noise scaling parameters (correlation, sigma), and average crustal V_P/V_S. The inversion algorithm uses multiple independent Markov chains and a random Monte Carlo sampling to find models with the highest likelihood.

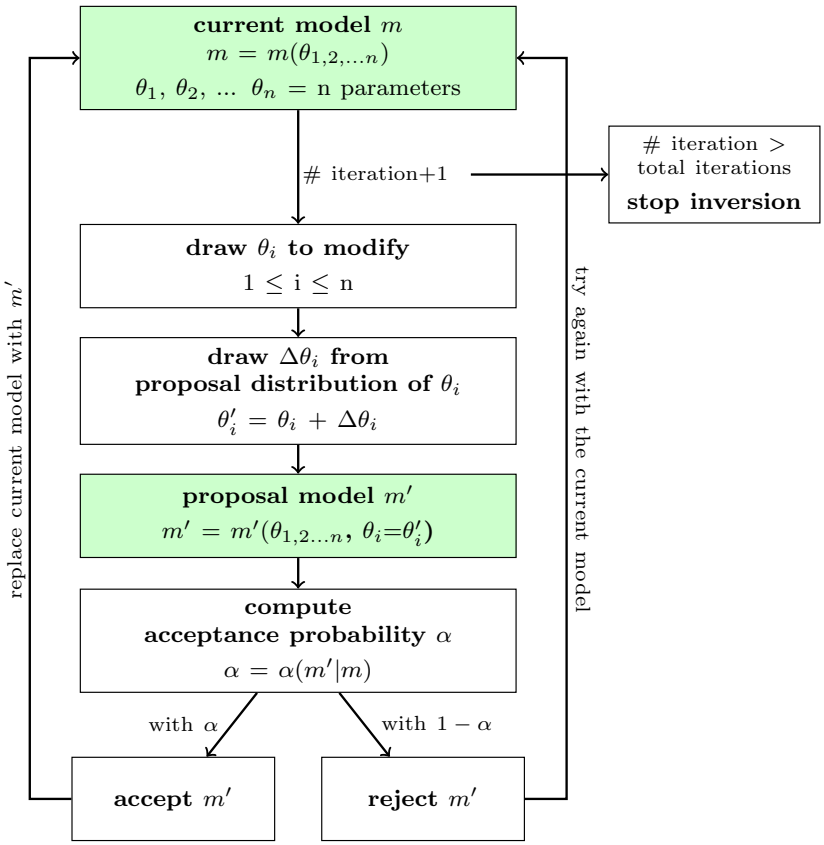

Figure 2.1: Schematic workflow of an McMC chain sampling the parameter space. The posterior distribution includes all accepted models of a chain after a chosen number of iterations.

How each chain is progressing through the parameter space is schematically illustrated in Figure 2.1. Each chain contains a current model. In each iteration a new model will be proposed, considering a proposal distribution, by modification of the current model. The acceptance probability is computed based on the prior, proposal and posterior ratios from proposal to current model. A proposed model gets accepted with a probability equal to the acceptance probability, i.e., if the likelihood of the proposed model is larger than the one of the current model, it gets accepted; but also models that are less likely than the current model get accepted with a small probability, which prevents the chain to get stuck in a local maximum. If a proposal model gets accepted, it will replace the current model; if not, the proposal model gets rejected and the current model stays unmodified. This process will be repeated for a defined number of iterations. Each accepted model of the exploration phase contributes to the posterior distribution of the parameters.

Equations given in section 2.1 are reduced to the number of those required in BayHunter, and will not be fully deduced. For mathematical derivations and details be referred to Bodin (2010) and Bodin et al. (2012), which inspired the idea of BayHunter and on which our algorithm is based on.

2.1 The Optimizer and Chain modules

The *BayHunter.mcmcOptimizer* manages the chains in an overarching module. It starts each single chain's inversion and schedules the parallel computing. Each chain and its complete model data can be accessed (in the Python environment) after the inversion has finished. Before the optimizer initializes the chains, a configuration file will automatically be saved, which simplifies the process of illustrating results after an inversion.

Each *BayHunter.SingleChain* explores the parameter space independently and collects samples by following Bayes theorem (Eq. 1.2). A chain has multiple tasks, which are described below in detail and begin with the random initialization of a starting model.

Initialize the targets. The first step towards an inversion is to define the target(s). Therefore, the user needs to pass the observed data to the designated BayHunter target type. For surface wave dispersion, period-velocity observables must be assigned to the class that handles the corresponding data (*RayleighDispersionPhase*, *RayleighDispersionGroup*, *LoveDispersionPhase*, *LoveDispersionGroup*). They default into the fundamental wave mode. Additionally, uncertainties can be attributed, which later control the weighting of each period. For receiver functions, observed time-amplitude data must be assigned to the *PReceiverFunction* (or *SReceiverFunction*) class from *BayHunter.Targets*. Parameters (Gauss filter width, slowness, water level, near surface velocity) for forward modeling should be updated, if the default values differ from the values used for RF computation.

Each of the target classes comes with a forward modeling plugin, which is easily exchangeable. For surface waves, a quick Fortran routine based on `CPS/surf96` (Herrmann and Ammon, 2002) is pre-installed. For receiver functions, the pre-installed routine is based on `rfmini` (Joachim Saul, GFZ). Also other targets can be defined.

Parametrization of the model. The model includes the velocity-depth structure and the noise parameters of the observed data.

- **Velocity-depth structure.** The velocity-depth model is parametrized through a variable number of Voronoi nuclei, the position of each is given by a depth and a seismic shear wave velocity (V_S). A model containing only one nucleus represents a half-space model, two nuclei define a model with one layer over a half-space and so on. The layer boundary (depth of an interface) lies equidistant between two nuclei. The advantage to use Voronoi nuclei over a simple thickness-velocity representation is, that one model can be parametrized in many different ways. However, a set of Voronoi nuclei defines only one model. The number of layers in a model is undefined and will be inverted for (transdimensional). V_P is computed from V_S through V_P/V_S, which is chosen by the user to be constant, or an additional parameter to solve for.

- **Covariance matrix of data noise.** The noise level is defined by two parameters, the correlation r (i.e., the correlation of adjacent data points) and the noise amplitude σ. Both r and σ are treated as unknown and can be estimated during the inversion. They are the part of the observed data that can not be fitted. Hence, the observed data vector can be described as

$$d_{obs}(i) = d_{True}(i) + \epsilon(i) \qquad i = [1, n] \tag{2.1}$$

where n is the size of the vector and $\epsilon(i)$ represents errors that are distributed according to a multivariate normal distribution with zero mean and covariance C_e.

$$C_e = \sigma^2 R \tag{2.2}$$

The covariance matrix C_e is dependent on σ and R, which is the symmetric diagonal-constant or Toeplitz matrix.

$$R = \begin{bmatrix} 1 & c_1 & c_2 & \dots & c_{n-1} \\ c_1 & 1 & c_1 & \dots & c_{n-2} \\ c_2 & c_1 & 1 & \dots & c_{n-3} \\ & & & \vdots & \\ c_{n-1} & c_{n-2} & c_{n-3} & \dots & 1 \end{bmatrix} \tag{2.3}$$

We consider two correlation laws for the correlation matrix R. The exponential law is given by

$$c_i = r^i \tag{2.4}$$

and the Gaussian law by

$$c_i = r^{(i^2)} \tag{2.5}$$

where $r = c_1$ is a constant number between 0 and 1. In BayHunter we consider the exponential correlation law for surface wave dispersion, and both the exponential and the Gaussian law for receiver functions.

Initialize a model. For each chain, the initial model parameters (starting model) are drawn from the uniform prior distributions. These are, for the velocity-depth structure, the distributions of V_S, depth, the number of layers and the average crustal V_P/V_S. The initial model has a number of layers equal to the minimum value of the corresponding prior distribution. If set to 0, a half-space model will be drawn, if set to 1 a one layer over a half-space model represents the initial model and so on. The initial number of layers determines how many Voronoi nuclei will be drawn, i.e., how many pairs of V_S and depth. If a velocity-depth model was drawn, V_P will be computed from V_P/V_S, which was either drawn hitherto or given as constant by the user. If appropriate, the user may wish to select a mantle specific V_P/V_S, by also assuming a V_S to distinguish the mantle from the crust. The density is computed by $\rho = 0.77 + 0.32V_P$ (Berteussen, 1977).

