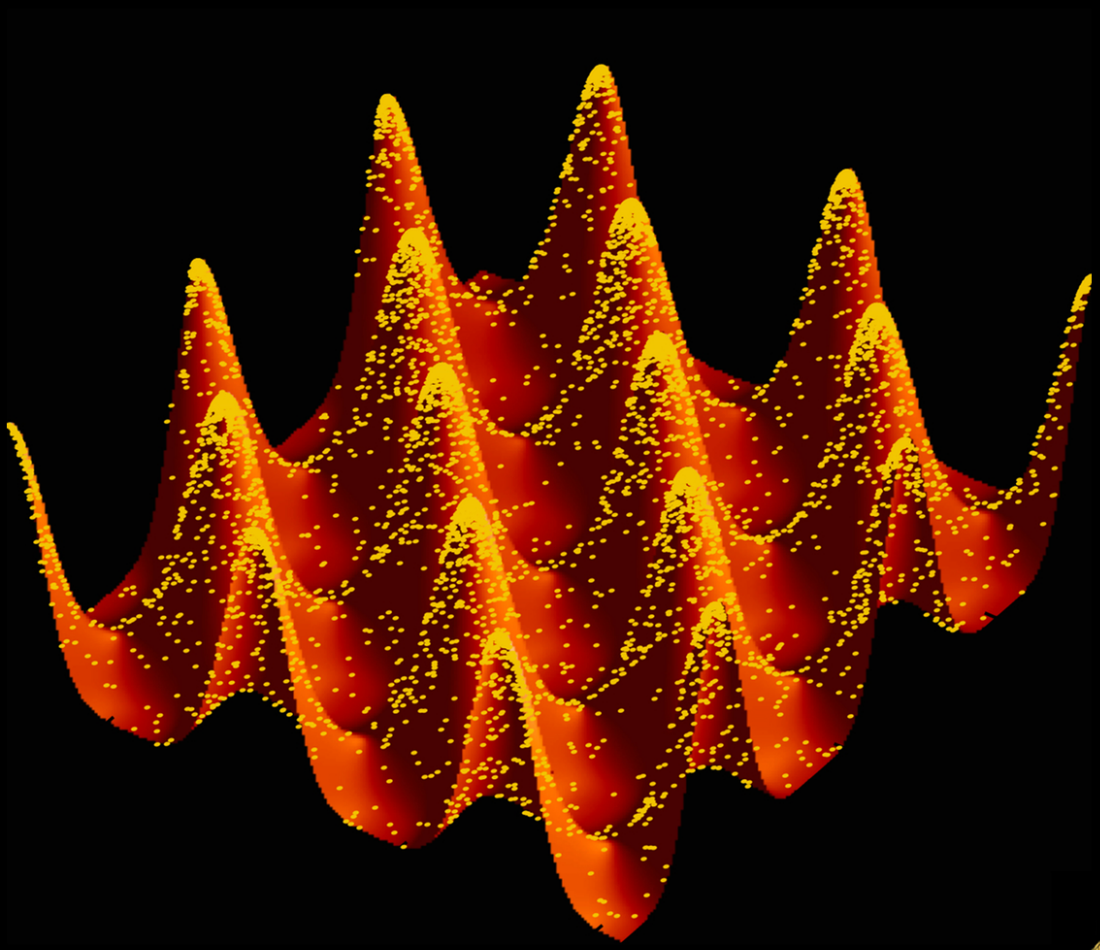
DIAMONDS™

BAYESIAN SOFTWARE

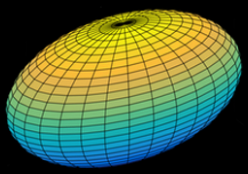
User Guide

Manual

Enrico Corsaro



Based on
Ellipsoidal Sampling



KATHOLIEKE UNIVERSITEIT LEUVEN

2015 Edition

User Guide
Manual

Enrico Corsaro

Research Associate Scientist
emncorsaro@gmail.com / enrico.corsaro@cea.fr

Laboratoire AIM, CEA/DSM – CNRS – IRFU/SAp, Centre de Saclay
Université Paris Diderot, France

Instituto de Astrofisica de Canaries & Departamento de Astrofisica
Universidad de La Laguna, Spain

2015
Edition

Contents

Abstract

The DIAMONDS (high-DImensional And multi-MOdal NesteD Sampling) code performs a Bayesian parameter estimation and model comparison by means of the nested sampling Monte Carlo (NSMC) algorithm, an efficient and powerful method very suitable for high-dimensional and multi-modal problems. This code can be used for any application involving Bayesian parameter estimation and/or model selection in general, with no given limitations imposed on both the extent of the datasets and the number of free parameters involved in the model. Developed in `C++11`, DIAMONDS is structured in classes for flexibility and configurability. Any new model, likelihood and prior probability density functions (PDFs) can be defined and implemented upon a basic template known as an abstract class. The first chapter of this user guide manual introduces the different parts of the code and their implementation in a standard format main function with directly usable code examples, thus continuing with a description of the individual outputs created at the end of the computation. In the second chapter we discuss the individual configuring parameters that characterize the ellipsoidal sampling algorithm, on which the current version of DIAMONDS is based. The third chapter is instead focused on troubleshooting of the main computational failures that may occur during the execution of a process. To conclude, in the fourth chapter we provide a detailed documentation on how to check and understand the reliability of the results, an essential step for using the code with consciousness. The official website with free software download, package content description, and installation guide and compilation troubleshooting for both Mac and Linux OS is available at the URL: https://fys.kuleuven.be/ster/Software/Diamonds/.

1 Getting started

The DIAMONDS (high-DImensional And multi-MOdal NesteD Sampling) code presented in this user guide manual is a general Bayesian inference (both parameter estimation and model comparison) tool developed in `C++11` and structured in classes in order to be as much flexible and configurable as possible. The working flow from the main function of the code is as follows (see also Figure 1.1):

1. Read an input dataset
2. Set up model, likelihood, and prior distributions to be used in the Bayesian inference
3. Set up a drawing algorithm
4. Configure and start the nested sampling
5. Compute and print the final results

This represents the general sequence of steps that is necessary to follow in the exact order indicated for guaranteeing the operations. Throughout this manual, we will assume that the user has already read the original paper of the code, provided by [7]. It is essential to have a basic knowledge about all the working method, the algorithm, and the features implemented in DIAMONDS to be able to use in the best way the information provided. All the documentation presented here, especially concerning the meaning and behavior of the configuring parameters of the code, the troubleshooting and the inspection of the results, is based on the experience acquired using DIAMONDS for the scientific works published by [7, 9, 8]. This manual will be updated from time to time whenever more useful testing of the code is available. Users are encouraged to provide their feedback to help us improving the tool and its description. We recommend to use DIAMONDS with consciousness, not as a black box, and to always inspect its results with critical sense before adopting them for any purpose.

We start in this chapter by showing some preliminary information to help the user becoming more familiar with the different parts of the code, thus complementing the documentation with code examples that can be directly implemented in the main function of your application based on DIAMONDS. We consider the drawing algorithm described in [7], namely the Multi Ellipsoidal Sampler based on the simultaneous ellipsoidal sampling (SES) and the X-means clustering algorithms. This drawing system is provided in the core package of the code, available for free download in the official website. A detailed description of the package content is given here, while a comprehensive installation guide for both Mac and Linux OS, including a troubleshooting to compilation errors, is available here.

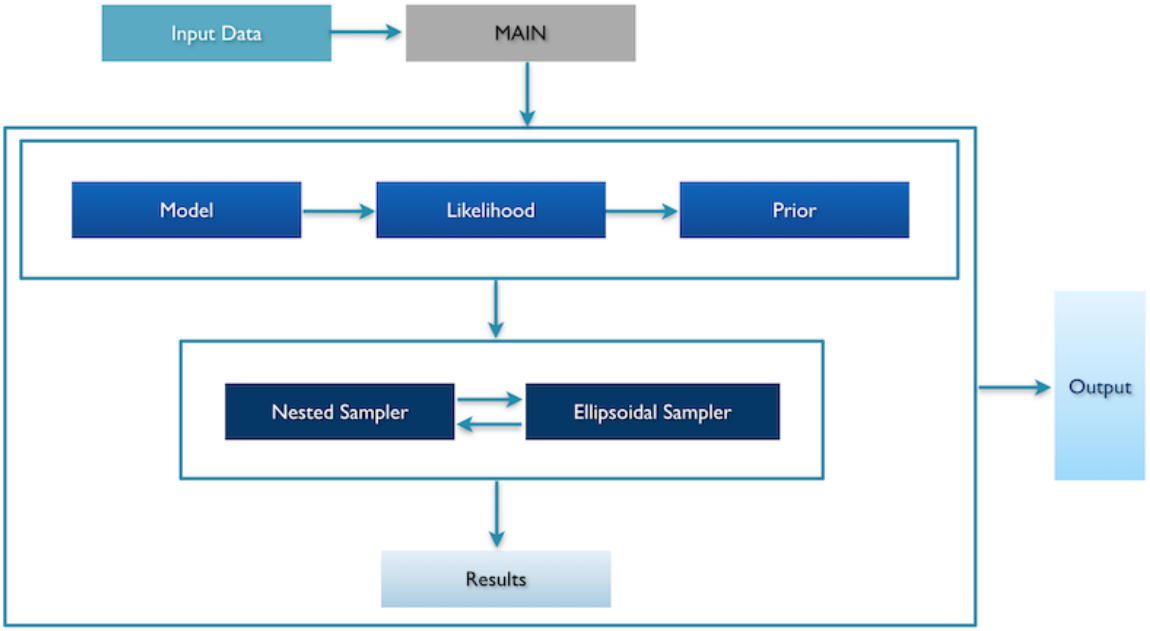


Figure 1.1: General working flow of the DIAMONDS code.

1.1 The input dataset

The first element in any Bayesian inference or model comparison problem is the input dataset that we want to investigate. The structure of an input dataset is represented by an ASCII file with a two-column format, namely the covariates (or independent variables, or x-axis values) and the observations (or dependent variables, or y-axis values). In some cases, the input data file could include a third column with uncertainties on the observations. The input data file must be read and stored in Eigen[1] arrays at the beginning of the main function of the specific application using DIAMONDS. Clearly this is an essential step to carry on with the computations. An example of code using Eigen arrays of double type values for reading an input dataset is

```
unsigned long Nrows;
int Ncols;
ArrayXXd data;
ifstream inputFile;
File::openInputFile(inputFile, inputFileName);
File::sniffFile(inputFile, Nrows, Ncols);
data = File::arrayXXdFromFile(inputFile, Nrows, Ncols);
inputFile.close();

// Creating arrays for each data type
ArrayXd covariates = data.col(0);
ArrayXd observations = data.col(1);
ArrayXd uncertainties = data.col(2);
```

[1] Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms that is used by DIAMONDS, and it is particularly efficient and optimized for cache-friendliness. You can find the official website with free download and full documentation here.

where we assumed that also uncertainties were provided as a third column. In order to read the content of the ASCII file, we used the C++ input file stream class ifstream, coupled with the functions arrayXXdFromFile(), openInputFile() and sniffFile() of the File namespace provided in the DIAMONDS code package. The integers Nrows and Ncols will contain the number of data bins sorted by row, and the number of data columns (3 for the given example), respectively. It is important to note that all the data arrays must contain exactly the same number of elements. If this is not not verified, the code will not be able to start the computation. The string variable inputFileName will contain the exact name and full path of the input file.

1.2 The model

A second preliminary step for the Bayesian inference concerns the definition of the model that we want to use for fitting the dataset. This model can be implemented starting from the basic abstract class Model included in the code package. The order of the free parameters defined in the implementation of the model is a sensible information through the computation process. The same order of the free parameters will be that of the prior distributions and of the output results. In order to compute the predictions to the observations, the model will require as input the array containing the covariates from the dataset. This is because the array containing the fitting model, usually termed *predictions*, will be of the same length of the covariates array. The object containing the model of the derived class, here labeled UserSpecifiedModel, has to be initialized as in the following example:

```
UserSpecifiedModel model(covariates);
```

1.3 The likelihood function

Given the dataset available, the choice of a proper likelihood function is essential. In most of the applications, a Normal likelihood function, as the one described by [6] and used by [9], implemented in the NormalLikelihood class of the code package, will be the one that we need. In this case, the input dataset must contain a three-column format, with uncertainties included as a third column. In other applications as those regarding the analysis of Fourier transform datasets (e.g. in asteroseismology), an Exponential likelihood function is instead the most suited (e.g for the problems described in [7, 8]), and you can find it implemented in the ExponentialLikelihood class of the package. Nonetheless, the user has the possibility to implement its own likelihood function by starting from the basic abstract class Likelihood. The likelihood function will require as input the object containing the model, created in the previous step, and the observations from the input dataset (and additionally the uncertainties if they are used). This is because for each data point corresponding to an observation, a prediction from the model is given so that the likelihood for that measurement can be evaluated. The basic package contains also another interesting likelihood, which is a Normal likelihood averaged over the uncertainties (see [5, 2] for more details), which can be useful for problems where the uncertainties on the observations are not entirely trusted and need to be verified (see the MeanNormalLikelihood class). An example code on how to create a NormalLikelihood class object reads

```
NormalLikelihood likelihood(observations, uncertainties, model);
```

while one for an ExponentialLikelihood class object is

```
ExponentialLikelihood likelihood(observations, model);
```

where this time no uncertainties are given.

1.4 The prior probability distribution

For each of the free parameters of the model, the user has to define a prior probability distribution, which is a key ingredient in the Bayesian approach. The choice of priors should be done consciously as it may have an impact on the final result. The prior distribution can be included either by using the existing distributions provided in the code package (in the classes UniformPrior, NormalPrior and SuperGaussianPrior), or by implementing your own one based on the basic abstract class Prior. The priors have to be defined in the main function of the code and they order must follow that in which the free parameters implemented in the model are defined. In addition, they clearly have to be in number equal to the number of free parameters of the model. If this is not happening, the computation will not be able to start.

The optimal prior distribution to guarantee highest computational speed is that of the uniform priors, which only requires the definition of lower and upper parameter bounds allowed within the inference problem, although assuming a fixed range of variation for a free parameter is not reflecting a total status of ignorance about that parameter itself [14], so they have to be chosen with care. However, users can define multiple prior types and combine them in any order based on the set of free parameters available. To use multiple prior types, one has to define a vector of pointers to abstract Prior class objects, as shown in the following example for two different types

```
int NpriorTypes = 2;
vector<Prior*> ptrPriors(NpriorTypes);
PriorClassName0 priorClassName0(hyperParameters0);
PriorClassName1 priorClassName1(hyperParameters1);
ptrPriors[0] = &priorClassName0;
ptrPriors[1] = &priorClassName1;
```

where PriorClassName<prior#> is either one of the prior classes provided in the DIAMONDS code package or implemented separately by the user. Each element in the ptrPriors vector will correspond to either a single free parameter or a set of consecutive (stacked contiguously) free parameters for which the same prior type is adopted. Each prior object has to be initialized with a specific set of so-called hyper parameters, which in turn will depend on the corresponding free parameters for which we define the prior PDF[2].