It is possible to give a single interface depth estimate (it can be any interface, e.g., the Moho). The estimate includes a mean and a standard deviation of the interface depth. When initializing a model - and only if an estimate was given - an interface depth is drawn from the given normal distribution and two nuclei will be placed equidistant to the interface. If the initial model only consists of a half-space, the interface estimate will be ignored. Giving an estimate can help the chains to converge more quickly, e.g., if computation capacity is limited, but might generate biased posterior distributions.

Each target has two noise scaling parameters (r, σ). The user needs to define the prior distributions for the overarching target type, i.e., SWD and RF, nevertheless, each target will sample an own posterior distribution. Single noise parameters can also be set constant during an inversion by giving a single digit, instead of a range. Initial values are then the given digits, and/or will be drawn from the prior range.

The drawn starting model automatically is assigned as the current model of the chain. The corresponding likelihood is computed. This model is also the first model in the model chain that gets collected for the burn-in phase.

Computation of model likelihood. The likelihood is an estimate of the probability of observing the measured data given a particular model m. It is an important measure for accepting and declining proposal models. The likelihood function is

$$p(d_{obs}|m) = \frac{1}{\sqrt{(2\pi)^n |C_e|}} \times exp\left\{\frac{-\Phi(m)}{2}\right\} \tag{2.6}$$

where $\Phi(m)$ is the Mahalanobis distance (Mahalanobis, 1936), i.e., the multidimensional distance between observed d_{obs} and estimated $g(m)$ data vectors.

$$\Phi(m) = (g(m) - d_{obs})^T C_e^{-1} (g(m) - d_{obs}) \tag{2.7}$$

As the likelihood often results in very small numbers, the log-likelihood is preferred.

$$\log p(d_{obs}|m) = -\frac{n}{2}\log(2\pi) - \frac{1}{2}\log(|C_e|) - \frac{\Phi(m)}{2} \tag{2.8}$$

The computation of the likelihood needs the inverse C_e^{-1} and determinant $|C_e|$ of the covariance matrix. For the exponential correlation law (Eq. 2.4), the covariance matrix is described by

$$C_e = \sigma^2 \begin{bmatrix} 1 & r & r^2 & \dots & r^{n-1} \\ r & 1 & r & \dots & r^{n-2} \\ r^2 & r & 1 & \dots & r^{n-3} \\ & & & \vdots & \\ r^{n-1} & r^{n-2} & r^{n-3} & \dots & 1 \end{bmatrix} \tag{2.9}$$

C_e^{-1} and $|C_e|$ can be solved through linear algebra, and are given by the analytical forms

$$C_e^{-1} = \frac{1}{\sigma^2(1-r^2)} \begin{bmatrix} 1 & -r & 0 & \dots & 0 & 0 \\ -r & 1+r^2 & -r & \dots & 0 & 0 \\ 0 & -r & 1+r^2 & \dots & 0 & 0 \\ & & & \vdots & & \\ 0 & 0 & 0 & \dots & 1+r^2 & -r \\ 0 & 0 & 0 & \dots & -r & 1 \end{bmatrix} \tag{2.10}$$

and

$$|C_e| = \sigma^{2n}(1-r^2)^{n-1} \tag{2.11}$$

These can be easily constructed and quickly computed in the Python language. Obviously, if the correlation of noise $r = 0$, the matrices R (Eq. 2.3) and R^{-1} are simply formed by the diagonal matrix, and the determinant is given by σ^{2n}. This is the default case for surface wave dispersion. If the dispersion measurements were assigned uncertainty values, then σ^2 is weighted by the relative uncertainties.

For receiver functions, unless they are computed utilizing an exponential filter, the Gaussian correlation law (Eq. 2.5) should be considered. In this case C_e^{-1} and $|C_e|$ cannot be solved analytically and the numerical computation of these is necessary. Considering a numerical computation each time a noise parameter is perturbed will increase the computation time tremendously. A trick to speed up the computation is accompanied by estimating r priorly and keeping it constant during the inversion. The equations

$$C_e^{-1} = (\sigma^2 R)^{-1} = \frac{1}{\sigma^2} R^{-1} \tag{2.12}$$

and

$$|C_e| = |\sigma^2 R| = \sigma^{2n}|R| \tag{2.13}$$

show, that R^{-1} and $|R|$ can be isolated from σ. Therefore, the numerical computations of R, R^{-1} and $|R|$ will be executed only once at the beginning of the inversion. R^{-1} and $|R|$ will be multiplied by the σ-terms and used in equations 2.7 and 2.8 to compute the likelihood. The correlation parameter r in R needs to be chosen by the user, but can be estimated. If r is set too large, R^{-1} becomes instable and small eigenvalues need to be suppressed.

The likelihood for inversions of multiple data sets is computed by the sum of the log-likelihoods from different targets.

Propose a model. At each iteration a new model is proposed using one of six modification methods. The method is drawn randomly and the current model will be modified according to the method's proposal distribution. Either a parameter is modified (V_S or depth of Voronoi nucleus, V_P/V_S, r, σ) or the dimension of parameters, i.e., the number of layers in the velocity-depth structure (layer birth, death). The methods are summarized below.

(1) Modification of V_S (Voronoi nucleus)
(2) Modification of depth (Voronoi nucleus)
(3) Modification of crustal V_P/V_S
(4) Modification of a noise parameter (r, σ)
(5) Modification of dimension (layer birth)
(6) Modification of dimension (layer death)

Each method except (4) is altering the velocity-depth structure. For (1) and (2), a random Voronoi nucleus from the current model is selected. For (1), the V_S of the nucleus is modified according to the proposal distribution of V_S. Therefore, a sample from this normal distribution (centered at zero) is drawn and added to the current V_S value of the nucleus. For (2), a sample from the depth proposal distribution is drawn and added to the depth-coordinate of the nucleus. For (3), if not constant, a sample from the V_P/V_S proposal distribution is drawn and added to the V_P/V_S value of the current model. For (4), one random noise parameter from one target is selected (r or σ). This parameter, if not constant, is modified according to the procedure before and according to its own proposal distribution. C_e assumed for surface wave dispersion is based on the exponential law. For receiver functions, the exponential law is only assumed, if the user wants to invert for r. If r is given by a constant, automatically the Gaussian correlation law is considered.

For (5) and (6), the proposal distributions are equal. For (5), a random depth-value will be drawn from the uniform depth prior distribution, where a new Voronoi nucleus will be born. The new velocity of this nucleus will be computed by the current model velocity at the drawn depth, modified by the proposal distribution. For (6), a random nucleus from the nuclei ensemble of the current model is chosen and removed. Here, the proposal distribution is only relevant for the computation of the acceptance probability, not for the actual modification of the model. Note that the proposal distributions for (5) and (6) relate to V_S.

For the six modification methods, the user must define five normal distributions as initial proposal distributions by giving their standard deviations. For the model modification methods (1)-(4), it is obvious, that small standard deviations of the distributions cause a high chance of only small parameter changes. So, the proposal models are very similar to the current model. On the other hand, if the proposal distribution width is large, the modifications tend to be larger and the proposal models are likely to be more different from the current model. For (5) and (6) however, the proposal distribution only plays a subordinate role. If a random nucleus is added or removed, the complete model structure between the adjacent nuclei is modified, which can cause large interface depth shifts – dependent on the proximity of the adjacent nuclei. A nucleus birth with a V_S modification of zero would still result in a shift of the layer interface.

The initial width of the proposal distribution, however, will be adjusted during the inversion to reach and maintain a specific acceptance rate of proposal models (see section 3.2).

As a feature in BayHunter, dimension modifications are disallowed in the first 1 % of the iterations. This enables a first simple approximation of the V_S-depth structure, before turning into more complex models by allowing layer birth and death. This is especially important if the inversion is only constrained by surface wave dispersion data.

Accept a model. After a model is proposed, it needs to be evaluated in comparison to the current model. Therefore, the acceptance probability α is computed. If any parameter of the proposed model does not lie within its prior distribution, the acceptance probability drops to zero and the model will automatically be declined. A model parameter can only lie outside the prior if the current value of it is very close to the prior limits or its proposal distribution width is very large. Further criteria that will force a refusal of the proposal model by setting $\alpha = 0$:

- a layer thickness is smaller than *thickmin*
- a low / high velocity zone does not fulfill the user defined constraint

If a model proposal clears the above criteria, the actual acceptance probability α is computed. The acceptance probability is a combined probability and will be computed from the prior, proposal and likelihood ratios of the proposal model m' and the current model m.

$$\alpha = \text{prior ratio x likelihood ratio x proposal ratio x Jacobian}$$

$$\alpha(m'|m) = \frac{p(m')}{p(m)} \times \frac{p(d_{obs}|m')}{p(d_{obs}|m)} \times \frac{q(m|m')}{q(m'|m)} \times |J| \tag{2.14}$$

The determinant of the Jacobian matrix equals 1 in any case of modification. Furthermore, the acceptance term can be rearranged dependent on the type of model modification.

- **Voronoi nucleus position, V_P/V_S, and covariance matrix.** The modification of the nucleus position (i.e., V_S and depth), V_P/V_S and the covariance matrix (i.e., r and σ) do not involve a change of dimension. For these model proposals, the prior ratio equals 1 and the proposal distributions are symmetrical, i.e., the probability to go from m to m' is equal to the probability to go from m' to m. Hence, the proposal ratio also equals 1. We can shorten the acceptance probability to the likelihood ratio:

$$\alpha(m'|m) = \frac{p(d_{obs}|m')}{p(d_{obs}|m)} \tag{2.15}$$

If the Voronoi nucleus position or V_P/V_S was modified, the factor $\frac{1}{\sqrt{(2\pi)^n |C_e|}}$ in the likelihood function (Eq. 2.6) is equal for proposed and current model and cancels out. Thus, α is only dependent on the Mahalanobis distance and is defined as:

$$\alpha(m'|m) = exp\left\{-\frac{\Phi(m') - \Phi(m)}{2}\right\} \tag{2.16}$$

If a noise parameter was modified, C_e are different for proposal and current model; therefore the mentioned factor in the likelihood function must be included in the computation of α. Note that the Mahalanobis distance also includes the covariance matrix C_e. The acceptance probability computes as follows:

$$\alpha(m'|m) = \left(\frac{|C_e|}{|C'_e|}\right)^{\frac{1}{2}} \times exp\left\{-\frac{\Phi(m') - \Phi(m)}{2}\right\} \tag{2.17}$$

- **Dimension change of velocity-depth model.** A dimension change of a model implies the birth or death of a Voronoi nucleus, which corresponds to a layer birth or death. In this case,

the prior and proposal ratios are no longer unity. For a birth step, the acceptance probability equals:

$$\alpha(m'|m) = \frac{\theta\sqrt{2\pi}}{\Delta v} \times exp\left\{\frac{(v'_{k+1} - v_i)^2}{2\theta^2} - \frac{\Phi(m') - \Phi(m)}{2}\right\} \tag{2.18}$$

where i indicates the layer in the current Voronoi tessellation c that contains the depth c'_{k+1} where the birth takes place. v_i and v'_{k+1} are the velocities at given depth of the current and the proposal model, i.e., before and after the birth. θ is the standard deviation of the proposal distribution for a dimension change. The acceptance probability of the birth step is a balance between the proposal probability (which encourages velocities to change) and the difference in data misfit which penalizes velocities if they change so much that they degrade the fit to the data.

The acceptance probability for a death of a Voronoi nucleus is:

$$\alpha(m'|m) = \frac{\Delta v}{\theta\sqrt{2\pi}} \times exp\left\{-\frac{(v'_j - v_i)^2}{2\theta^2} - \frac{\Phi(m') - \Phi(m)}{2}\right\} \tag{2.19}$$

where i indicates the layer that was removed from the current tessellation c and j indicates the cell in the proposed Voronoi tessellation c' that contains the deleted point c_i. v_i and v'_j are corresponding velocities.

The proposal candidate will be accepted with a probability of α, or rejected with a probability of $1 - \alpha$. The computational implementation is a comparison of α to a number u, which is randomly drawn from a uniform distribution between 0–1. The model is accepted if $\alpha > u$, which is always the case if $\alpha > 1$. As we consider the log-space for our computations, we use $\log(\alpha)$ and $\log(u)$.

When a proposal model is accepted, it will replace the current model. On the other hand, when a model is rejected, the current model stays unmodified. In the next iteration, a new model is proposed. This process will be repeated until the defined number of iterations is reached. The accepted models form the Markov chain and define the posterior distribution of the parameters after the burn-in phase.

The posterior distribution. After a chain has finished its iterations, it automatically saves ten output files in `.npy` format (NumPy binary file), holding V_S-depth models, noise parameters, V_P/V_S ratios, likelihoods and misfits for the burn-in (p1) and the posterior sampling phase (p2), respectively. Every i-th chain model is saved to receive a p2-model collection of $\sim$ *maxmodels*, a constraint given by the user. The files are saved in *savepath/data* as follows:

`c*_p1models.npy`	`c*_p2models.npy`	
`c*_p1noise.npy`	`c*_p2noise.npy`	
`c*_p1vpvs.npy`	`c*_p2vpvs.npy`	
`c*_p1likes.npy`	`c*_p2likes.npy`	*three-digit chain identifier number
`c*_p1misfits.npy`	`c*_p2misfits.npy`	

While V_P/V_S and the likelihood are vectors with the lengths defined by *maxmodels*, the models, noise and misfit values are represented by matrices, additionally dependent on the maximum number of model layers and the number of targets, both also defined by the user. The models are saved as Voronoi nuclei representation. For noise parameters, the matrix contains r and σ of each target. For the RMS data misfit, the matrix is composed of the misfit from each target and the joint misfit.

2.2 The Saving and Plotting modules

The *BayHunter.Plotting* module cannot only be utilized for data illustration, but also for outlier detection and re-saving of *BayHunter.SingleChain* results.

Outlier detection. Not every chain converges to the optimum solution space. BayHunter provides a method for outlier chain detection based on the median likelihood. For each chain, the median likelihood of the exploration phase is computed. A threshold is computed below which chains are declared as outliers. The threshold is a percentage of the maximum reached median likelihood from the chain ensemble. The percentage is defined by the user in terms of deviation from the maximum likelihood. For instance, if the deviation *dev*=0.05 (5 %), all chains not reaching a median likelihood of 95 % of the maximum median likelihood, are declared as outlier chains. If no or another outlier detection method is preferred, the user may chose a large value for *dev*, e.g., *dev*=5. The chain identifiers of the outlier chains will be saved to a file, which will be overwritten when repeating outlier detection.

Final posterior distribution. The *BayHunter.PlotFromStorage* class provides a method called *save_final_distribution*, which can be used to combine *BayHunter.SingleChain* results and store final posterior distribution files. Therefore, two arguments need to be chosen. The deviation *dev* is considered for outlier detection. *maxmodels* is the number of models that define the final posterior distribution. An equal number of p2-models per chain (except outlier chains) is chosen to assemble the posterior distribution of the inversion. Five files will be saved in `savepath/data` and represent the V_S-depth models, noise parameters, V_P/V_S ratios, likelihoods and misfits, respectively. The filename contains neither a chain identifier nor a phase tag and is e.g., for the V_P/V_S ratios: `c_vpvs.npy`.

Plotting methods. The plotting methods utilize the configuration file that was stored by the Optimizer module after initiation of the inversion. A list of plotting methods is presented below. Plots generated by these methods can be found in section 3.3.

plot_iiter*	* likes, nlayers, noise, vpvs, misfits	parameter with iterations (Fig. 3.2)
plot_posterior_*	* likes, nlayers, noise, vpvs, misfits, models1d, models2d	parameter posterior distribution or V_S-depth models (Fig. 3.4)
plot_current*, plot_best*	* datafits, models	data fits or V_S-depth models from current or likeliest models (Fig. 3.3)
plot_refmodel	add reference model to posterior distributions (Figs. 3.3, 3.4)	

2.3 The BayWatch module

During a BayHunter inversion the user can live-stream progress and results with the BayWatch graphical interface. This makes it easy to see how chains explore the parameter space, how the data fits and models change, in which direction the inversion progresses and if it is necessary to adjust parameters or prior settings. If the user sets *baywatch=True* in the inversion start command, BayHunter spawns a process only for streaming out the latest chain models. When starting BayWatch, those models are received and temporarily stored in memory, and will be visualized as shown in the screen shot (Fig. 2.2).