There is also the possibility to store the hyper parameters used in the computation in an ASCII file that one can retrieve afterwards. This can be useful if the user wants to keep track of which prior hyper parameters were used for that specific process, in case multiple tests need to be performed. To do so, we can use the member function writeHyperParametersToFile() of the individual prior class and write the following line of code

[2]See [7] for more details on what are the hyper parameters for each prior type of those provided in the package.

```
string fullPathHyperParameters = outputPathPrefix + "hyperParametersClassName.txt";
priorClassName.writeHyperParametersToFile(fullPathHyperParameters);
```

where the string specifies the name of the file completed by the full path where we desire store all the output files (including a filename prefix to identify the name of the application), which we labeled as the string `outputPathPrefix`. The format of the file will be that of a table of values in which each row corresponds to a free parameter, in the same order given by the model, while each column represents one hyper parameter of the prior in the same order as given as input. We need to keep in mind that if we are using different prior types, this operation must be done separately for each of them. Following the example given before we will have

```
string fullPathHyperParameters0 = outputPathPrefix + "hyperParametersClassName0.txt";
priorClassName.writeHyperParametersToFile(fullPathHyperParameters0);
string fullPathHyperParameters1 = outputPathPrefix + "hyperParametersClassName1.txt";
priorClassName.writeHyperParametersToFile(fullPathHyperParameters1);
```

in which we changed both the prior class name and the file name to avoid overwriting.

1.5 Configuring the cluster algorithm

There are two distinct groups of input parameters that are used to set up the core of the sampling algorithm. The first group is related to the X-means clustering algorithm, with minimum and maximum number of clusters to be specified, and implemented in the `KmeansClusterer` class. A more detailed discussion about the number of clusters to be used is addressed in Section 2.1. A code example of how to configure the clusterer with input parameters read from an input file is given below:

```
// Read minimum and maximum number of clusters from an input file
inputFileName = outputDirName + "Xmeans_configuringParameters.txt";
File::openInputFile(inputFile, inputFileName);
File::sniffFile(inputFile, Nparameters, Ncols);
ArrayXd configuringParameters;
configuringParameters = File::arrayXXdFromFile(inputFile, Nparameters, Ncols);
inputFile.close();
int minNclusters = configuringParameters(0);
int maxNclusters = configuringParameters(1);

// Define an Euclidean metric for the X-means clustering
int Ntrials = 10;
double relTolerance = 0.01;
EuclideanMetric myMetric;
KmeansClusterer kmeans(myMetric, minNclusters, maxNclusters, Ntrials, relTolerance);
```

where we need to define an object, of the class type `Metric`, which specifies the metric to be used within the X-means. This is usually an Euclidean metric for all standard applications, hence an `EuclideanMetric` class. Similarly to what done when reading the input dataset, the integers `Nparameters` and `Ncols`, to be declared earlier in the main function, will contain the number of parameters sorted by row (in this case 2), and the number of columns (1), respectively. The parameters defining the number of trails and the relative tolerance are kept

fixed to the values indicated because they were calibrated in the testing phase. We considered that the input file containing the configuring parameters is stored in the main folder containing the results. This is a good rule of thumb to avoid messing up the files used for an individual process with those coming from a different run. Reading the configuring parameters from an input files is a very convenient choice to avoid recompiling the code every time we intend to try a different value for one or more of them.

1.6 Configuring the nested sampling algorithm

The second group of configuring parameters concerns the nested sampling algorithm and the effective drawing mechanism done by means of the ellipsoids, implemented in the classes `NestedSampler` and `MultiEllipsoidSampler`, respectively. These parameters were already introduced and described by [7], Section 4.1, but in the context of this user guide we will put more focus on using two of them, namely the initial enlargement fraction f_0 and the shrinking rate α of the ellipsoids. The ellipsoids have the great advantage of allowing us reducing prior volume to draw from as the computation evolves, hence speeding up the computation itself [12, 13]. Nonetheless, ellipsoids have the drawback of requiring careful configuration because a wrong set of input parameters may affect the sampling, hence the reliability of the results, especially of the uncertainties on the free parameters. In the worst case, a wrong configuration will cause the computation to stop prematurely. In the following Chapter 3 and Section 4 we shall investigate the most common problems arising when dealing with the ellipsoids. The recipes provided should ensure a proper configuration of DIAMONDS and therefore producing reliable results for a wide variety of applications. The values suggested for the different parameters and an explanation are provided in Section 2.

The code below shows how to read the configuring parameters of the nested sampler based on the SES from an input file, assuming the parameters are listed in a row format, similarly to what done for the clusterer.

```
inputFileName = outputDirName + "NSMC_configuringParameters.txt";
File::openInputFile(inputFile, inputFileName);
File::sniffFile(inputFile, Nparameters, Ncols);
configuringParameters.setZero();
configuringParameters = File::arrayXXdFromFile(inputFile, Nparameters, Ncols);
inputFile.close();

// Print results on the screen
bool printOnTheScreen = true;

// Initial number of live points
int initialNlivePoints = configuringParameters(0);

// Minimum number of live points
int minNlivePoints = configuringParameters(1);

// Maximum number of attempts when trying to draw a new sampling point
int maxNdrawAttempts = configuringParameters(2);

// The first N iterations, we assume that there is only 1 cluster
int NinitialIterationsWithoutClustering = configuringParameters(3);

// Clustering is only happening every N iterations
int NiterationsWithSameClustering = configuringParameters(4);
```

```
// Fraction by which each axis in an ellipsoid has to be enlarged
// It can be a number >= 0, where 0 means no enlargement
double initialEnlargementFraction = configuringParameters(5);

// Exponent for remaining prior mass in ellipsoid enlargement fraction.
// It is a number between 0 and 1.
// The smaller the slower the shrinkage of the ellipsoids.
double shrinkingRate = configuringParameters(6);

// Termination factor for nested sampling process
double terminationFactor = configuringParameters(7);
```

where we enabled the flag `printOnTheScreen` to visualize the status of the computation on the terminal screen every 50 nested iterations (see Section 1.7 for more details). In the final part, we have to pass this information to the sampler, hence execute the algorithm by using the member function `run()` of the `NestedSampler` class, as shown in the following code

```
MultiEllipsoidSampler nestedSampler(printOnTheScreen, ptrPriors, likelihood, myMetric,
kmeans, initialNlivePoints, minNlivePoints, initialEnlargementFraction, shrinkingRate);

double tolerance = 1.e2;
double exponent = 0.4;
PowerlawReducer livePointsReducer(nestedSampler, tolerance, exponent, terminationFactor);

nestedSampler.run(livePointsReducer, NinitialIterationsWithoutClustering,
NiterationsWithSameClustering, maxNdrawAttempts, terminationFactor, outputPathPrefix);
```

where the `PowerLawReducer` class object allows for a reduction of the live points according to the power law relation provided by [7] (see their Section 4.4 for more details). Alternatively one could use the law provided by [4], which is implemented in the `FerozReducer` class. The values of the parameters defining the tolerance and the exponent of the live points reducer are only shown as an example. The reducer has to be defined in any case, even if not used (see also Section 2). The string variable `outputPathPrefix` is the same as that used for the prior hyper parameters, Section 1.4, and usually contains a subfolder of `outputDirName` named with the label of the run number that we are using[3] (see also in Section 1.7).

1.7 What output is obtained and how to use it

When the flag `printOnTheScreen` is turned on, a line of output is printed on the screen every time that 50 consecutive nested iterations are completed. An example of a line of this output reads

```
Nit: 50  Ncl: 1  Nlive: 2000  CPM: 0.0251776  Ratio: 6.45e+102  log(E): -2.375  IG: 3.68
```

[3]The string `outputPathPrefix` can be defined as in the following: `outputPathPrefix = outputDirName + runNumber + "/applicationName_"`, where `runNumber` is the string corresponding to the label of the given run (usually you have more than one run and you can label them with progressive integer numbers, e.g. `00, 01, ...`), while the last bit of the string represents the label prefix that identifies the given application and that is attached to all the output file names, e.g. for the case of a peak bagging analysis it would be `peakbagging_`, and we will therefore have for instance the parameter summary file the name `peakbagging_parameterSummary.txt`. This is a simple way to avoid confusing the results from different runs and applications.

Nit is the number of the current nested iteration, in this case corresponding to the second line printed on the screen (the first one is for Nit: 0). Ncl is the actual number of clusters (and not the input ones we provided), and it will be able to vary within the allowed range as the computation proceeds. Nlive is the number of live points used in the given nested iteration, and this information becomes useful especially when the reduction of the live points is taking place, which causes Nlive to vary. CPM represents the cumulated prior mass, which converges to 1 (total prior probability) toward the end of the process. Ratio is the final termination condition, termed $\delta_{\rm final}$ by [7], and it is an important parameter to track for understanding how much is left to sample from the posterior probability distribution. When this number reaches the value of the termination factor given as input, the computation will end and the results can be generated. The last two numbers, log(E) and IG, are the natural logarithm of the cumulated (up to the given nested iteration) Bayesian evidence, and the information gain from the previous iteration, respectively.

When the computation is completed, one displays a message specifying the final natural logarithm of the Bayesian evidence and the error bar, and the total computational time of the process, as in the example below:

```
------------------------------------------------
 Final log(E): 3.65068e+06 +/- 0.270612
------------------------------------------------
 Total Computational Time: 1.16 hours
------------------------------------------------
```

After the computation is completed and this message appears, assuming that there were no errors throughout the process (see Chapter 3 for all the details), the class Results is invoked and all the general output files of DIAMONDS are created, except for the one containing the configuring parameters, which is created by the NestedSampler class itself as discussed below. There are several choices that can be included in the printing of the final results, but a standard (and complete) set will include the following ASCII files:

- a 1-column format configuringParameters.txt file, containing all the configuring parameters used for the computation. This file can be very useful especially when running several different processes because it allows us to keep track of which values were used for each configuring parameter of the code, even if we changed the input files. This is the only file that is created automatically by the NestedSampler class after the computation starts, and it contains the following default list of parameters:

```
# List of configuring parameters used for the NSMC.
# Row #1: Ndimensions
# Row #2: Initial(Maximum) NlivePoints
# Row #3: Minimum NlivePoints
# Row #4: NinitialIterationsWithoutClustering
# Row #5: NiterationsWithSameClustering
# Row #6: maxNdrawAttempts
# Row #7: terminationFactor
# Row #8: Niterations
# Row #9: Optimal Niterations
# Row #10: Final Nclusters
# Row #11: Final NlivePoints
# Row #12: Computational Time (seconds)
```

However, not all the parameters are stored automatically, and those belonging specifically to the ellipsoidal sampler must be included in a separate step. This is to allow for more flexibility in the event of a future replacement of the currently adopted ellipsoidal sampler. Since the related output file stream is not closed after the computation has ended, we can append additional parameters to it, by using the following standard sample code made for the ellipsoidal sampler:

```
nestedSampler.outputFile << "# List of configuring parameters used for the
ellipsoidal sampler and X-means" << endl;
nestedSampler.outputFile << "# Row #1: Minimum Nclusters" << endl;
nestedSampler.outputFile << "# Row #2: Maximum Nclusters" << endl;
nestedSampler.outputFile << "# Row #3: Initial Enlargement Fraction" << endl;
nestedSampler.outputFile << "# Row #4: Shrinking Rate" << endl;
nestedSampler.outputFile << minNclusters << endl;
nestedSampler.outputFile << maxNclusters << endl;
nestedSampler.outputFile << initialEnlargementFraction << endl;
nestedSampler.outputFile << shrinkingRate << endl;
nestedSampler.outputFile.close();
```

If no further parameters are to be printed, one should not forget to close the output file stream in any case.

- a 1-column format parameter<parameter#>.txt file, for as many free parameters as the model and priors have included. These files contain the original sampling of the parameter values following the exact order imposed by the nested sampling, hence sorted in increasing likelihood values, as given in the `logLikelihood.txt` file (hence the parameter values are not sorted in increasing parameter value conversely to what is given in the `marginalDistribution<parameter#>.txt` files). These files can be useful to construct the marginal probability distributions if one wants to start from the original sampling obtained by DIAMONDS. In addition they are needed for correlation maps jointly with the `logLikelihood.txt` file, and they are important to check for the sampling evolution of the free parameters and to inspect problems related to a bad choice of the parameter boundaries in uniform prior distributions that causing premature interruptions of the computations, as discussed in Chapter 3. These files can be created by calling the `writeParametersToFile()` member function of the `Results` class, as shown below:

```
Results results(nestedSampler);
results.writeParametersToFile("parameter");
```

where this time only the initial bit of the string (shown as `parameter`) is provided, since the number of the parameter is attached automatically and changes for each free parameter used. This numbering follows the exact order in which we defined the free parameters in the model and in the priors. In addition, these files are always produced even if the computation stops prematurely with one of the error types discussed in Chapter 3. The number of parameter values stored will correspond to the total number of nested iterations plus the number of live points used in the final iteration, which are merged to the nested sampled points to complete the final sampling of the parameter space.

- a 1-column format `logLikelihood.txt` file, containing all the values of the natural logarithm of the likelihood function, sorted in the same order as the computation has evolved, hence by increasing likelihood, and in the same number as in the `parameter<parameter#>.txt` files. This file is useful to construct correlation maps, jointly with pairs of parameter values provided in the files `parameter<parameter#>.txt` (see the previous item), as those shown by [7]. An example of correlation map that can be obtained[4] is shown in Figure 1.2. This output file can be created by calling the `writeLogLikelihoodToFile()` member function of the `Results` class, as shown below:

```
results.writeLogLikelihoodToFile("logLikelihood.txt");
```

  Likewise the `parameter<parameter#>.txt` file(s), this file is stored even if the computation stops prematurely with one of the error types discussed in Chapter 3.

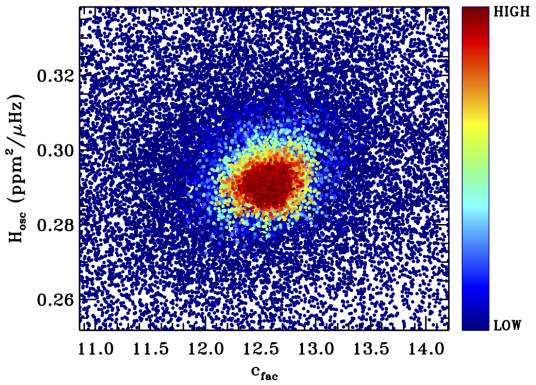


Figure 1.2: Example of correlation map that is obtained by combining the information contained in the file `logLikelihood.txt`, shown in a colored scale, with the sampling of a pair of free parameters stored in the files `parameter<parameter#>.txt`, here reporting the result obtained by [7] for the background fitting of a solar-like oscillator.

- a 3-column format `evidenceInformation.txt` file, containing the total Skilling's Bayesian evidence in natural logarithm, the corresponding error bar derived through the information theory, and the information gain. This file is essential for performing Bayesian model comparison based on the Bayesian evidence information but can be ignored if we are only interested in the parameter estimation problem. It can be created by using the `writeEvidenceInformationToFile()` member function of the `Results` class, as shown below:

```
results.writeEvidenceInformationToFile("evidenceInformation.txt");
```

  This file is stored even if the computation stops prematurely with one of the error types discussed in Chapter 3.