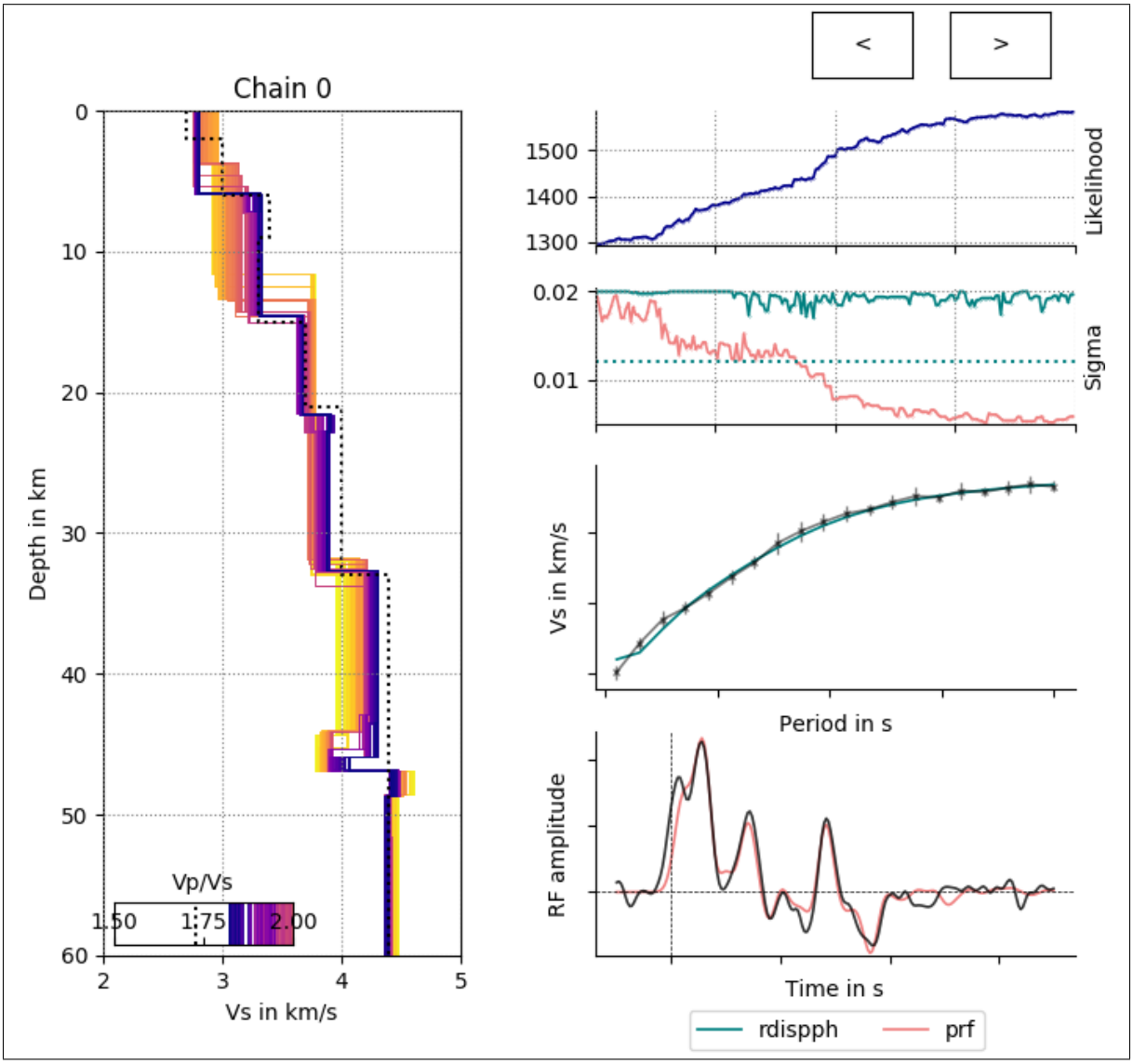

Figure 2.2: Screen shot of BayWatch live-stream showing the evolution of chain models with likelihood, i.e., the evolution of the V_S-depth structure, V_P/V_S (with the darkest colored model being the current model) and σ (sigma) for the two targets. The live-stream shows an inversion of synthetic data from a six-layer velocity-depth model as described in section 3.3. The Colored dotted lines represent the "true" model values.

3 Tutorial

This chapter contains the installation instructions of BayHunter, followed by an example of how to set up and run an inversion. A minimalistic working example is shown in the appendix. Furthermore, results from an inversion using synthetic data are shown and discussed. For the full tutorial including data files be referred to the GitHub repository[1].

3.1 Requirements and installation

BayHunter is compatible with Python 2 and Python 3. After installation of the required Python-modules, simply type the following to install BayHunter:

```
sudo python setup.py install
```

Required Python-modules and usage	
`numpy`	numerical computations
`matplotlib`	plotting library
`pyPdf`	merging PDFs
`configobj`	configuration file
`zmq`	BayWatch, inversion live-streaming
`Cython`	C-extensions for Python

The forward modeling codes for surface wave dispersion and receiver functions are already included in the BayHunter package and will be compiled when installing BayHunter. BayHunter uses a Python wrapper interfacing the `CPS/surf96` routine from Herrmann and Ammon (2002), and `rfmini` developed for BayHunter by Joachim Saul (GFZ).

3.2 Setting up and running an inversion

Setting up the targets. As mentioned in section 2.1 (Initialize the targets), BayHunter provides six target classes (four SWD and two RF), which use two types of forward modeling plugins (`CPS/surf96`, `rfmini`). For both targets, the user may update the default forward modeling parameters with *set_modelparams* (see appendix). Parameters and default values are given in Table 3.1.

If the user wants to implement own forward modeling code, a new forward modeling class for it is needed. After normally initializing a target with BayHunter, an instance of the new forward modeling class must be initialized and passed to the *update_plugin* method of the target. If an additional data set is wished to be included in the inversion, i.e., from a non pre-defined target class, a new target class needs to be implemented, additionally to the forward modeling class that handles the synthetic data computation. For both, the forward modeling class and

[1] https://github.com/jenndrei/BayHunter

Table 3.1: Default forward modeling parameters for SWD and RF.

SWD	mode = 1	1=fundamental mode, 2= 1st higher mode, etc.
RF	gauss = 1.0	Gauss factor, low pass filter
	water = 0.001	water level stabilization
	p = 6.4	slowness in deg/s
	nsv = None	near surface velocity in km/s for computation of incident angle (trace rotation). If `None`, nsv is taken from velocity-model.

the new target class, a template is stored on the GitHub repository. It is important that the classes implement specifically named methods and parameters to ensure the correct interface with BayHunter.

Setting up parameters. Each chain will be initialized with the targets and with parameter dictionaries. The model priors and inversion parameters that need to be defined are listed with default values in Table 3.2, and are explained below in detail.

Table 3.2: Default model priors and inversion parameters. Model prior tuples define the limits (min, max) of a uniform distribution. `None` implies that the constraint is not used. SI indicates corresponding units of the international system. Abbreviations and constraints are explained in the text.

Model priors			**Further parameters**		
vs	= (1, 5)	km/s	nchains	= 3	
z	= (0, 60)	km	$iter_{burnin}$	= 4096	
layers	= (1, 20)		$iter_{main}$	= 2048	
vpvs	= (1.5, 2.1)		acceptance	= (40, 45)	%
mantle[1]	= `None`	(km/s, -)	propdist[3]	= (0.015, 0.015, 0.015,	
mohoest[2]	= `None`	(km, km)		0.005, 0.005)	SI
r_{RF}	= (0.35, 0.75)		thickmin	= 0.	km
σ_{RF}	= (1e-5, 0.05)		lvz	= `None`	
r_{SWD}	= 0.		hvz	= `None`	
σ_{SWD}	= (1e-5, 0.1)	km/s	rcond	= `None`	
[1]i.e., (vs_m, $vpvs_m$), e.g., (4.2, 1.8)			station	= 'test'	
[2]i.e., (z_{mean}, z_{std}), e.g., (40, 4)			savepath	= 'results/'	
[3]i.e., (vs, z_{move}, $vs_{birth/death}$, noise, vpvs)			maxmodels	= 50 000	

The priors for the velocity-depth structure include V_S and depth, the number of layers, and average crustal V_P/V_S. The ranges as given in Table 3.2 indicate the bounds of uniform distributions. V_P/V_S can also be given as a float digit (e.g., 1.73), indicating a constant value during the inversion. The parameter *layers* does not include the underlying half space, which is always added to the model. A mantle condition (vs_m, $vpvs_m$), i.e., a vs_m threshold beyond which V_P is computed from $vpvs_m$, can be chosen if appropriate. There is also the option to give a single interface depth estimate through *mohoest*. It can be any interface, but the initial

idea behind was to give a Moho estimate. As explained in section 2.1 (Initialize a model), this should only be considered for testing purposes. Each noise scaling parameter (r, σ) can be given by a range or a digit, corresponding to the bounds of a uniform distribution (the parameter is inverted for) or a constant value (unaltered during the inversion), repectively.