[4] In the example given, the plot is obtained by means of IDL using the `surface` routine set plotting in a 3D space, and the `plots` routine to add sampling points following the gradient in likelihood.

- a 1-column format posteriorDistribution.txt file, containing the probability values coming from the computation of both likelihood and Bayesian evidence, and sorted according to the evolution of the nested sampling, hence by increasing likelihood as given in the logLikelihood.txt file. These values. which are in the same number as in the logLikelihood.txt file, can be used to construct the marginal probability distribution for each free parameter, as explained by [7], by using the complementary information contained in the corresponding parameter<parameter#>.txt file. The posterior probability values can be created by calling the writePosteriorProbabilityToFile() member function of the Results class, as shown below:

```
results.writePosteriorProbabilityToFile("posteriorDistribution.txt");
```

This file is stored even if the computation stops prematurely with one of the error types discussed in Chapter 3.

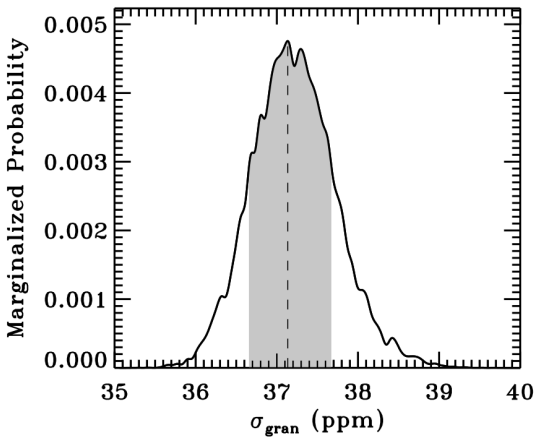


Figure 1.3: Example of a marginal probability distribution obtained by [7] using the information stored in a marginalDistribution<parameter#>.txt file for one free parameter of the background model of a solar-like oscillator. The shaded region represents the 68.3% Bayesian credible region computed by the Results class.

- a 7-column format parameterSummary.txt file, including all the Bayesian estimates computed from the marginal probability distributions (namely mean, median and mode), the Bayesian credible limits, the variance and the skewness of the distribution for each free parameter in the same order as that defined in the model and in the priors. This file is certainly the most important one in the context of Bayesian parameter estimation because it contains the desired estimates of the free parameters of the model. This information can be created by calling the writeParametersSummaryToFile() member function of the Results class, in the way shown below:

```
double credibleLevel = 68.3;
bool writeMarginalDistributionToFile = true;
results.writeParametersSummaryToFile("parameterSummary.txt",
credibleLevel, writeMarginalDistributionToFile);
```

in which we specified a 68.3 % probability of the Bayesian credible intervals (formally equivalent to the frequentist 1-σ error bar), and we required to also compute the marginal probability distributions through the flag `writeMarginalDistributionToFile` (see next item). If an assertion failure occurs, as explained in Section 3.1, this file will not produced.

- a 2-column format `marginalDistribution<parameter#>.txt` file, for as many free parameters as the model and priors have included, and labeled with the same numbering used by the `parameter<parameter#>.txt` files. These files contain the final marginal probability distribution (MPD), hence the parameter values and the corresponding marginal probabilities, sorted in increasing parameter values. The number of values listed is not corresponding to that given in the previous files, since for their computation the algorithm performs a sorting and a re-binning of the initial sampling. These files are needed when plotting the final results and check their reliability, as it will be discussed in Section 4. An example of MPD from the application done by [7] is shown in Figure 1.3. These output files are created only if specified in the `writeParametersSummaryToFile()` member function of the `Results` class, as described in the previous item. If an assertion failure occurs, as explained in Section 3.1, one or more of these files, depending on which free parameter caused the process to stop, will not be produced.

All the ASCII files will be stored in the output folder of the run, which is specified as an input string in the `run()` member function of the `NestedSampler` class, as shown in Section 1.6.

2 The enlargement fraction f of the sampling ellipsoid

To sample the parameter space in an efficient manner, the nested sampling algorithm implemented in DIAMONDS uses an ellipsoidal sampler based on the existing pioneering works by [12, 13, 3, 4]. The basic concept of the ellipsoidal sampler was already explained by [7], but we briefly recall it in this chapter to help with the reading of the attached documentation. The procedure to draw a new point from the parameter space is as follows:

1. The set of live points at a given nested iteration is processed by the X-means clustering algorithm, in order to identify which and how many clusters the live points can be split into.

2. For each cluster identified, a multidimensional ellipsoid is computed, with a number of dimensions imposed by the number of free parameters of the inference problem. The volume contained within each ellipsoid will globally allow to approximate the volume of parameter space where all the live points are situated.

3. An ellipsoid among those so constructed is randomly chosen from the set (following some criterions), and a new sampling point is drawn from its volume with an exact algorithm.

4. The likelihood of the drawn point is evaluated, in order to verify whether it satisfies the hard likelihood constraint imposed by the nested sampling, namely that the new point must correspond to a likelihood value that is at least better than the lowest among the current set of live points.

5. If the hard constraint in likelihood is satisfied, then the point is accepted and added to the remainder sample of points identified during the nested sampling process, thus moving to the next nested iteration. If not, the drawing process is repeated from step #3 onwards, until a new point is found.

Figure 2.1 shows an example of this process at a fixed nested iteration, where we find a set of live points in three dimensions that is distributed forming two distinct groups, identified as two different clusters by X-means. The 3D ellipsoids overlaid represent the geometrical construction from the covariance matrix obtained from the individual clusters. As it appears evident, the two three-dimensional shapes well surround each cluster of points, thus allowing us to improve the efficiency in drawing a new point. This is because the volume we are considering for the drawing process is no longer the entire parameter space, in this case a box in three dimensions, but only the small portion contained in each of the ellipsoids that is placed inside this space.

Nonetheless, a simple ellipsoid directly obtained from the covariance matrix does not provide an adequate solution, as discussed by [12, 13]. The reason is that such ellipsoid would be too

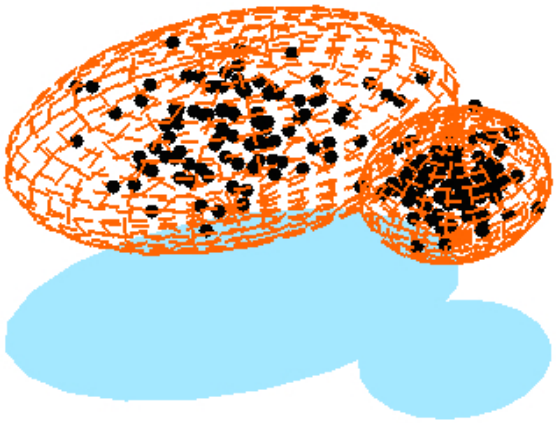


Figure 2.1: An example of two clusters of live points (black dots) in three dimensions for which two distinct ellipsoids are constructed. As visible from the figure, the volume of the ellipsoids (the 3D surfaces shown with red edges) nearly contains all the live points considered, and therefore represents a good approximation of the current distribution of sampling points in the parameter space. This allows drawing a new point very efficiently.

small to contain all or at least most of the points in the cluster, hence we need to enlarge its size to avoid losing important regions of parameter space to sample from. For this reason, [12] introduced the so called enlargement of the ellipsoids, an extra parameter that allows the default size of the ellipsoids to be adjusted in order to improve the overall sampling efficiency. Following studies done by [3, 4] showed that this enlargement can also be made dynamic, meaning that it can change through the evolution of the nested sampling in order to additionally improve the efficiency of the method. At this stage we need to understand how the code intervenes on the size and the dynamical evolution of the ellipsoids. We recall the definition of the *dynamical enlargement* f of the ellipsoids (equivalently the total enlargement fraction, or simply the enlargement fraction), provided by [3] and then also adopted in DIAMONDS (see [7]). For a given nested iteration and for a single ellipsoid we have:

$$f\left(f_0, \alpha, N_{\text{live}}, n\right) = f_0 X^{\alpha} \sqrt{\frac{N_{\text{live}}}{n}} \tag{2.1}$$

where we made explicit its dependence upon the parameters f_0, the initial enlargement fraction, the shrinking rate α of the remaining prior volume X, the number of live (or active) points N_{live} used in the nested sampling, and the total number of live points belonging to the ellipsoid considered, n, as extracted by the clustering algorithm. The parameter n in particular depends in turn on N_{live} and N_{clust}, as we will see in detail in Section 2.5. These represent the parameters that require to be configured and that are given as input to the code.

Equation 2.1 is crucial for understanding how f works, and especially how f_0 and α should be tuned by the user, so it is important to have it clear in mind for the remainder of this user guide. The enlargement fraction is applied to the length of each principle axis[1] (or principle semi-axis) of the ellipsoid according to $\lambda_{\text{enlarg}} = \lambda_0(1 + f)$, where λ_0 is the default length of a principle axis, and λ_{enlarg} is the length of the principle axis of the enlarged ellipsoid. In Figure 2.2 we show the effect produced by an enlargement fraction $f = 0.5$ on a 3D ellipsoid, meaning that we have extended the original axis length by 50% in each dimension.

[1] In a bi-dimensional case, the principle (or principal) axes of an ellipse are certain lines that correspond to the major and minor axes of the ellipse. The principle axes are always orthogonal.

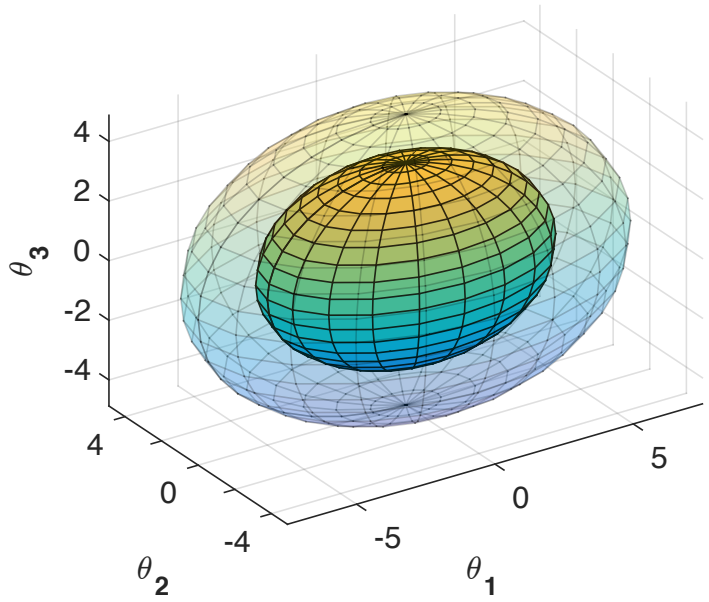


Figure 2.2: An example of a 3D ellipsoid, with initial size shown by the shaded surface in the center of the plot, to which an enlargement fraction of $f = 0.5$ was applied. The enlarged ellipsoid, shown as the transparent shaded surface around the default one, has semi-axes enlarged by 50% with respect to the original length. The three coordinates $\theta_1, \theta_2, \theta_3$ represent three arbitrary free parameters whose prior PDFs set the size of the parameter space in which the sampling takes place.

To proceed we provide more insights about all the configuring parameters involved in the nested sampling based on the multi ellipsoidal sampler. In reference to the sample code shown in Section 1.6, for most of the applications we can consider that the number of live points, represented by the integer variable `initialNlivePoints`, is not subject to any reduction law of the live points, as those presented by [4] and [7], meaning that we can adopt the condition `minNlivePoints` = `initialNlivePoints`. In addition, from now on we will consider that the variables

- `maxNdrawAttempts` (M_{attempts})
- `NinitialIterationsWithoutClustering` (M_{init})
- `NiterationsWithSameClustering` (M_{same})
- `terminationFactor` (δ_{final})

are set to the values $M_{\text{attempts}} = 10^4$, $M_{\text{init}} = N_{\text{live}}$, $M_{\text{same}} = 50$, $\delta_{\text{final}} = 0.01$, following the indications provided by means of the testing done by [7]. These configuring parameters will not be further discussed because either they are known to have a fixed value or their tuning is trivial, thus not representing a crucial point for the success of the computation.

2.1 The number of clusters N_{clust}

This number, provided as a minimum and maximum allowed as shown in Section 1.5, is not explicitly appearing in Equation 2.1 but it has a direct impact on the number of live points

falling within each ellipsoid, which we indicated as n (see Section 2.5 for more details). As already discussed by [7], and then additionally tested by [8], setting $1 \leq N_{\rm clust} \leq 10$ offers a good compromise between efficiency of the sampling of the parameter space for complex shapes of the likelihood distribution (e.g. pronounced degeneracies, multi-modality, etc.) and the computational time, in a wide variety of applications. This range can of course change depending on the specific problem, although a maximum number of clusters larger than 10 is usually not recommended. When it is possible, e.g. for problems in which no strong degeneracies are expected or the solution is uni-modal, it is useful to reduce the maximum number allowed down to e.g. 6 or even 4. This is because the more clusters are computed the more the nested sampling slows down, causing the cluster algorithm to become the bottleneck of the whole process in cases where the number of dimensions of the inference problem grows up considerably (e.g. > 50). Sometimes a large number of clusters allowed is not necessarily improving things and one should avoid stretching the upper bound beyond what is really needed.

Given the problem in exam is fixed, it is suggested not to change the range of $N_{\rm clust}$ because it may have a strong impact on Equation 2.1, hence on the other configuring parameters of DIAMONDS, as we shall discuss thoroughly in Section 2.5. In the remainder part of the guide we will assume that this range is constant, unless specified otherwise.

2.2 The number of live points $N_{\rm live}$

This number is represented by the integer variable `initialNlivePoints` used in the sample code shown in Section 1.6, and it is subject to change depending on the complexity of the likelihood distribution in exam, which is mainly related to multi-modality and curving degeneracies. When the problem at hand is the same in multiple analyses (e.g. the fitting of the stellar oscillations in a sample of red giant stars), there is no particular reason to change $N_{\rm live}$ since we may expect the likelihood distribution to remain approximately of the same type. Some values of $N_{\rm live}$ are provided by the authors in [7, 9, 8], and can of course be adopted for other computations involving similar inference problems. Values ranging from 500 to 2000 should guarantee an optimal sampling for a variety of applications spanning from uni-modal distributions, even with degeneracies and in dimensions from a few up to about one hundred, to highly multi-modal distributions up to at least $\sim$20 different modes in $\sim$10 dimensions, as demonstrated by [7], Section 6.7. The tuning of $N_{\rm live}$ is in general not particularly difficult and critical since even by changing its value by several hundred units the other configuring parameters should not be affected. This aspect is further discussed in Section 2.5, which we recommend to read.

As a general advise, the number of live points can be increased if for example:

- A larger (and/or higher-precision) dataset is used (e.g. NASA's *Kepler* datasets spanning more than one year or solar time-series obtained by GOLF and VIRGO), which will cause the maxima of the likelihood distribution to become highly localized (a condition usually termed as *strong data*, see also [16]), hence requiring a finer sampling to detect them properly. In this case up to 1000 live points are recommended.

- More complex likelihood distributions (e.g. with multiple modes or pronounced degeneracies or both of them) are expected. To guarantee a better sampling of the posterior distribution at each nested iteration and be able to detect all (or most of) the modes from the early stages, a number of 2000 live points could be needed (e.g. see [8], Section 6.7).

- The number of dimensions in the problem is subject to vary up to two orders of magnitude

(from ∼1 to ∼100 dimensions). In this situation having more live points guarantees the initial sampling of the parameter space to be enough dense to detect the different maxima of the likelihood distribution even if the considered prior volume is increasing. We suggest values between 1000 and 2000 for similar applications.

A number of live points larger than those mentioned here will likely slow down the computation without a significant gain in accuracy for both the sampling of the local maxima and the evaluation of the final Bayesian evidence, as we shall see more in detail below. Conversely, if $N_{\rm live}$ is too low, we will not be able to retrieve good results because the chances to miss the modes of the likelihood distribution increase (we do not have enough live points to sample adequately all the parameter space and understand which regions should be sampled more than others). In the worst case it may also happen that a computational failure is produced, of the types discussed in Sections 3.3 and 3.4, since the initial sampling provided is too sparse to allow the algorithm to find better likelihood points at later stages of the process.

In Figure 2.3 we show two important correlations that highlight the performances of the DIAMONDS code. For this purpose we used the values that were obtained from a set of 152 independent computations made by [8] for the peak bagging analysis of a sample of red giant stars. In this analysis the basic type of model is always the same but the datasets considered change every time (although presenting similar characteristics), as well as the number of dimensions involved in the model, hence in the Bayesian inference that is performed (see also Section 4.5 for more discussion). This study represents an interesting benchmark to investigate and understand the impact of the number of dimensions on the efficiency and computational speed of DIAMONDS.

The top panel is related to the number of nested iterations (or equivalently the number of sampling points, or the likelihood evaluations[2]), $N_{\rm nest}$, which is plotted as a function of the number of dimensions k involved in the analysis. Each one of the points displayed corresponds to a different number of dimensions and it was computed by averaging $N_{\rm nest}$ from all the available computations provided by [8] having the same dimensionality, thus resulting in a total of 54 independent measurements. In particular, we notice the presence of two distinct groups of points, which we marked with different colors. These groups correspond to a different choice of the parameter $N_{\rm live}$, whose value is doubled from one set to another (11 measurements for $N_{\rm live} = 500$ in blue and 43 measurements for $N_{\rm live} = 1000$ in purple). We made this distinction to explicitly quantify the impact that the number of live points has on the nested sampling algorithm implemented in DIAMONDS. We can see a quite clear correlation between $N_{\rm nest}$ and k in both cases. The solid lines provide a power law fit to each set of points, yielding

$$N_{\rm nest}^{(500)} = (1369 \pm 254) \cdot k^{0.57\pm0.06} \tag{2.2}$$

$$N_{\rm nest}^{(1000)} = (2737 \pm 432) \cdot k^{0.58\pm0.05} \tag{2.3}$$

for 500 and 1000 live points, respectively. The exponent of the power law, roughly equal to 3/5, is essentially unchanged in the two cases, as one would expect. This is because the way the nested sampling algorithm is sensitive to the number of dimensions is intimately related to the working principle of the drawing mechanism (SES and X-means in this case) and is thus independent of the number of live points being used, provided that they are numerous enough to properly detect the modes of the likelihood distribution in exam. We therefore show that

[2]We consider one likelihood evaluation as the evaluation of the likelihood function for a drawn point that satisfies the hard constraint in likelihood imposed by the nested sampling. In practice, however, many more likelihood evaluations are done during the drawing phase because not all the points are accepted and the process can be repeated up to $M_{\rm attempts}$ different times for each nested iteration to be completed, see also the discussion at the beginning of this chapter.

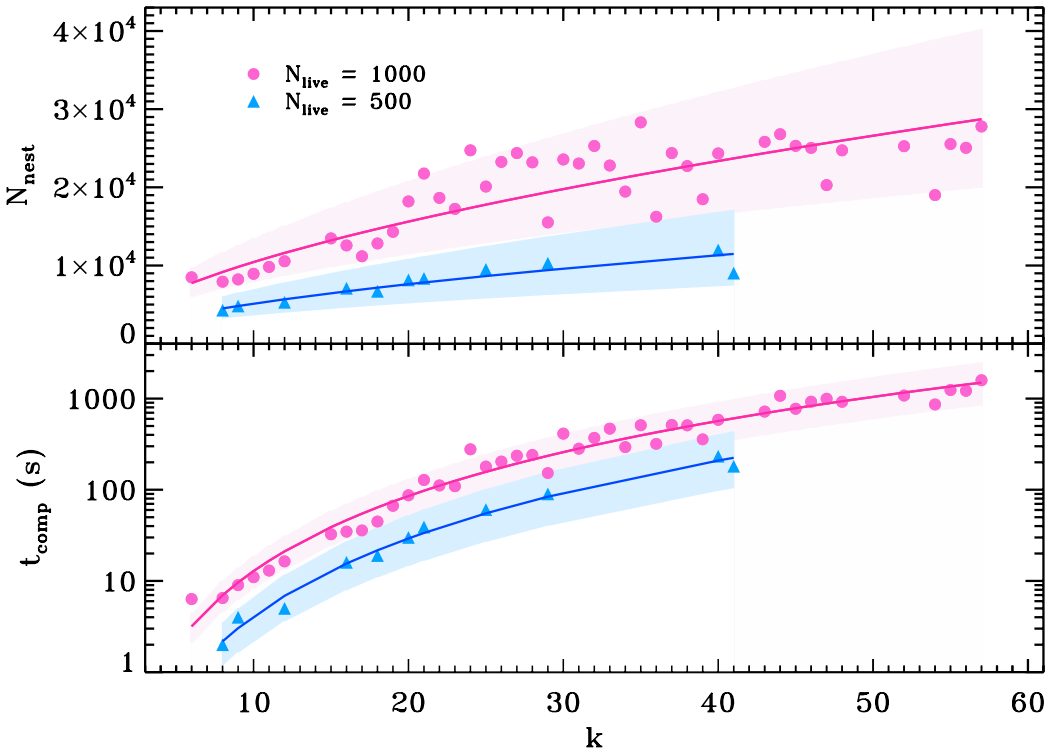


Figure 2.3: *Top panel*: the average number of nested iterations (or likelihood evaluations) $N_{\rm nest}$ as a function of the number of dimensions involved in the inference problem, k, showing that DIAMONDS has $N_{\rm nest} \sim \mathcal{O}(k^{3/5})$. The values plotted are 54 average measurements obtained from 152 independent computations provided by [8]. The different colors and symbols represent values corresponding to computations where we adopted $N_{\rm live} = 500$ (blue triangles) and $N_{\rm live} = 1000$ (purple circles). The colored bands in the same tonality show the 1-σ confidence regions for the power law fits (thick solid lines). *Bottom panel*: similar to the top panel but for the average computational time (in seconds), $t_{\rm comp}$, yielding $t_{\rm comp} \sim \mathcal{O}(k^{2.8})$. For a better visualization a logarithmic scale in the y-axis was adopted.

DIAMONDS has $N_{\rm nest} \sim \mathcal{O}(k^{3/5})$, at least for the benchmark considered. The proportionality term of the power law is instead doubled by doubling $N_{\rm live}$, meaning that for a fixed dimension the number of nested iterations increases linearly with the number of live points, or that equivalently $N_{\rm nest} \sim \mathcal{O}(N_{\rm live})$. This result is also in agreement with our expectations. The reason is that every time we start a new process, the initial prior volume (usually termed X as shown in Equation 2.1) is subdivided into different portions (namely shells of prior mass identified by iso-likelihood contours, see [15, 14]) that we inspect by moving from one another by reducing prior mass at each nested iteration by a factor $\sim \exp(-i/N_{\rm live})$, where i is the nested iteration considered. This means that the smaller is the number of live points, the larger will be the reduction in prior mass that we apply at every step, hence fewer nested iterations are needed to move from cover the entire prior volume (from $X = 1$ to $X = 0$). Since the algorithm sequentially samples a new live point by moving from one shell to another having a better likelihood constraint, when doubling the number of live points we also double the number of portions in which we subdivide the total prior volume, meaning that the algorithm takes twice the number of nested iterations to reach the same final level in likelihood.

In the second plot appearing in the bottom panel of Figure 2.3 we consider the compu-

tational time, $t_{\rm comp}$, as a function of the number of dimensions, for the same set of points presented in the top panel. We find another correlation, with power law fits

$$t_{\rm comp}^{(500)} = (0.006 \pm 0.002) \cdot k^{2.73\pm0.07} \tag{2.4}$$

$$t_{\rm comp}^{(1000)} = (0.024 \pm 0.006) \cdot k^{2.84\pm0.11} \tag{2.5}$$

for 500 and 1000 live points, respectively[3]. For the same reasons discussed before, the exponent of the power law is not changing significantly with varying $N_{\rm live}$, thus yielding the result that $t_{\rm comp} \sim \mathcal{O}(k^{2.8})$. However, this time when doubling the number of live points the proportionality term becomes about four times larger, meaning that $t_{\rm comp} \sim \mathcal{O}(N_{\rm live}^2)$. This result shows that increasing the number of live points can significantly slow down the computation and thus motivates the suggestions made at the beginning of this section. Increase $N_{\rm live}$ only if it is really necessary! Besides, the spread observed in all the relations and quantified by the shaded regions overlaid in the plots, is arising because we are using different *real* datasets for every computation, causing the likelihood distributions to always change from one analysis to another, though not significantly. We also note that the average computational time shown in Figure 2.3 is only related to the specific application of the peak bagging for the sample of stars analyzed by [8], and has not to be taken as an absolute reference to understand how fast is the code for a given dimension. The computational time will change every time we change application because it depends on several aspects such as the length of the dataset, the complexity of the likelihood distribution, the configuration of the code parameters, the evolution of the nested sampling algorithm, and of course the available computational power.

Finally we remind that using a different $N_{\rm live}$ has an impact on the estimation of the final Bayesian evidence because the number of nested iterations changes and consequently also the total number of points sampling the posterior distribution (see also [11] for more details). This could cause a variation of the termination condition needed to stop the nested sampling and it suggests that when performing a Bayesian model comparison, the different computations should be carried out by having exactly the same number of live points in order to be able to compare models for which a consistent sampling method has been applied. Based on our discussion and given that the type of Bayesian inference problem is fixed, we shall keep $N_{\rm live}$ as constant in the different computations, even if the datasets are being changed[4].

2.3 The initial enlargement fraction f_0

Specified as the variable `initialEnlargementFraction` in the sample code shown in Section 1.6, this is certainly the most important and most frequently tuned configuring parameter of the ellipsoidal sampler. The reason is that it allows us to have an immediate control on the size of the ellipsoids. According to Equation 2.1 this is done by changing each principle axis length of the ellipsoid by the same fraction, as already shown at the beginning of this chapter. We therefore have that $f_0 \geq 0$, where $f_0 = 0$ means that no enlargement is applied (but never $f_0 < 0$, as this may lead to a negative axis length!). Typical values of this parameter stay on the order of the units and do not change significantly (they will stay within the same order of

[3] These times refer to a 2.7 GHz CPU and dataset lengths on the order of 2000 bins for each computation.

[4] We are assuming that the datasets in use roughly remain of the same type, meaning that for example we could analyze one star by using a time-series spanning one year of observation, and then move to another star with the same or at least similar length of observation as the previous one, although the information contained is clearly different. In situations where we decide to completely change the nature of the datasets, e.g. by adopting a new dataset with one order of magnitude more data bins then the first one, then attention has to be paid on what number of live points should be used, and possibly increase it following the suggestions given in this section.

magnitude). The analyses published by [7, 9, 8] show a variety of applications by providing the related values of f_0.

There is an optimal number of f_0 for each problem considered. The general rule is that on one side if f_0 is too large, the sampling efficiency decreases and the computation slows down, or it may even fail in the worst case. This is because very big ellipsoids may happen to be larger than the parameter space itself, which is pointless, hence disrupting any possible improvement that can be obtained by reducing the useful prior volume to draw from (see [7]). On the other side, if f_0 is too small lot of information is lost already from the first nested iterations because as seen before, the default size of the ellipsoids is not enough to properly include all or at least most of the live points it is originating from. As a result the final sampling will likely lead to wrong results because too small ellipsoids sample the posterior distribution only partially. The way to understand whether f_0 is correct is thoroughly discussed in the coming Chapters 3 and 4. When setting f_0 we usually deal with two different scenarios:

1. We fix the model and the number of free parameters (the dimensions of the problem), and perform a Bayesian inference on different datasets. In this case there is no particular reason to vary f_0 since we expect the problem to behave in a similar way each time. However, Section 2.5 discusses a particular case in which a significant change in the parameter n may cause f_0 to be readjusted.

2. We have a single dataset but this time we use different models, with a different number of free parameters involved. For this situation we need to tune f_0 by using steps of 0.1 or 0.2 each time.

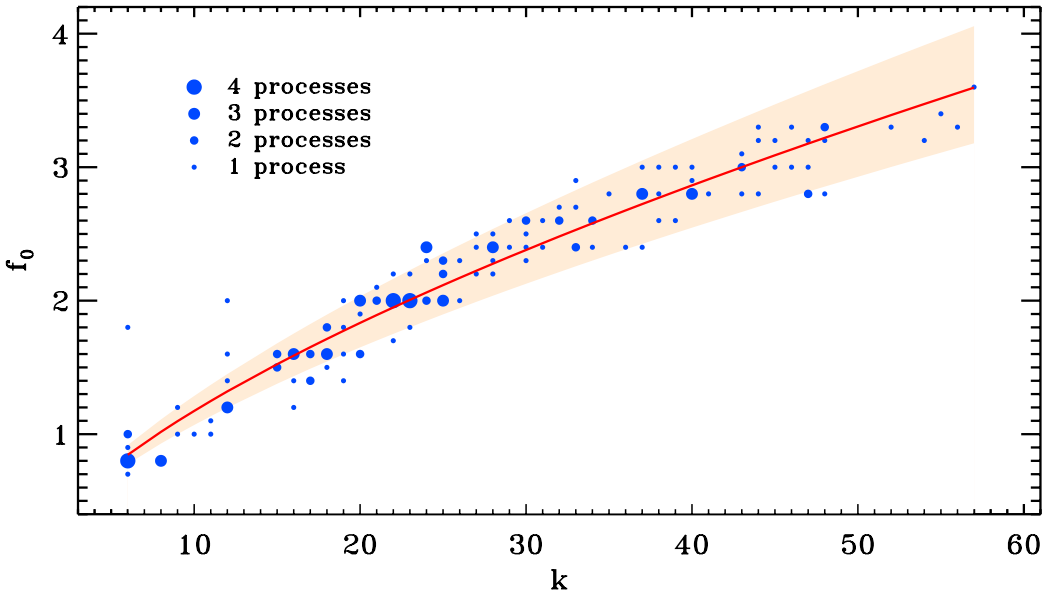


Figure 2.4: The initial enlargement fraction f_0 as a function of the number of dimensions involved in the inference problem, k. The 152 independent computations provided by [8] used 4 clusters each to sample the parameter space. The size of the circles is proportional to the number of processes for which the same f_0 was used. The colored band shows the 1-σ confidence region for the power law fit (thick red line). This relation shows us that $f_0 \sim \mathcal{O}(k^{2/3})$.