For surface waves, the exponential correlation law (Eq. 2.4) is a realistic estimate of the correlation between data points and is automatically applied. For receiver functions, the assumed correlation law should be Gaussian (Eq. 2.5), if the RFs are computed using a Gaussian filter, and exponential, if the RFs are computed applying an exponential filter. The inversion for r_{RF} is viable for the latter, however, not for the Gaussian correlation law as of computational reasons (section 2.1, Computation of model likelihood). Only if r_{RF} is estimated by giving a single digit, the Gaussian correlation law is considered. Otherwise, if given a range for r_{RF}, the exponential correlation law is used. Note that the estimation of r_{RF} using the exponential law during an inversion, may not lead to correct results if the input RF was Gaussian filtered.

Nevertheless, r_{RF} can be estimated, as it is dependent on the sampling rate and the applied Gaussian filter width. Figure 3.1 shows an application of the BayHunter implemented tool to estimate r_{RF}. You will find a minimalistic code example in the appendix and an executable file with plenty of comments in the tutorial folder of the repository.

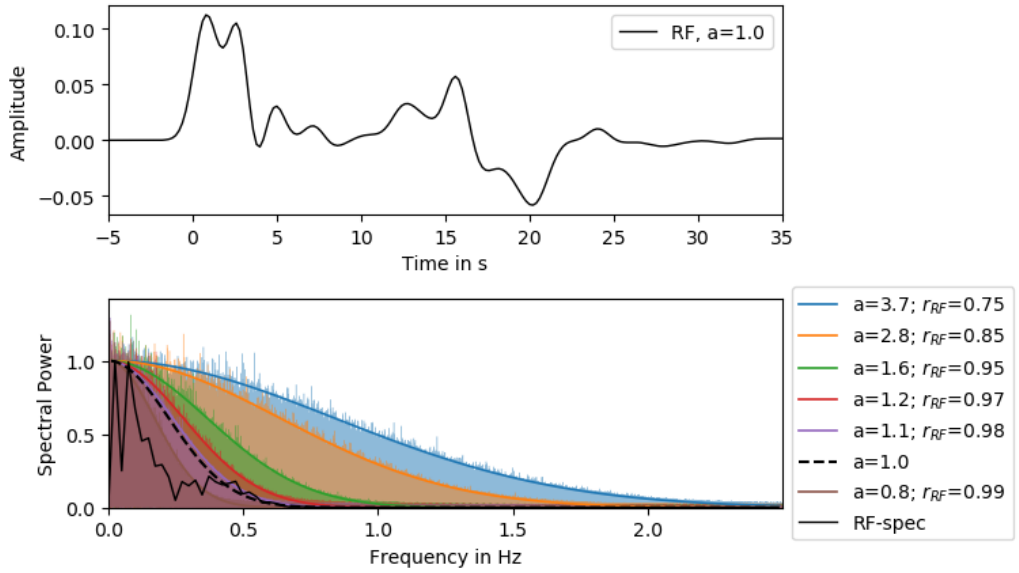

Figure 3.1: Visual estimation of r_{RF}. Top: Synthetic receiver function from a 3-layer crustal model, applying a Gaussian low pass filter with the Gaussian factor a=1. Bottom: Frequency spectrum of synthetic receiver function (solid black) and Gaussian filter with a=1 (dashed black). Transparently colored areas correspond to the spectra of large sample draws of synthetic Gaussian correlated noise using different values of r_{RF}. The solid colored lines represent the Gaussian curves matching the data 'envelope'. The legend displays corresponding r_{RF} and a values. For the given receiver function, a proper estimate of r_{RF} is 0.98.

If r_{RF} is too large (i.e., very close to 1), R^{-1} becomes instable and small eigenvalues need to be suppressed. The user can define the cutoff for small singular values by defining *rcond*. Singular values smaller than *rcond* x the largest singular value (both in modulus) are set to zero. *rcond* is not ascribed to the prior dictionary, but to the inversion parameter dictionary

(see configuration file).

The inversion parameters can be subdivided into three categories: (1) actual inversion parameters, (2) model constraints and (3) saving options. Parameters to constrain the inversion are the number of chains, the number of iterations for the burn-in and the main phase, the initial proposal distribution widths, and the acceptance rate. A large number of chains is preferable and assures good coverage of the solution space sampling, as each chain starts with a randomly drawn model only bound by the priors. The number of iterations should also be set high, as it can benefit, but not guarantee, the convergence of the chain towards the global likelihood maximum. The total amount of iterations is $iter_{total} = iter_{burnin} + iter_{main}$. We recommend to increase the ratio towards the iterations in the burn-in phase (i.e., $iter_{burnin} > iter_{main}$), so a chain is more likely to have converged when entering the exploration phase for the posterior distribution.

The initial proposal distributions, i.e., Gaussian distributions centered at zero, for model modifications, must be given as standard deviations according to each of the model modification methods (section 2.1, Propose a model). The values must be given as a vector of size five, the order representing following modifications: (1) V_S, (2) depth, (3) birth/death, (4) noise, and (5) V_P/V_S. The first three distributions represent V_S-depth model modifications referring to alterations of V_S (1,3) and z (2) of a Voronoi nucleus. There is one proposal distribution for both noise parameters r and σ (4) and one for V_P/V_S (5).

If the proposal distributions were constant, the percentage of accepted proposal models would decrease with ongoing inversion progress, i.e., the acceptance rate decreases at the expense of an efficient sampling. To efficiently sample the parameter space, an acceptance rate of ∼40–45 % is forced for each proposal method by dynamically adapting the width of each proposal distribution. We implemented a minimum standard deviation of 0.001 for each proposal distribution.

The most accepted model modifications are (1) and (2); their acceptance rates get easily forced to the desired percentage without even coming close to the defined minimum width of a proposal distribution. Birth and death steps, however, barely get accepted after an initial phase of high acceptance; if not limiting the proposal distribution width to a minimum, the standard deviations for (3) will get as small as 10^{-10} km/s and smaller to try to keep the acceptance rate up. However, as discussed in 2.1 (Propose a model), the distribution width does not in the first place influence the model-modification, but the added or removed Voronoi nucleus. Models modified by birth and death steps will naturally not be accepted very often and even less the further the inversion progresses. Therefore, the overall acceptance rate is stuck with a specific level below the forced rate. An estimate of the actual overall acceptance rate can be made, assuming a realistic acceptance for the birth and death steps, e.g., 1 %. A user given target rate of 40 % for each method would give an actual overall acceptance rate of ∼3 %. (→ 6 methods, 4 reach 40 %, 2 only 1 % = 30 % over all.) The target acceptance rate must be given as an interval.

There are three additional conditions, which might be worthwhile to use to constrain the velocity-depth model. However, using any of them could bias the posterior distribution. The user is allowed to set a minimum thickness of layers. Furthermore low and high velocity zones

can be suppressed. If not `None`, the value for *lvz* (or *hvz*) indicates the percentage of allowed V_S decrease (or increase) from each layer of a model relative to the underlying layer. For instance, if *lvz*=0.1, then a drop of V_S by 10 %, but not more, to the underlying layer is allowed. As V_S naturally increases with depth, and the algorithm only compares each layer with the layer underneath, the *hvz* criteria should only be used if observing extreme high velocity zones in the output. Otherwise sharp (but real) discontinuities could be smoothed out, if chosen too small. The *lvz* and *hvz* criteria will be checked every time a velocity-depth model is proposed and the model will be rejected if the constraints are not fulfilled.

The saving parameters include the *station*, *savepath* and *maxmodels*. The *station* name is optional and is only used as a reference for the user, for the automatically saved configuration file after initiation of an inversion. *savepath* represents the path where all the result files will be stored. A subfolder *data* will contain the configuration file and all the *SingleChain* output files, the combined posterior distribution files and an outlier information file. *savepath* also serves as figure directory. *maxmodels* is the number of p2-models that will be stored from each chain.