Figure 2.4 shows the evolution of f_0 as a function of the number of dimensions involved in the inference problem, k, for the same set of 152 independent computations used in Section 2.2.

The number of clusters involved, allowed to vary in the range $1 \leq N_{\text{clust}} \leq 4$, was $N_{\text{clust}} = 4$ for all the computations considered. By considering all the individual measurements, a sub-linear proportionality between the two quantities appears evident. By fitting a simple power law relation, we obtain

$$f_0 = (0.267 \pm 0.014) \cdot k^{0.643 \pm 0.017} \,. \tag{2.6}$$

The exponent to the number of dimensions is roughly equal to 2/3 meaning that at least for this application $f_0 \sim \mathcal{O}(k^{2/3})$. This relation, which is ensured to hold in the range of dimensions shown in the plot, can be very useful if adopted as a guideline to tune f_0 with a varying number of dimensions[5]. In situations where the number of cluster varies from one computation to another, one can apply the correction to the proportionality term discussed in Section 2.5 and given by Equation 2.9. We also notice that based on our findings, as expected f_0 is not changing when N_{live} is different, as it is instead happening for the number of nested iterations and the computational time that we discussed in Section 2.2. This is explained by Equation 2.1 itself, which is not changing f by varying N_{live} alone since the quantity $\sqrt{N_{\text{live}}/n}$ is on average only depending on the number of clusters, as we shall point out in Section 2.5. Such result is also intuitively correct if we consider that varying N_{live} is not changing the overall efficiency of the nested sampling, as shown in Section 2.2, provided that N_{live} is sufficiently large to ensure all the likelihood maxima to be identified.

2.4 The shrinking rate α

The parameter `shrinkingRate` of the ellipsoidal sampler represents a sensible information for the ellipsoids because it controls the dynamics of the enlargement fraction during the evolution of the nested sampling. It is always a number so that $0 \leq \alpha \leq 1$, and never $\alpha < 0$ since we do not want to enlarge the ellipsoids when we shrink the prior mass in use! For all the real applications in which DIAMONDS was used, we found α lying in the range $0.01 \leq \alpha \leq 0.04$ only, where steps of 0.01 were adopted to tune its value. In the analyses done up to now, we did not find α correlating with any other parameter such as the number of dimensions or the number of clusters involved, hence also with f_0 itself. However, it is important to note that even a variation of 0.01 could in some cases lead to significant changes both in the computational speed, especially for stages in which the computation approaches the end, and in the resulting marginal distributions of the inferred parameters, as it will be discussed later in Chapter 4. This parameter has a strong impact on evolution of the process and therefore it should be chosen with care.

Similarly to f_0, each process has its optimal value of shrinking rate. A useful guideline is that on one hand if this parameter is too large, meaning that the ellipsoids shrink very quickly, one risks to lose sensible information by sampling too small regions of the posterior distribution by the time the nested sampling reaches the termination condition. Conversely, if α is too small, meaning that the ellipsoids shrink very slowly, the parameter may cause the computation to drop both in speed and efficiency because we are not optimizing anymore for the prior volume of the sampling, especially during the final nested iterations where more focus on the likelihood maxima is required[6]. In the worst case, this could lead to a premature interruption of the process, as it will be discussed in Chapter 3 more in detail. Finally, to see the effect of α coupled with that of f_0 we recommended reading Section 4.1.

[5] This relation is calibrated with an application involving $N_{\text{clust}} = 4$. In principle, when a different number of clusters is used the relation should hold too, but we propose to accurately verify this by the time new computations will be available.

[6] See also the explanation presented at the beginning of this chapter, and in Figures 2.1 and 2.2.

2.5 How $N_{\rm clust}$ can change the enlargement fraction f

One last discussion to face, but not less important than the others, is related to the way the number of clusters involved in the sampling impacts the enlargement fraction of the ellipsoids. We recall from Equation 2.1 and Section 2.1 that the number of clusters $N_{\rm clust}$ is not explicitly appearing in the definition of the enlargement fraction but its dependance is hidden in the parameter n, the number of live points falling in an ellipsoid. It is through the quantity $\sqrt{N_{\rm live}/n}$ of Equation 2.1 that $N_{\rm clust}$ has a direct influence on the size of the individual ellipsoids and we shall understand how. This quantity represents the error on the eigenvalues stemming from the covariance matrix of a finite sample of points, which is $\propto 1/\sqrt{n}$, and it is normalized by the total number of live points used, as discussed by [3]. What happens in practice is that the more clusters are used during the computation, the lower will be, on average, the number of live points that can be contained within each ellipsoid, because the same amount of live points, $N_{\rm live}$, has to be split among more clusters. The parameter n becomes smaller because statistically we have that its average is given as $\langle n\rangle = N_{\rm live}/N_{\rm clust}$, hence resulting in a net additional enlargement to the ellipsoids[7] aimed at encompassing a larger degree of uncertainty in the corresponding axis lengths.

It is important to notice that changing $N_{\rm live}$ alone (by an amount on the order of several hundreds units) is not necessarily changing n since it is the number of identified clusters that matters. This means that even by changing $N_{\rm live}$ we are not changing the enlargement fraction, as also demonstrated with the computations shown in Section 2.3. We can understand that $N_{\rm clust}$ is in principle independent of $N_{\rm live}$ because it mainly relies on the shape of the likelihood distribution and on the way the nested sampling evolves (unless the live points are so sparse and low in number that the initial sampling is completely wrong). In addition to the numerical result discussed at the end of Section 2.3, we can statistically prove that the total enlargement fraction is not influenced by the number of live points by simply replacing the parameter n from Equation 2.1 with its averaged value $\langle n\rangle$, already introduced before, yielding the average enlargement fraction

$$\langle f\rangle = f_0 X^{\alpha}\sqrt{N_{\rm clust}}\,, \tag{2.7}$$

in which the apparent dependence on $N_{\rm live}$ has now disappeared.

To show a clarifying example, consider the case in which we want to analyze the granulation signal in the Fourier domain of several stars, meaning that we keep the fitting model fixed, hence the number of dimensions involved, but we change the dataset for each new analysis. Eventually it may happen that the number of clusters involved during the computation changes from one star to another. Now suppose we have tuned the configuring parameters of DIAMONDS for a first computation in which 4 clusters are being used. If we use the same set of configuring parameters for a new analysis but this time the number of clusters increases by, say, 4 (meaning a total of 8 clusters), this will imply enlargement fractions significantly larger ($\sqrt{2}$ times in the given example) than those calibrated at the beginning. The reason is that the parameter n will be on average $\sqrt{2}$ times smaller than in the first case because the number of clusters is now doubled but the number of live points is still the same.

[7] The average of n is considered over the number of clusters (or equivalently ellipsoids) involved at a certain iteration of the nested sampling. By knowing that $N_{\rm live}$ is constant, if we have, say, p different clusters identified, hence $N_{\rm clust} = p$, each one containing a number of live points n_i, with $i = 1, 2, \ldots, p$, it must be that $N_{\rm live} \equiv n_1 + n_2 + \ldots n_p$ by definition. From the simple arithmetic average we thus have that

$$\langle n\rangle = \frac{n_1 + n_2 + \ldots n_p}{p} = \frac{N_{\rm live}}{N_{\rm clust}}\,.$$

To formally quantify this effect, we shall consider two different numbers of clusters involved in two separate computations, $N^{(1)}_{\mathrm{clust}}$ and $N^{(2)}_{\mathrm{clust}}$. The relative difference in the two corresponding average enlargement fractions $\langle f^{(1)} \rangle$ and $\langle f^{(2)} \rangle$, as from Equation 2.7, thus reads

$$\frac{\langle f^{(2)} \rangle}{\langle f^{(1)} \rangle} = \sqrt{\frac{N^{(2)}_{\mathrm{clust}}}{N^{(1)}_{\mathrm{clust}}}} \,. \tag{2.8}$$

As highlighted in the example above, this aspect becomes relevant if we plan to extend the same analysis to multiple datasets by using a single set of configuring parameters (e.g. those calibrated for a single dataset that is a good representative of all the others, see also Section 4.5). Then, this problem should be taken into account if we notice that by using the same configuring parameters for a new computation either the marginal distributions that are computed appear affected by one of the issues presented in Chapter 4 or the nested sampling itself stops prematurely, as described in Chapter 3. This means that with the new computation (i.e. with the analysis of another dataset) a significant change in the number of clusters being used must have occurred, typically on the order of 4-6 clusters. A way to quickly recognize a possible change in the number of clusters is to check the output numbers printed on the screen during the computation, as shown in Section 1.7.

For correcting this undesired behavior of the enlargement fraction consider again that we have fixed the configuring parameters to an optimal set found for our first computation. Now assume we use the same f_0 of the first run in a second one, where the number of clusters is changing as shown before from $N^{(1)}_{\mathrm{clust}}$ to $N^{(2)}_{\mathrm{clust}}$. By considering Equation 2.8, one can rescale the initial enlargement fraction f_0 of the second computation accordingly, yielding the corrected value

$$f_{0,\mathrm{corr}} = f_0 \sqrt{\frac{N^{(1)}_{\mathrm{clust}}}{N^{(2)}_{\mathrm{clust}}}} \,. \tag{2.9}$$

To explain this relation, we assume that the new computation, leading by default to a new enlargement fraction $f^{(2)}$ as from Equation 2.1, has used e.g. a higher number of clusters than the first one for which we calibrated the configuring paramters. Then, the condition $N^{(2)}_{\mathrm{clust}} > N^{(1)}_{\mathrm{clust}}$ will produce a value of $f^{(2)}$ that is larger than $f^{(1)}$ on average by a factor given by Equation 2.8. As we learned at the beginning of this chapter, we cannot change f in a direct way but only intervene on the corresponding input f_0. To readjust $f^{(2)}$, which is now off from its optimal value, and change it back to approximately the former value $f^{(1)}$, which we know to be working fine for the given problem, we need to evaluate $f_{0,\mathrm{corr}}$ by dividing our input f_0 (the same in both computations) by the same factor for which $f^{(2)}$ has increased, easily retrievable by our knowledge of the difference in the number of clusters. According to the example given, we will have that $f_{0,\mathrm{corr}} < f_0$ since we want to compensate for a total enlargement fraction that is larger than expected.

However, this problem is not very common and can easily be addressed either by readjusting the enlargement fraction following the recipes presented in this manual (first taking into account the effect described in this section and eventually tuning it according to what discussed in Chapter 4), or by restricting the allowed range in N_{clust} to avoid two different computations to end up using a significantly different number (e.g. by using a range $4 \leq N_{\mathrm{clust}} \leq 6$, instead of the wider $1 \leq N_{\mathrm{clust}} \leq 10$, if we know that in general the problem requires around 5 clusters only).

To conclude this chapter, it is useful to comment about the methodology proposed in this section by asking ourselves the question: "Why do we incorporate the term $\sqrt{N_{\mathrm{clust}}}$ in

the average dynamical enlargement of Equation 2.7 (or equivalently the quantity $\sqrt{N_{\text{live}}/n}$ in Equation 2.1) if we eventually need to correct for it when the number of clusters is changing from one computation to another?". To answer this question, we have to consider that this aspect is only useful for the specific case of applications that are in principle identical (or at least of the same type), apart for the different datasets in use. In such kind of applications we have no particular reasons to believe that a difference in the number of clusters being used could be supported by a real change in complexity of the shape of the likelihood distribution. Instead, we expect that the likelihood distribution, though it will never be the same if we change dataset, will still preserve a similar topology, proving that a change in N_{clust} is more likely to be the only result of a different initial distribution and subsequent evolution of the live points used by the nested sampling algorithm. For general applications the term $\sqrt{N_{\text{clust}}}$ is a desired quantity, useful to produce more reliable ellipsoid enlargements, as discussed earlier in this section. Therefore, the correction to the initial enlargement factor given by Equation 2.9 is meaningful simply because we have first calibrated f_0 by means of the original definition of the dynamical enlargement, Equation 2.1, which implicitly incorporates the number of clusters being used.

Now that the basic elements to set up a working flow have been provided, and that we have understood what is the role of the enlargement fraction of the ellipsoids and that of the configuring parameters of the code, in the upcoming chapter we will focus on the incomplete computations that do not allow us to retrieve the final outputs from the Bayesian inference. Lastly we will explain how to inspect the results when the process is completed without errors, thus describing the assessment of their reliability.

3 Tackling incomplete computations

The DIAMONDS code implements a nested sampling algorithm, which as any other Monte Carlo algorithm is subject to the characteristic that a process and its evolution with time will be different every time we execute a new one [15, 14]. The randomness taking place in both the way the live points are distributed at the first iteration and the way a new live point is drawn from the prior PDF at every nested iteration, can sometimes lead to samplings that cause the computation to fail, just accidentally. A general advise is to try relaunching the code if a failure of any of the types discussed in Sections 3.4, 3.3, 3.1, occurs, but only if this is happening two or three times in a row, at most.

Based on our adoption of the Multi Ellipsoidal Sampler, we can classify a few main cases that do not allow us to retrieve the desired final results. In this section we will describe each of them in detail by sorting them by increasing severity and complexity, thus explaining what is their possible origin and how to troubleshoot. It is useful to bear in mind that there are two general reasons at the base of the computational failures related to the sampler:

- Wrong choice of the size of the ellipsoids and/or of their dynamical evolution.
- Wrong choice of the prior PDFs for one or more free parameters (especially in the case of uniform priors).

In addition we include a fourth case that is not linked to the sampler itself but only depends on the dataset and the implementation of the model and likelihood used in the inference analysis. We discuss this additional case in Section 3.2. From now onwards we will often talk about *optimal values* when referring to those values of the configuring parameters that allow the code to successfully complete a computation and produce reliable results. We stress that all the troubleshooting provided here is not guaranteed to work at all times and for any application since there is no precise and absolute recipe for a Monte Carlo method. The descriptions and documentation provided in this chapter are based on the applications where DIAMONDS was used, the detailed knowledge of the working mechanism of the sampling algorithm and its implementation, and the use of some logic reasoning.

Documentation on sampling methods different than the ellipsoidal sampler is not part of this user guide. To help us improving this manual for the future, we encourage you to send your feedback and share the experience gained in testing the code for any other application, directly to emncorsaro@gmail.com.

3.1 Assertion failure

This is the most common error that one can encounter. It is not difficult to handle but it requires some investigation. Although often related with the ellipsoidal sampler, an assertion failure is not an internal error of the sampler itself but it originates directly from the Eigen

library adopted by DIAMONDS due to some wrong array element indexing. It can display as in the following examples

```
Assertion failed: (index >= 0 && index < size()), function operator(), file
/localPath/Diamonds/include/Eigen/src/Core/DenseCoeffsBase.h, line 394.
```

or

```
Assertion failed: (((SizeAtCompileTime == Dynamic && (MaxSizeAtCompileTime==Dynamic ||
size<=MaxSizeAtCompileTime)) || SizeAtCompileTime == size) && size>=0), function resize,
file /localPath/Diamonds/include/Eigen/src/Core/PlainObjectBase.h, line 262.
```

and only arises during the computation of the Bayesian estimates from the MPDs in the `Results` class. This is the basic type of error and the least severe type among those discussed in this chapter. It could appear after the nested sampling has reached the imposed termination condition δ_{final}, in which case it is the only error showing up, or because the process was terminated prematurely with one of the errors presented in Sections 3.2, 3.3, and 3.4, hence in combination with other errors. This assertion failure implies that despite all the output files of DIAMONDS that contain the original sampling are created, neither parameter estimates nor all the marginal distributions could be computed, meaning that the file `parameterSummary.txt` and the files `marginalDistribution<parameter#>.txt` for one or more free parameters are not generated (see Section 1.7 for more information on the output files).

We suggest that if this computational error appears for the first time, one relaunches the code at least one or two times to understand whether it is only a problem of a bad sampling or not, unless the assertion failure is caused by following the error discussed in Section 3.2. If the problem persists then we should stick to what explained in the following. Since we cannot use the MPDs, tackling this issue requires the inspection of the sampling of the individual free parameters that we are trying to estimate. To do this we have to plot the values of all the output files `parameter<parameter#>.txt`, which we have available in the directory containing the results, containing the sampling of the individual free parameters as a function of the nested iterations. Figure 3.1 shows an example of plotting this information for three different free parameters, as derived from the computations performed in the peak bagging analysis described by [8]. The blue dots are the sampling with the standard evolution of the process starting from iteration $M_{\text{init}} = 1000$[1]. The y-axis in all panels shows the allowed range of values for each free parameter as set by the uniform prior PDFs. For the parameter θ_1 in the top panel we have an ideal case where the sampling coming about the end of the process is lying in the middle of the adopted prior range. The sampling appears also symmetric around the final estimate of the parameter, a fortunate case in which no strong correlations with other free parameters appear to be present.

Conversely to the first example, the sampling of the parameter θ_2, displayed in the middle panel, converges to a region close to the edge of its prior boundary. Following this example, a more enhanced situation in which the sampling directly hits the edge of the allowed parameter range could potentially lead to an assertion failure. To explain this, it can be useful to complement the reading with Section 4.4, where we deal with a MPD falling at the edge of our prior interval. On one hand, given that the MPD was already computed, when the algorithm implemented in the function `writeParametersSummaryToFile()` of the `Results` class

[1] We discarded the first M_{init} points because they do not provide any sensible information on the parameter estimation.

attempts to evaluate the estimators and the Bayesian credible intervals it may happen that the estimators and/or the Bayesian credible limits fall outside the range of allowed parameter values if the MPD is trimmed at one extreme of the prior boundary. The consequence is that the output file containing the MPD (`marginalDistribution<parameter#>.txt`) is created, but not that containing the parameter estimates (`parameterSummary.txt`). This suggests that we are missing the likelihood maximum just by a small extent of parameter range and we therefore need to adjust the prior boundaries accordingly[2] by either shifting the limiting values or extending them. If the prior range cannot be enlarged along the side that is causing the problem[3], one could change the scale of the free parameter into a logarithmic one that would help us obtaining a type of evolution similar to that of the parameter θ_1, which is the most preferred condition. This transforms the uniform prior on θ_2 in a uniform prior in $\ln \theta_2$, now corresponding to a Jeffreys' prior in θ_2, namely $\propto \theta_2^{-1}$ (see also [10, 6]). On the other hand, in a more problematic occurrence it could happen that the MPD itself cannot be computed because the sharp trimming is not even allowing the parameter estimation algorithm to perform the cubic interpolation over the re-binned set of marginal probabilities. As a consequence, both the files containing the MPD and the parameter estimators will not be produced. Similarly to what explained before, we need to intervene on adjusting the prior ranges also this time.

Lastly the bottom panel, representing the sampling evolution of the parameter θ_3, is the typical case where outcome is showing a wide MPD as compared to the extent of the parameter interval, hence it is a potential risky situation for computational assertion failures. This is again because a significant portion of the MPD may fall outside the prior boundaries (and this time being trimmed at both sides), thus preventing us from computing all the Bayesian estimators and possibly the MPD itself. To solve the issue it is necessary to enlarge the prior volume (i.e. the interval) considered, at least for the free parameter that is affected by this problem.

We stress that what discussed up to now is mainly related to the adoption of uniform priors. An assertion failure may not appear if a Gaussian or a super Gaussian prior PDF is adopted, where no sharp edges are taken into account. However, Gaussian and super Gaussian priors are computationally less efficient than the uniform priors, as pointed out by [7], and care is required in setting the proper widths of the distributions. Such kind of non-limited range priors are also more prone to cause a computational failure of the type of Sections 3.3 and 3.4, since they can easily lead to a significant drop in the sampling efficiency if they are not adequately set.

Suppose now that we have already faced the phase of adjusting the prior PDFs by following what explained in the above paragraphs, and that we are finally confident about the choice done. If the assertion failure persists eventually it will be appropriate to intervene on correcting the enlargement fraction of the ellipsoids. The general advise here is to try reducing f_0 and/or increasing α by using steps of 0.1 and 0.01, respectively, for every attempt made until the assertion failure error disappears and the nested sampling is able to generate the output files

[2] It is important to stress that in a *pure (full)* Bayesian approach the choice of the prior PDFs is totally independent of the likelihood distribution and it is done before the inference analysis is performed, based solely on our knowledge *a priori* of the problem. In many applications, however, we need to ensure that the likelihood maxima are sampled in their entirety because we want to exploit all the information contained in the data. Since the uniform priors require the limits of the allowed parameter range to be set, we often have to intervene *a posteriori* on their choice, meaning that we apply an *empirical* Bayesian approach. This implies that the posterior distribution is used as a guideline to improve the prior PDFs of a subsequent revised inference analysis of the same problem. We therefore need to pay attention when setting up uniform prior PDFs because they are not reflecting a simple status of ignorance about the inferred parameters, but they are instead imposing a strong limiting condition to the possible outcomes of the parameter estimation (see also [14] for more discussions).

[3] For example, in the case of the free parameter θ_2 presented we have the condition $\theta_2 > 0.00$, meaning that we cannot extend the prior boundary below 0.00.

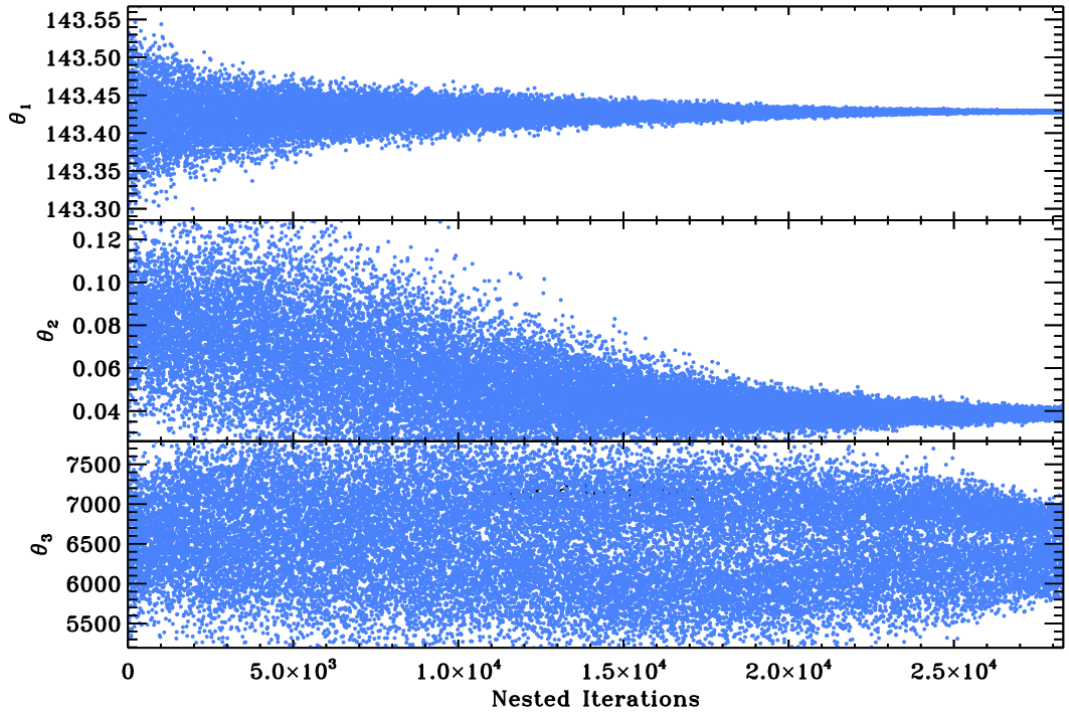


Figure 3.1: The evolution of three different free parameters of the peak bagging analysis performed by [8] with the number of nested iterations (or computational time), where uniform priors were adopted. In the top panel, the optimal case of a prior boundary that is well symmetric around the estimator of the free parameter. In the middle panel, a more critical case of an estimator evaluated close to the edge of the prior boundary, which could lead to a MPD cut at the edge of the range, as also presented in Section 4.4. In the bottom panel, the case of a sparse sampling implying either a small prior range or a large enlargement fraction, also discussed in Section 4.1.

without any other error. After the results are printed, it will be necessary inspecting them with critical sense. For this purpose we recommend the reading of the subsequent Chapter 4, and in particular of Section 4.1, which explains the effect of f_0 and α on the MPDs. This final aspect is required for understanding whether the final results are reliable or not. If we intend to use our results for other purposes, this represents an inevitable step for any inference process that was successfully completed.

3.2 Computation unable to start with NaN values

This computational error only occurs under some specific circumstances, and conversely to the others presented in this chapter, it has nothing to do with the sampling algorithm and the Eigen library. This error is rare but we still believe it is helpful to provide a description and troubleshooting for it. It may happen that right after launching the code we get the following error message:

```
Nit: 0   Ncl: 1   Nlive: 500   CPM: 0.001998   Ratio: nan   log(E): nan   IG: nan
-----------------------------------------------
 Final log(E): nan +/- nan
-----------------------------------------------
```

```
 Total Computational Time: 3 seconds
----------------------------------------------
Assertion failed: (((SizeAtCompileTime == Dynamic && (MaxSizeAtCompileTime==Dynamic
|| size<=MaxSizeAtCompileTime)) || SizeAtCompileTime == size) && size>=0), function resize,
file /localPath/Diamonds/include/Eigen/src/Core/PlainObjectBase.h, line 262.
```

hence an assertion failure just after the first nested iteration (`Nit:  0`), with `NaN` values for all the parameters related to the Bayesian evidence (namely the Bayesian evidence, its statistical error and the information gain, see also Section 1.7 for more details). This is generated by a very simple, but subtle, mistake present in the dataset. The assertion failure from the Eigen library appears just because the `Results` class is attempting to produce MPDs with only the initial set of live points (from the very first nested iteration), which does not contain any usable information. If the model we selected to fit the observations does not allow a covariate to have a zero value (e.g. because somewhere we are dividing by the covariate itself), this will lead to a `NaN` value, thus causing the program to terminate already after the first nested iteration. For example, when fitting the background model of a solar-like pulsator, or when performing a peak bagging analysis, we have to pay attention to the frequency axis in use, especially the very first element of the covariates. Since the model is based on a response function (windowing) given as a sinc^2 of the covariates, we cannot include a covariate that corresponds to a zero value of the frequency[4]. A similar mistake could be done when using a particular likelihood function in which for instance we divide by the values of the observations. If for some reasons we have zero values somewhere in the set of observations provided by the dataset, these will produce `NaN` numbers that will cause the code to crash. So the advice is, especially when fitting Fourier transform data, to always pay attention to your input dataset in relation to the model and likelihood you adopt for the inference, eventually removing any zero element that is present.

3.3 No better likelihood points found

This is a quite common computational failure that one can encounter and it is typically not difficult to troubleshoot. It can easily be identified when the following error message appears at some point on the terminal screen during the execution of the process:

```
Can't find point with a better Likelihood.
Stopping the nested sampling loop prematurely.
```

The error is produced by default by the `NestedSampler` class but originates directly from the ellipsoidal sampler. It occurs in the process of searching for a new live point that satisfies the hard constraint of having a better likelihood value than the worst of the current set of live points[5]. What happened here is that this searching — involving a drawing of a new live point from the prior PDFs for every attempt made — did yield any new solution before the maximum number of attempts (M_{attempts}) was reached. As also explained by [7], the parameter M_{attempts} is helpful to avoid very large numbers of unsuccessful attempts (over 10^5) to happen when drawing from the priors, otherwise an unacceptable bad sampling efficiency takes place. The reason why this threshold is reached depends on a combination of different factors, mainly the shape of the likelihood distribution, the dimensions of the sampling ellipsoids, our choice of the prior boundaries, and sometimes the number of live points adopted with respect to the quality of the dataset. In the following we shall try to understand how this error pops up.

[4] Even a single zero element present in a very large dataset will cause the computation to stop at the beginning.
[5] This follows from the working principle of the nested sampling algorithm (e.g. see [7], Section 3)

If the likelihood region we are sampling is rather flat, hence characterized by a low positive gradient in its values, it will be more difficult than in the case of a stronger positive gradient to quickly find a better likelihood point, implying that on average more attempts need to be made. A situation where a low gradient in likelihood is present could originate from a region of the likelihood distribution that is far from the mode (or any of the multiple modes), which can in turn also be caused by a bad choice of the prior boundaries for one or more free parameters of the fitting model. Because of the lack of a clear positive gradient (see the working principle of the nested sampling for more details, [15, 14]), flat or nearly-flat distributions in likelihood can lead to a set of live points that does not have a clear structure, and prone to form a sparse group of sampling points that are scattered over a relatively large region of the parameter space. If a cluster including such a sparse group is eventually identified by the X-means or any other clustering algorithm in use, this will produce large axis lengths from the covariance matrix of the associated ellipsoid.

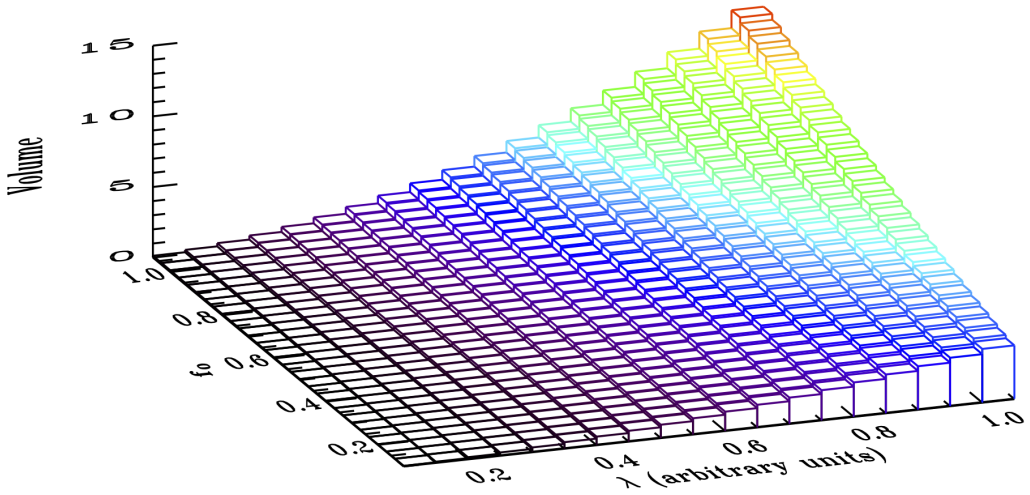


Figure 3.2: The volume of an ellipsoid with symmetric principle axes in $k = 2$ dimensions (namely a circle with $V = \pi\lambda^2$), as a function of the semi-axis length in arbitrary units, λ, and of the initial enlargement fraction f_0. The surface in colored scale shows that when by default λ is larger, the enlargement fraction produces a more enhanced effect on increasing the volume of the ellipsoid than in the case in which λ is smaller.