Running an inversion. The inversion will start through the *optimizer.mp_inversion* command with the option to chose the number of threads, *nthreads*, for parallel computing. By default, *nthreads* is equal to the number of CPUs of the user's PC. One thread is occupied if using BayWatch. Ideally, one chain is working on one thread. If fully exhausting the capacity of a 8 CPUs PC, give *nthreads*=8 and *nchains*=multiple of *nthreads* or (*nthreads*-1) if using BayWatch. This would cause *nthreads*(-1) chains to run parallel at all times, until *nchains* are worked off.

The speed of the inversion will not increase by choosing a larger *nthreads*. In fact, the speed is determined by the number of CPUs. If, for instance, the user doubles *nthreads*, the number of chains running parallel at once is also double, but the chains queue for some non-threadable computations blocking one CPU at a time, so each chain runs half the speed. To decrease *nthreads* offers a possibility to minimize the workload for a PC and that it is still accessible for other tasks during an inversion.

Although having access to a cluster, inversions were also performed on a single work station to determine the duration of an inversion with standard PC equipment (e.g., Memory: 16 GB, Processor model: 3.60 GHz x 8 cores). The runtime is not only dependent on the PC model, but also on the number of chains and iterations, and the number of layers of the actual velocity-depth structures, which directly influences the computational time of the forward modeling. The inversion for the example given in section 3.3 with 21 chains, 150,000 iterations and models with 3–10 layers, took 20.4 minutes; so each batch of 7 chains took 7 minutes.

Another argument to set when starting an inversion is *baywatch*. If set to True, model data will be send out with an interval of *dtsend*=0.5 s and can be received by BayWatch until the inversion has finished.

3.3 Testing with synthetic data

A set of test data was computed with the *BayHunter.SynthObs* module, which provides methods for computing receiver functions (P, S), surface wave dispersion curves (Love, Rayleigh, phase, group), and synthetic noise following the exponential or the Gaussian correlation law. We computed the P-RF and the fundamental mode SWD of the Rayleigh wave phase velocity from a six-layer model including a low velocity zone. We computed non-correlated noise for SWD and Gaussian correlated noise for the RF with values for r and σ as given in Table 3.3 (*true*). Noise and synthetic data were then added to create observed data. An example script, including these steps, can be found in the appendix and the online repository.

Table 3.3: Model priors and inversion parameters for synthetic test inversion and *true* values used for modeling of the observed data. Model prior tuples define the limits (min, max) of a uniform distribution.

Model priors		true	**Further parameters**	
vs	= (2, 5)	see plots	nchains	= 21
z	= (0, 60)	see plots	$iter_{burnin}$	= 100,000
layers	= (1, 20)	6	$iter_{main}$	= 50,000
vpvs	= (1.5, 2.1)	1.73	acceptance	= (50, 55)
r_{RF}	= 0.92	0.92	propdist	= (0.005, 0.005, 0.005,
σ_{RF}	= (1e-5, 0.05)	0.0052		0.005, 0.005)
r_{SWD}	= 0.	0.	rcond	= 1e-6
σ_{SWD}	= (1e-5, 0.1)	0.01	station	= 'st6'

Two targets (*PReceiverFunction*, *RayleighDispersionPhase*) were initialized with the "observed" data and combined to a *BayHunter.JointTarget* object. The latter and the two parameter dictionaries of model priors and inversion parameters (Tab. 3.3) were passed to the Optimizer. Parameters that were not defined fall back to the default values. We purposely show a run with only 150,000 iterations to visualize the convergence of different chains and the outlier detection method. The inversion finished after 20 minutes, saving and plotting methods were applied afterwards.

Figure 3.2 shows the likelihood development over the iterations for all and for a selection of chains. A strong increase of likelihood can be observed at the first iterations in the burn-in phase, converging towards a stable value with increasing number of iteration. Some chains reached the final likelihood plateau in the burn-in phase (e.g., $c0$), some within the posterior sampling phase (e.g., $c4$), and some did not converge at all ($c2$). The chain $c2$ (also $c1$ and $c3$) had a good chance of reaching the maximum likelihood, if the small number of iterations would not have stopped the exploration at this early stage. However, the number of iterations cannot be eternal; in any case it is necessary to compare the convergence level of the chains.

Here, we defined a 0.02 deviation condition for outliers. With a maximum median posterior likelihood of 1674 ($c13$), the likelihood threshold is 1640, which declared 13 chains with deviations of 0.032–0.159 as outliers (see Tab. 3.4). In a real case inversion, the number of iterations should

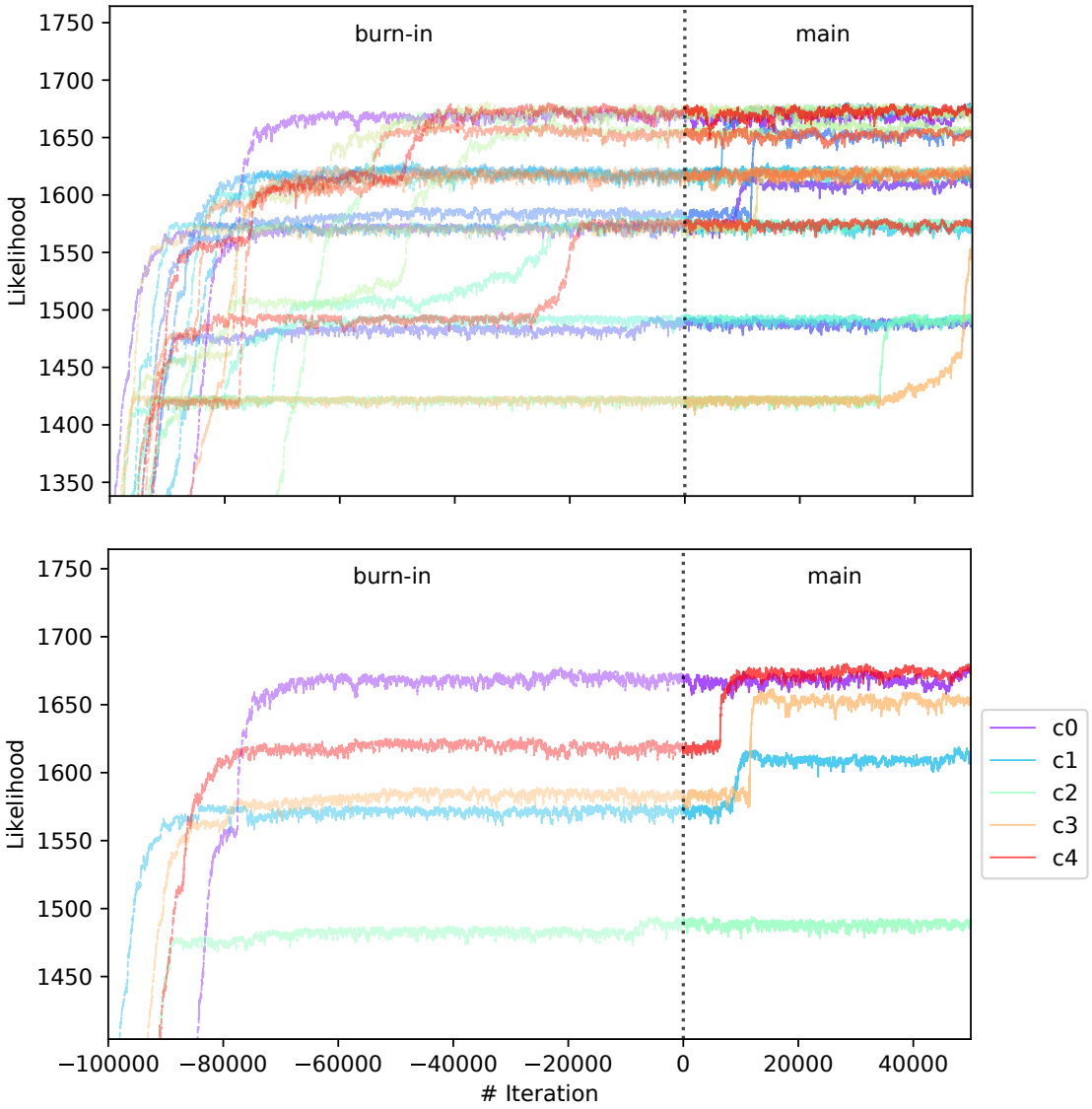

Figure 3.2: Development of likelihood over iteration for all 21 chains (top) and a small selection of chains (bottom).

be much higher, and the number of outlier chains is small. The detected outlier chains will be excluded from the posterior distribution.

Table 3.4: Deviations of each chain's median likelihood from the maximum median likelihood of the chain ensemble. Only outlier chains with deviations >0.02 (2 %) are listed.

c1	0.039	c6	0.061	c9	0.059	c15	0.150	c19	0.059
c2	0.111	c7	0.033	c10	0.150	c16	0.033		
c5	0.032	c8	0.109	c14	0.033	c17	0.033		

Figure 3.3 shows the current V_S-depth models from different chains and corresponding data fits (same chains as in Fig. 3.2, bottom). Chains $c1$ and $c2$ show the worst data fits; they were declared as outliers. The other chains ($c0$, $c3$, $c4$) show a reasonably good data fit with very similar velocity models. Chains $c0$ and $c4$ already found a six-layer model, $c3$ found a five-layer

model averaging the low velocity zone.

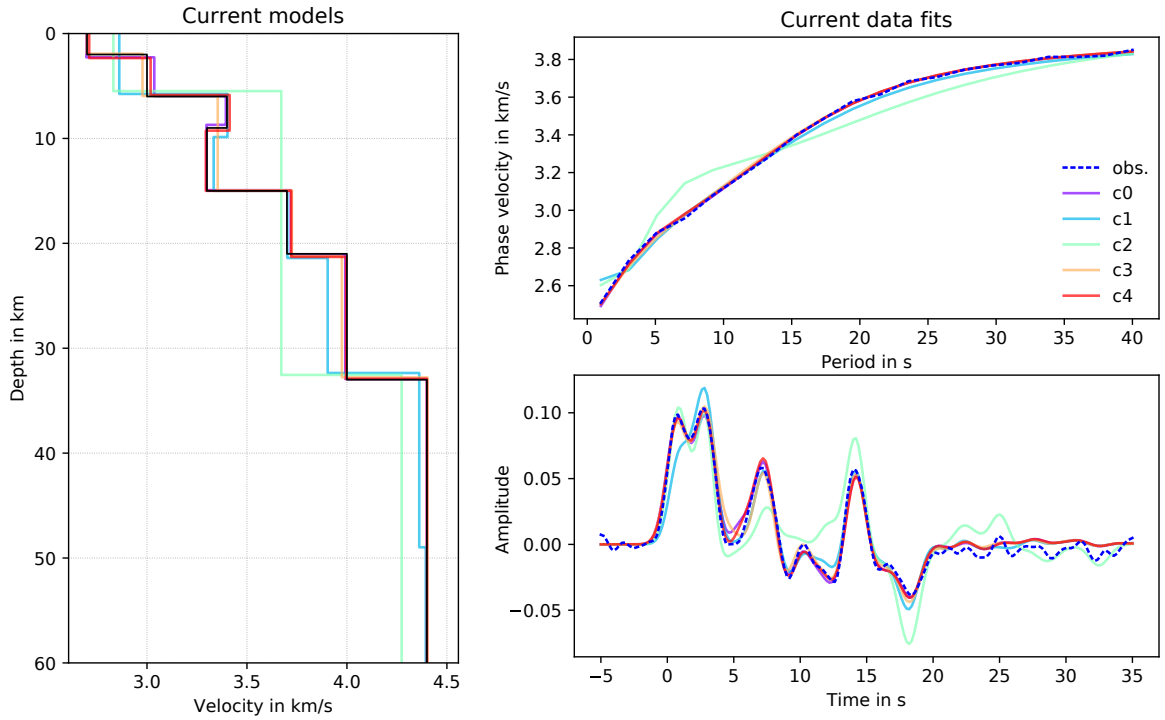

Figure 3.3: Current velocity-depth models and data fits of corresponding SWD and RF data from different chains with likelihoods as illustrated in Figure 3.2 (bottom). The black line (left) is the synthetic V_S-depth structure.

The posterior distribution of the eight converged chains, containing 100,000 models, are illustrated in Figure 3.4. The mean (and mode) posterior V_S-depth structure images the true model very well, including the low velocity zone. The number of layers is determined to be most likely six. The σ distributions of both, RF and SWD show a Gaussian shape, inhering a tail of higher values from models of chains that only converged within the exploration phase of the inversion (e.g., $c3$ and $c4$). The distribution of σ_{SWD} already represents a good estimate, slightly overestimated, which falls back to the number of iterations. Tests with more iterations show that the median of σ_{SWD} is in perfect agreement with the true value.

σ_{RF} is underestimated, which theoretically means that noise was interpreted as signal and receiver function data is overfitted. The difference to SWD is the type of noise correlation (= Gaussian) and the assumption of the correlation r of data noise ($r \neq 0$). We computed synthetic RF data applying a Gaussian lowpass filter with a Gaussian factor of 1. Separately, noise was generated randomly with a correlation r estimated to represent the applied Gauss filter, and added to the synthetic data. The random process of generating noise does not output a noise vector which exactly matches the given values of r and σ. If only drawing one single realization with a determined amount of samples from the multivariate normal distribution may produce deviations from the targeted r and σ. From the generated noise the true σ can be computed by the standard deviation. However, the true r is not easy to reconstruct. Assuming a wrong r for the covariance matrix of noise cannot lead to the correct σ.

It is possible to clarify whether the assumed correlation parameter r is in tendency correct.

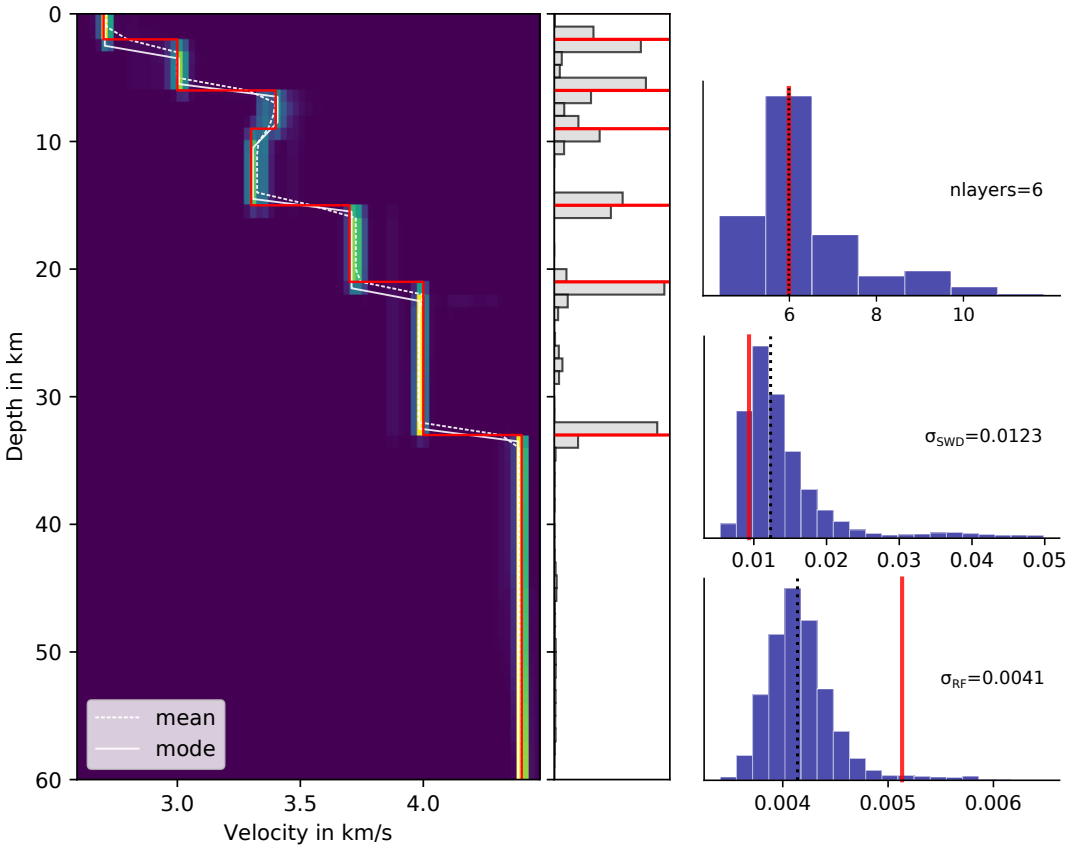

Figure 3.4: Recovered posterior distributions of V_S, interface depth, number of layers, and noise level for synthetic data. Red lines indicate the true model, as given in Table 3.3. The posterior distribution is assembled by 100,000 models collected by 8 chains.