Let assume for simplicity that we take all the principle axes[6] of the given ellipsoid to be symmetric, i.e. to have the same length, and that we term the generic axis in one dimension as λ. The corresponding analytical volume is then given as that of an hyper-sphere, $V \propto \lambda^k$, where k is once again the number of dimensions of the problem. As shown in Figure 3.2 with colored iso-volume levels for $k = 2$, the volume of the ellipsoid increases more rapidly with f_0 when its intrinsic value[7] is larger, namely for higher values of λ (see the colored shades on the surface). The effect is even more pronounced when the dimensions increase. This means that when by default a new ellipsoid is larger than the others, the enlargement fraction will enhance its volume in comparison to that of other ellipsoids that are instead smaller. We recall that the drawing takes place each time within an ellipsoid that is selected among the

[6] It is clear that the principle axes are in number equal to the number of dimensions of the inference problem.

[7] With the term intrinsic volume we refer to the original volume when no enlargement is applied, that is for $f_0 = 0$.

others with a probability proportional to its hyper-volume[8], as defined by [3]. This significantly increases the chance of the faulty ellipsoid to be selected for the drawing process every time. Drawing repetitively from this ellipsoid, which we assumed to be localized in a region where the likelihood has a low gradient, will quickly push over the threshold the number of attempts needed to find a new point. Since the sampling volume considered is much larger than the optimal condition requires, we have a drastic fall in the sampling efficiency of the algorithm.

Nonetheless, sometimes this computational error is not necessarily depending on the presence of a flat likelihood distribution or a bad set up of the prior boundaries, but it could originate from a wrong choice of the number of live points N_{live}. As we mentioned earlier in Section 2.2 it is important to have enough live points to guarantee an initial sampling that is able to detect all the maxima of the likelihood distribution, or at least a good number of them. If we are not using enough live points for the given parameter space (especially when the number of dimensions is high, say around 100), they will result sparsely distributed. For the same reasons explained before, eventually the identified clusters will lead to very large ellipsoids and cause the sampling efficiency to drop very quickly.

To overcome this computational failure, we certainly do not want to change the likelihood distribution itself. Based on the given description and the experience about he problem, when attempting to troubleshoot this error for the first time after setting up DIAMONDS for the inference we need to take into account three sequential steps:

1. We suggest to check whether N_{live} is reasonably large for the problem at hand (see Section 2.2 for more details), hence possibly repeat the computation with a higher number.

2. If the problem persists, we will have to operate on the volume of the ellipsoids, which is likely to be too large as we said in the first part of this section. According to this, the general rule is that we should apply some reduction of the enlargement fraction of the ellipsoids (see also Section 2.3. This will allow the sampling process to become more efficient, hence increase the chance of finding a new live point.

3. If the first two attempts do not solve the error then it will be proper to inspect our choice of the prior PDFs for all the free parameters. This means that if we are unsure about the priors it will be useful to plot the evolution of the free parameters by means of the information contained in the files `parameter<parameter#>.txt`, as already discussed in detail in Section 3.1. In this way we can understand whether the sampling is reaching the limit imposed by the prior boundaries (meaning that the nested sampling wants to explore regions outside the selected parameter space) or it is lying well within the range that we investigated. We thus extend or shift the prior boundaries in order to accommodate for the trend shown by the sampling, if any, and we repeat the computation.

In practice, most of the times we deal with an application that we already managed to configure by finding the proper number of live points and an adequate choice of the prior PDFs. This means that we will often troubleshoot making use of the attempt #2 of the list. Unfortunately there is no specific rule on the extent f_0 should be reduced as it strictly depends

[8] Suppose that at a given nested iteration we have p different ellipsoids available, each one with a hyper-volume V_i, with $i = 1, 2 \ldots, p$. The drawing is done from one ellipsoid only, which is selected with a probability

$$p_k = \frac{V_k}{V_1 + V_2 + \cdots + V_p},$$

meaning that the larger is the hyper-volume V_k the higher is the probability that the k-th ellipsoid is selected. This is done because in general larger volumes correspond to regions of parameter space that require more sampling to be obtained.

on the application, the dimensionality of the inference, and the initial value that was used. By following the recipes provided in Sections 2.3 and 2.5 to set up an initial value, one could try readjusting (hence reducing in this case) f_0 with steps of 0.1 or 0.2 each time until reliable results are produced, following what discussed in Section 4. In relation to this, it is important to note that this sampling failure could pop up at different stages of the computation. We can identify three main ones:

1. The error appears during the first iterations, typically right after the iteration number N_{init}.

2. The error appears around half way toward the end of the process (at least several times N_{init}).

3. The error appears when approaching the termination condition of the nested sampling where the sampling slows down, namely for iterations close or very close to the end of the process.

The error will be more severe if it is happening at an earlier phase of the computation. In particular, in the first case it is likely that f_0 is several times off from its optimal value[9]. Another possibility is that the number of live points is wrong (too small) and it is advised to inspect the sampling of each free parameter to discover what is going on (see also the examples shown in Figure 3.1). In the second case, a good chance is that f_0 is off from its optimal value by a few to several steps and it is advised to act by directly reducing it, without intervening on any other configuring parameter. In the third case instead, f_0 might still be a good value for most of the computation but α could be wrong this time, since the ellipsoids tend to be too large only in the ending part of the process. We have to remember that the enlargement of the ellipsoids is dynamic, as shown by Equation 2.1, and it becomes smaller by the time more nested iterations are completed. It is then appropriate to first try changing the shrinking rate only, by increasing its value to allow the ellipsoids shrinking faster and producing smaller enlargements in the final stages of the computation (see also Section 2.4). If by increasing α by 0.02 or 0.03 will not solve the problem, then an additional decrease of f_0 will be needed, with careful steps of 0.1 each time. Since we experienced that intervening on f_0 and α is what happens more frequently, the error discussed in this section is more likely to pop up when the computation is approaching the termination condition.

To conclude this section, we highlight that the computational failure presented is often followed by an assertion failure, as in the example below

```
Can't find point with a better Likelihood.
Stopping the nested sampling loop prematurely.
-------------------------------------------------
 Final log(E): -9689.26 +/- 0.069862
-------------------------------------------------
 Total Computational Time: 4 seconds
-------------------------------------------------
Assertion failed: (index >= 0 && index < size()), function operator(), file
/localPath/Diamonds/include/Eigen/src/Core/DenseCoeffsBase.h, line 394.
```

[9] In an extreme case, the error discussed in this section could appear even before the first nested iteration is completed. This means that the value adopted for f_0 is really too large. In this particular situation one should reduce f_0 of a factor 10 and check whether the error disappears. Eventually, f_0 will have to be readjusted to tune its value to the optimal condition by using steps of 0.1 or 0.2.

This is because after the error message is executed, the `NestedSampler` class exits from the nested sampling loop and the code starts immediately with the computation of the results using the `Results` class. This is done to avoid losing the sampling information obtained up to the nested iteration where the process has stopped. The `Results` class will generate all the output files that contain the available sampled values (see Section 1.7 for more details), which we can use for inspection according to what described in Section 3.1. When no better likelihood points are found, it is likely that the sampling of the parameter space is wrong and therefore also the parameter estimation and/or the computation of the MPDs done by the member function `writeParametersSummaryToFile()` is subject to fail, but not necessarily all the times.

3.4 Ellipsoid matrix decomposition failed

This is the least common failure error caused by the sampler but at the same time it is the least intuitive and most troublesome. It can be identified by the following error message appearing on the terminal screen:

```
Ellipsoid Matrix decomposition failed.
Quitting program.
```

It is caused by the `Ellipsoid` class used by the `MultiEllipsoidSampler` class, and it is intimately related to the covariance matrices of the clusters of live points. To understand why this error is happening, consider for instance that at a given nested iteration the clustering algorithm identifies a cluster of live points that is very elongated in one dimension (highly prolate shape), as shown in Figure 3.3. Eventually the associated ellipsoid matrix may become a *ill-conditioned matrix*, meaning that it cannot be decomposed in an eigenvalue problem due to the presence of matrix elements that differ numerically among each other by many orders of magnitude[10]. This stems from the high proximity (or conversely the high dispersion) of the live points along one (or more) principle axis in comparison to the others, e.g. because the variance of the live points computed along one direction is very small compared to that along a direction orthogonal to it. If this happening, the process must be terminated because the entire ellipsoidal sampling method is not anymore reliable.

In practice this failure is typically generated by a wrong choice of the prior PDFs (especially if uniform priors) coupled with the presence of a high positive gradient in likelihood (steep increasing variation in the likelihood values). What could happen is that a maximum of the likelihood that is falling off the boundaries of the prior distribution may yield a sampling of the parameter space that is highly thickened along the edge of the allowed prior range toward the direction imposed by the positive gradient of the likelihood maximum. This produces very elongated shapes of sampling points, which as already discussed earlier, can in turn give rise to a ill-posed eigenvalues problem. In this case one has to readjust the prior boundaries with care, in order to make them more reliable. Most of the times when using uniform priors the real problem if such an error appears is that we have adopted parameter intervals that are either too small or lying off from the values imposed by the likelihood distribution (see also the discussion in Section 3.1 for more details). In addition, this error type is more likely to

[10]This error is occurring in the process of evaluating the overlapping regions between pairs of ellipsoids. The criterion to find that a live point is lying in this region is to check whether it is contained in both the ellipsoids. Numerically this is accomplished by constructing an ellipsoid matrix from the covariance matrices of the two ellipsoids involved in the overlap, according to the algorithm presented by [1]. If the covariance matrices are already ill-conditioned because of e.g. highly prolate shapes, they could cause the eigenvalues decomposition of the ellipsoid matrix to fail.

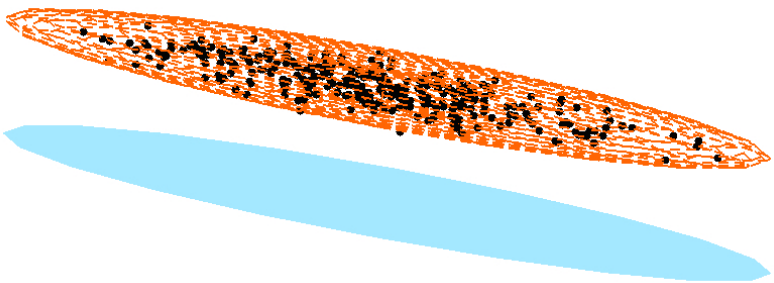


Figure 3.3: An example of a cluster of live points in three dimensions, with an associated ellipsoid very elongated in one dimension, causing a prolate shape that in an extreme case could lead to an ellipsoid matrix decomposition error.

be produced when highly informative datasets are being used, which cause the maxima of the likelihood distribution to become narrower and steeper in gradient.

More rarely the error could also appear in situations where the enlargement fraction of the ellipsoids is either too small or too large, even by several times, but always in the presence of highly informative datasets that produce very steep gradients in likelihood. This time one has to operate on the initial enlargement fraction by increasing (or decreasing) its value by many steps. Jointly with f_0 one could also attempt reducing (or increasing accordingly) the shrinking rate α since this will make the ellipsoids becoming less small (or big) toward the end of the process. The general experience in this occurrence is however suggesting that this error usually pops up when the enlargement fraction is too small.

Finally, similarly to what shown in Section 3.3, we recall that this error could show up followed by an assertion failure, as in the following example:

```
Ellipsoid Matrix decomposition failed.
Quitting program.
-----------------------------------------------
 Final log(E): 3.65051e+06 +/- 0.276146
-----------------------------------------------
 Total Computational Time: 1.23 hours
-----------------------------------------------
Assertion failed: (((SizeAtCompileTime == Dynamic && (MaxSizeAtCompileTime==Dynamic ||
size<=MaxSizeAtCompileTime)) || SizeAtCompileTime == size) && size>=0), function resize,
file /localPath/Diamonds/include/Eigen/src/Core/PlainObjectBase.h, line 262.
```

meaning that once again it was not possible to compute either the parameter estimators or the MPDs, or both of them. However, in any case all the output files containing the original sampling done up to the iteration in which the process was terminated, are stored and retrievable for inspection since the `Results` class is executed after the nested sampling has stopped.

4 Checking the results and understanding their reliability

In Section 3 we have shown that when using a Monte Carlo method there is no universal set up to ensure the algorithm successfully converges to the final results for any possible application. At this point we can present the general methodology for inspecting the outcomes and assess their reliability. This process is at the base of a correct use of DIAMONDS and we explicitly recommend a careful reading of all the upcoming sections, especially if you plan to use the results for scientific purposes.

To calibrate the code for a specific application and understand how the individual configuring parameters contribute to the different aspects of its functioning we need to rely on testing and well known examples used as benchmarks. By exploiting the applications done by [7] (including the demos) and later on by [9, 8], we could extrapolate sensible characteristics that can be applied in a more general way and that we explain here to help the user addressing a wider variety of inference problems. Therefore from now on we will consider that the nested sampling could reach the imposed termination condition without any failure error, which would otherwise cause the computation to stop prematurely and not producing the final results that we need. This enables DIAMONDS creating all the desired output files, including the `parameterSummary.txt` and all the files containing the marginal distributions (see Section 1.7). These information constitute the primary material of this last part of the guide that is focused on the results. Since an ellipsoidal sampler requires a proper configuration for each application, it is important and conscientious to understand whether we have done a good job or not.

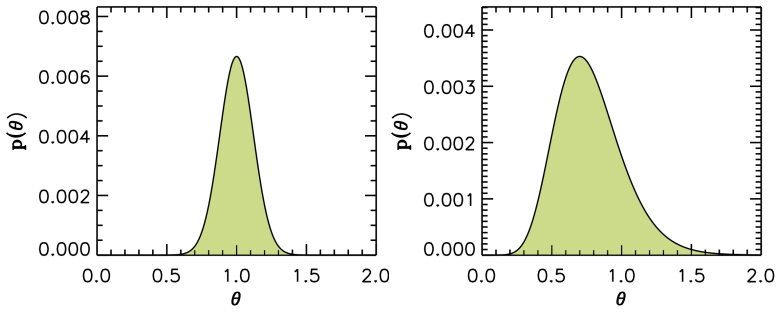


Figure 4.1: Marginal probability distributions with an optimal configuration of the code. In the left panel an example of symmetric Gaussian MPD, while in the right panel an asymmetric Gaussian MPD.