Figure 3.5 shows a comparison of (1) the RF data residuals of the best fitting model and (2) one realization of noise with the given correlation r and the estimated σ_{RF}; both noise vectors should be of coherent appearance in frequency and amplitude, if the estimate of r_{RF} is appropriate.

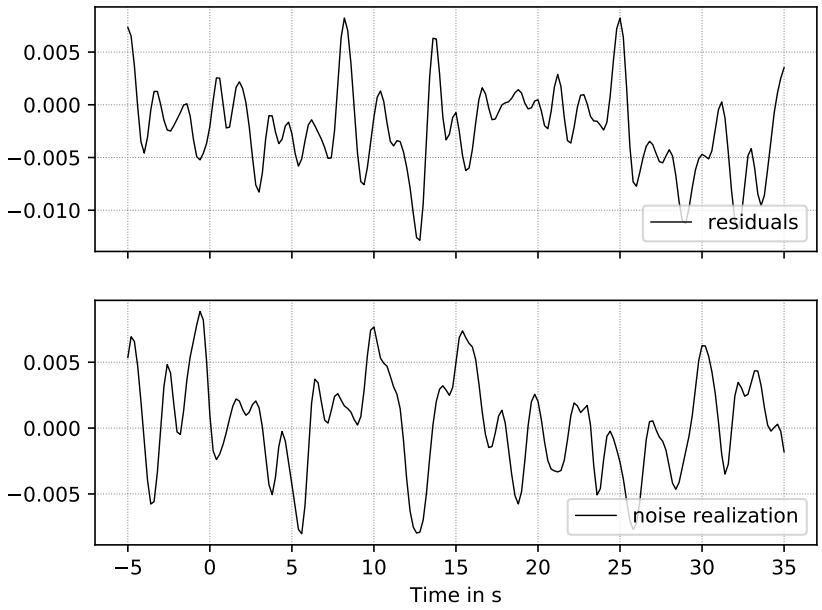

Figure 3.5: Comparison of residuals of the best fitting RF model and one realization of noise through C_e^{RF} for receiver functions. Both noise vectors are of coherent appearance in frequency and amplitude, hence, the estimate of r_{RF} is appropriate.

Bibliography

Bayes, T. An Essay Towards Solving a Problem in the Doctrine of Chances. By the late Rev. Mr. Bayes, F. R. S. communicated by Mr. Price, in a letter to John Canton, A. M. F. R. S. 1763. doi: 10.1098/rstl.1763.0053.

Berteussen, K.-A. Moho depth determinations based on spectral-ratio analysis of NORSAR long-period P waves. *Physics of the Earth and Planetary Interiors*, 15(1):13–27, 1977. doi: 10.1016/0031-9201(77)90006-1.

Bodin, T. *Transdimensional Approaches to Geophysical Inverse Problems*. PhD thesis, The Australian National University, 2010.

Bodin, T., Sambridge, M., Tkalčić, H., Arroucau, P., Gallagher, K., and Rawlinson, N. Transdimensional inversion of receiver functions and surface wave dispersion. *Journal of Geophysical Research: Solid Earth*, 117(B2):B02301, 2012. doi: 10.1029/2011JB008560.

Dreiling, J., Tilmann, F., Yuan, X., Haberland, C., and Seneviratne, S. Crustal structure of Sri Lanka derived from joint inversion of surface wave dispersion and receiver functions using a Bayesian approach. *Journal of Geophysical Research: Solid Earth*, 2020. doi: 10.1029/2019JB018688.

Green, R., Sens-Schönfelder, C., Shapiro, N., Koulakov, I., Tilmann, F., Dreiling, J., Luehr, B., Jakovlev, A., Abkadyrov, I., Droznin, D., and Gordeev, E. Magmatic and sedimentary structure beneath the klyuchevskoy volcanic group, kamchatka, from ambient noise tomography. *Journal of Geophysical Research: Solid Earth*, 2020. doi: 10.1029/2019JB018900.

Herrmann, R. B. and Ammon, C. J. Computer Programs in Seismology: Surface waves, receiver functions and crustal structure, 2002.

Mahalanobis, P. C. On the generalized distance in statistics. 1936.

Mauerberger, A., Sadeghisorkhani, H., Maupin, G. O., V., and Tilmann, F. The multifaceted scandinavian lithosphere imaged by surface waves and ambient noise (in prep). n.a.

Appendix

Inversion example

```
import numpy as np
import os.path as op
from BayHunter import utils
from BayHunter import SynthObs
from BayHunter import Targets
from BayHunter import MCMC_Optimizer
from BayHunter import PlotFromStorage

# ------------------------------------------------- obs SYNTH DATA
# Load observed data (synthetic test data)
xsw, _ysw = np.loadtxt('rdispph.dat').T
xrf, _yrf = np.loadtxt('prf.dat').T

# Create noise and add to synthetic data
ysw = _ysw + SynthObs.compute_expnoise(_ysw, corr=0, sigma=0.012)
yrf = _yrf + SynthObs.compute_gaussnoise(_yrf, corr=0.92, sigma=0.005)

#  ------------------------------------------------------- TARGETS
# Assign data to target classes
target1 = Targets.RayleighDispersionPhase(xsw, ysw)
target2 = Targets.PReceiverFunction(xrf, yrf)
target2.moddata.plugin.set_modelparams(gauss=1.0, water=0.01, p=6.4)

# Join the targets
targets = Targets.JointTarget(targets=[target1, target2])

#  ------------------------------------------------- PARAMETERS
# Define parameters as dictionaries ...
priors = {'vs': (2, 5),
          'layers': (1, 20),
          'vpvs': 1.73,
          'rfnoise_corr': 0.92,
          ...
          }

initparams = {'nchains': 21,
              'iter_burnin': 100000,
              'iter_main': 50000,
              ...
              }

# ... or load from file
initfile = 'config.ini'
priors, initparams = utils.load_params(initfile)

#  ------------------------------------------------ MCMC INVERSION
```

```
# Save configfile for baywatch
utils.save_baywatch_config(targets, priors=priors, initparams=initparams)
optimizer = MCMC_Optimizer(targets, initparams=initparams, priors=priors,
                           random_seed=None)

# start inversion, activate BayWatch
optimizer.mp_inversion(nthreads=8, baywatch=True, dtsend=1)

#  ------------------------------------------------ SAVING / PLOTTING
# Initiate plotting object
path = initparams['savepath']
cfile = '%s_config.pkl' % initparams['station']
configfile = op.join(path, 'data', cfile)
obj = PlotFromStorage(configfile)

# Save posterior distribution to combined files, incl. outlier detection
obj.save_final_distribution(maxmodels=100000, dev=0.05)

# Save a selection of important plots
obj.save_plots()
obj.merge_pdfs()
```

Estimation of r_{RF}

```
import numpy as np
import matplotlib.pyplot as plt
from BayHunter import utils

# ------------------------------------------------ RF and parameters
# load observed data (synthetic RF, Gauss factor a=1)
rfx, rfy = np.loadtxt('observed/st3_prf.dat').T
rfa = 1  # a

# define parameters
dt = 0.2
draws = 40000
rrfs = [0.75, 0.85, 0.95, 0.97, 0.98, 0.99]

pars = {'rfx': rfx, 'rfy': rfy, 'rfa': rfa,
        'a': rfa, 'dt': dt, 'rrfs': rrfs,
        'draws': draws}

# ------------------------------- visualize 'raw' data and estimates
fig = utils.plot_rrf_estimate(pars=pars)
fig.savefig('st3_rrf_estimate.pdf', bbox_inches='tight')


# ---------------------------- return values for costum visualization
# update parameters...
pars['rrfs'] = np.linspace(0.9, 0.999, 25)
pars['draws'] = 2000
```

```
28
29 # get rrf-values with corresponding a-values
30 rrf, a = utils.rrf_estimate(pars=pars)
31
32 # custom plot example
33 fig, ax = plt.subplots()
34 ax.plot(rrf, a, color='k', marker='x', ls='')
35 ax.axhline(rfa, color='gray', label='reference')
36 ax.set_xlabel('$r_{RF}$')
37 ax.set_ylabel('Gauss factor a')
38 ax.grid(color='lightgray')
39 ax.legend(loc=1)
40 fig.savefig('rrf-a_rel.pdf', bbox_inches='tight')
```