To set up a detailed discussion and present the different problems arising in understanding the reliability of the results, we take by approximation all the expected MPDs to have

a Gaussian-like shape[1]. This type of shape represents a very common outcome even when adopting different likelihood functions as those provided in the code package[2], thus opening up the possibility to apply this methodology to a large class of applications. Two examples of an optimal MPD for a free parameter θ are shown in Figure 4.1, with a characteristic bell-like shape that is evident even in the asymmetric case. In studies where we are aware that the final likelihood distribution is totally different from that of a Gaussian[3], what follows will not be possible to be accomplished. However, the general sense is to manipulate the configuring parameters of DIAMONDS in such a way that we reproduce (sample) the shape of the likelihood distribution in the best way that is possible.

4.1 How f_0 and α can change the MPD

In Chapter 2 we have seen a detailed overview of the different configuring parameters of the code, especially the initial enlargement fraction f_0 and the shrinking rate α. We have understood that by reducing or enlarging f_0 and/or α we can adjust the dynamical enlargement of the ellipsoids and its evolution within the nested sampling according to Equation 2.1. The parameters f_0 and α are certainly the most important ones to set up in the current version of DIAMONDS based on the multi ellipsoidal sampling. Throughout this manual we have mentioned several times that by directly acting on these two parameters we are able to tune the sampling efficiency and possibly solve some of the main computational failures shown in Chapter 3. In particular in Section 3.1 (see Figure 3.1) we have described that the sampling evolution and its dispersion along the prior range for the given free parameter can be controlled by f_0 and α in order to prevent an assertion failure when the resulting MPDs are larger than the extent of the prior interval[4].

But what if the computation terminates without failures and one or more of the MPDs that we obtain do not show this typical bell-like structure? We shall start by investigating the three main wrong outcomes that can be encountered, plotted in Figure 4.2. All these three MPDs correspond to a set of configuring parameters of DIAMONDS that is however not very far from that of the optimal values, and this is important to remember for improving the result.

The first example on the left panel of Figure 4.2 represents a Lorentzian, characterized by long (or fat) tails and a narrow central part. This distribution shape can appear quite commonly and arises when the enlargement fraction of the ellipsoids is smaller than the optimal value, yielding a bad sampling in regions of parameter space surrounding the given likelihood maximum, hence a lack of sampling points around it. Despite the fact the MPD may be lying in the middle of the parameter range (as in the example), this is *not* a good result because it is affecting the estimation of the Bayesian credible intervals, which will appear smaller than they are in reality. This is happening because the central region of the distribution is narrow, hence

[1] This means that the shape originates from a *multivariate normal distribution*, namely the MPD can be either a symmetric Gaussian or an asymmetric one due to the presence of correlations among the different free parameters. In practice, the entire concept of the ellipsoidal sampler arises from assuming that the sampling live points could distributed by following a mixture of multivariate normal distributions. This is however not limiting the applications of the code for likelihood functions that do not follow such statistics, as clearly demonstrated by the demos shown by [7] and provided in the code package.

[2] The likelihood defined in the classes `NormalLikelihood`, `MeanNormalLikelihood` and `ExponentialLikelihood`, are those provided in the basic package of the code, where the latter one in particular is specific for all the inferences involving Fourier transform datasets

[3] This is the case of multimodal distributions such as the eggbox and Rastrigin functions, and the one obtained by [7], or peculiar shapes with pronounced curving degeneracies like the Gaussian shell cylinders or the Rosenbrock function, see the demos presented by [3, 4, 7]

[4] We recall that this would be the case in which we are already confident about our choice of the prior PDFs and we do not need to change them.

providing larger probabilities for smaller intervals of θ as compared to the case of a Gaussian MPD, the latter having a wider central region instead. To solve the problem the first attempt to do is to always try enlarging f_0, using steps of 0.1 each time. In some cases, this may not be enough, meaning that the shrinking rate we are using is still too strong and has to be reduced by steps of 0.01 each time[5]. The optimal values of f_0 and α can thus be found by tuning them in the suggested direction until we recover MPDs similar to those presented in Figure 4.1. This applies to all the free parameters in exam, because if we have even a single one that shows such a MPD, then the global enlargement fraction is still too small and needs to be increased. Nonetheless, we have to keep in mind that exceeding with the increase of f_0 and/or the decrease of α may quickly lead to the computational failures discussed in Sections 3.1 and 3.3. If this happens, one has to act the other way around, as already explained in those sections.

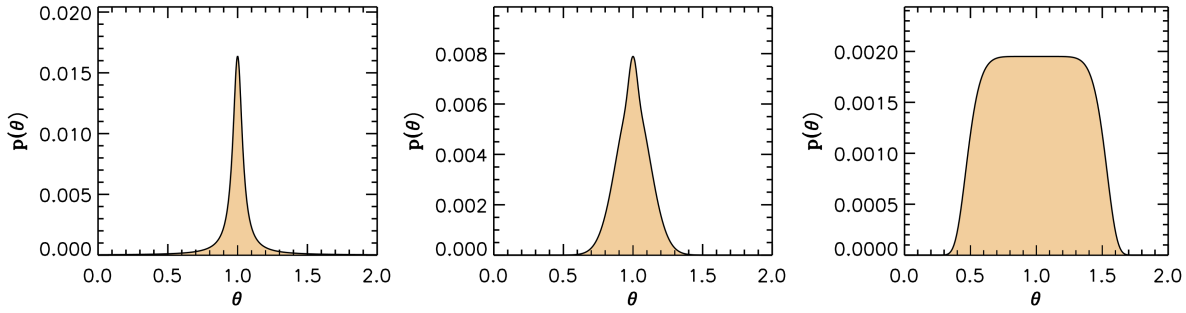


Figure 4.2: Marginal probability distributions resulting from a non-optimal configuration of the code. In the left panel the example of using too small ellipsoids, with a typical Lorentzian MPD. In the middle panel, the case of a quasi-Gaussian distribution with a spiky mode, typical of shrinking rates stronger than the optimal value. In the right panel the example of using too large ellipsoids, with a corresponding super Gaussian MPD, characterized by a plateau in the middle.

The second distribution (middle panel of Figure 4.2) has the shape similar to a Gaussian (this time not anymore fat tails as for the Lorentzian) but with a spiky central region, where the mode of the distribution is located. This pyramidal-like MPD is the closest possible to the optimal one and is obtained it will not significantly affect any of the results, but for the sake of using the code in the best way possible, especially in the light of computing an accurate Bayesian evidence, it should be corrected at least to some degree. The pyramidal shape arises because the sampling ellipsoids become very small too rapidly in the late stage of the process, narrowing down the prior volume, hence the parameter range used to sample new points, more than it should actually be done. To correct for this we usually need to act on the shrinking rate α, taking care of reducing its value by 0.01 only once (or twice at most), then repeating the computation. Exceeding with the reduction of α could prevent us from completing the entire computation, yielding the typical error described in Section 3.3, caused by a drastic fall in the sampling efficiency during the final phase of the nested sampling.

The third case to take into account is represented by a super Gaussian MPD, shown in the right panel of Figure 4.2. This shape is less common than the other two but if present can seriously hamper the final results, especially the Bayesian credible intervals that will result larger than they would be in the optimal case, and the mode of the distribution that this time is not even well defined. This distribution could appear when we use a total enlargement fraction that is larger than the optimal value but still not enough large to generate an assertion failure

[5]In practice it could happen sometimes that more than one pair of values for (f_0, α) yields the same optimal result. If this is the case, one can stick to any of the possible pairs being tested.

(see Section 3.1). This could arise in situations where we either use a value of f_0 that is larger than the optimal value or a shrinking rate that is too small, or a combination of the two. What happens is that we sample a rather large prior volume even toward the end of the process, meaning that the size of the ellipsoids we are using is too big with respect to the initial stage of the sampling to properly approximate the region containing the likelihood maximum. As a consequence, also the runtime of the code will considerably increase, especially in the ending phase of the computation. The best way to proceed is to directly act once again on f_0, this time by reducing its value with steps of 0.1 each time. If the problem persists one ought to change α by increasing it with steps of 0.01 each time. If the reduction of the total enlargement fraction is too strong, we will end up in a Lorentzian shape again, or a pyramidal MPD if we are even closer to the optimal solution.

We conclude this section with a few useful comments. The computations done by [7, 9, 8] showed that:

- f_0 has the strongest impact on the shape of the MPDs, while α usually remains unchanged for a given application, even if the dimensionality of the inference changes (there is no dependence of the shrinking rate on the number of dimensions, as discussed in Section 2.4).

- Sometimes changing α is not changing the result. This means that the result is quite stable because the likelihood maximum we are sampling is enough well defined (strong data condition, e.g. see [16]) to prevent us from performing a bad sampling even if the dynamical enlargement is not optimal.

- As we have demonstrated in Section 2.3, it is essential to consider the effect of the dimensionality of the problem in exam. For instance, suppose we want to perform a model comparison by using the same dataset and two different models with a different number of free parameters. Clearly, we need to perform a Bayesian inference with both models and retrieve the final total evidence cumulated by the nested sampling in each one of the two cases. Suppose that by starting with the first model, we have found the optimal configuration of DIAMONDS to run its related Bayesian inference. To perform the computation for the second model we shall readjust f_0 according to its new dimensionality, following the recipes provided in Section 2.3 and possibly in Section 2.5. It will be crucial to always inspect the final MPDs in order to understand whether they are regular or not, according to what explained before. Eventually we can tune f_0 and α again and repeat the computation until good results are produced (see also Section 4.5 for more discussion). If this is not done, at least one of the estimated Bayesian evidences will be wrong, implying that the whole model comparison cannot be reliable!

4.2 False multimodal MPD

Another problem that we need to present is related to MPDs that show a multi-modality where there is no real multi-modal solution[6]. Two examples of this situation are depicted in Figure 4.3, where the left plot shows a MPD with three modes and the right plot one with two modes. The reason why the multimodality of these MPDs is false is because it could be artificially produced with an erroneous configuration of the ellipsoidal sampler. What happens in practice is that if the enlargement fraction of the ellipsoids is smaller than the optimal

[6] A multi-modal solution is a solution in which multiple modes of the likelihood distribution are found. In this case it is not possible to conclude the inference process by providing a unique outcome of the corresponding free parameter.

value and by chance more than one cluster (at least two) of live points is identified during the nested sampling process, this could give rise to a sampling evolution similar to that shown in Figure 4.4, an evidence that the sampling is occurring in parallel in multiple regions of the parameter space for a significant portion of the total number of nested iterations. These regions could in principle correspond to either real multiple modes of the likelihood distribution that were correctly identified, or just small portions of a broader likelihood mode that was poorly sampled. Such a multiple sampling, if prolonged enough, could create multiple modes in the corresponding MPD, with these modes occurring at the same positions in θ where the sampling was obtained. To understand whether it is a real effect or not, we need to increase the initial enlargement fraction f_0 by steps of 0.1 each time, and see if the MPD changes, meaning that could recover the optimal case of a Gaussian MPD or one distribution close to it. In most of real applications there is no particular reason to believe that such multi-modality is real, and an investigation of the result is always recommended.

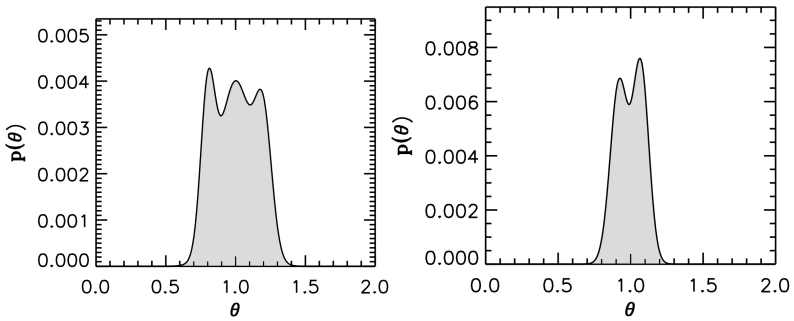


Figure 4.3: Example of marginal probability distributions with a false multi-modality. They could be caused by small ellipsoids coupled with the presence of multiple clusters, which in the plots from left to right are at least 3 and 2, respectively.

Sometimes, however, we expect a priori a clear multimodality of the likelihood distribution because of the particular model and dataset adopted. This could be for instance the case of the multi-modal model approach shown by [7] for the peak bagging analysis, or any application in which the datasets produce multi-modal distributions like the eggbox or the Rastrigin functions[7]. Conversely to what just discussed, in all genuine multi-modal likelihood problems we do not intervene on changing the size of the ellipsoids if a multi-modal MPD is obtained, since multi-modality is what we want this time[8].

4.3 False spike-like MPD

The non-optimal marginal distributions described so far are the most common ones that we can encounter and they are the closest cases to the optimal MPDs shown in Figure 4.1. However, there could be another situation that requires some explanation, although it is quite rare. Since DIAMONDS is a code based on a Monte Carlo method, it may happen due to its stochasticity

[7] An example of a real application involving a likelihood distribution similar to an eggbox or a Rastrigin function is that of the Scanning Electron Microscopy, in which scanning a material at atom resolution gives rise to many local maxima regularly distributed.

[8] It is important to bear in mind that for multi-modal solutions the Bayesian estimators such as the mode, the mean and the median become meaningless (they can only be defined for a uni-modal MPD). Even if the MPD is computed properly for this case, the evaluation of the estimators could easily yield an assertion failure discussed in Section 3.1. For these applications we therefore recommend a different approach that is not involving the use of MPDs, as also discussed by [7].

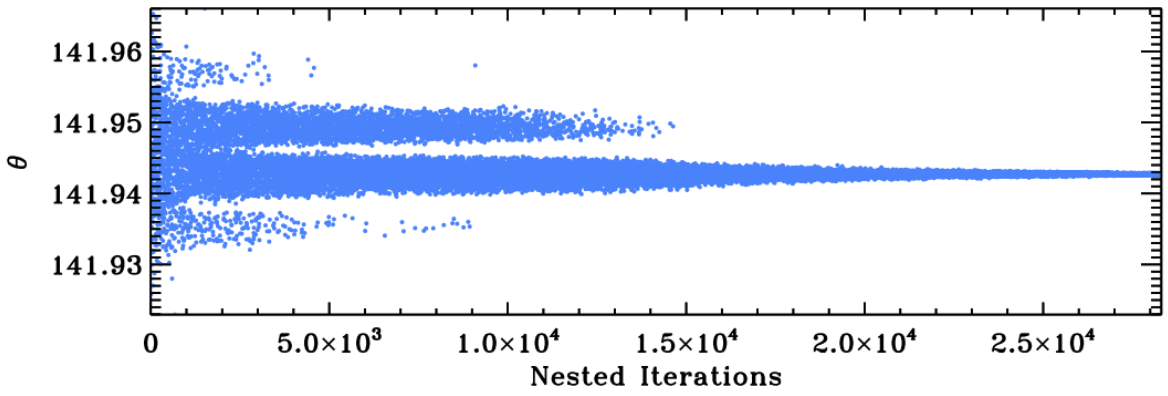


Figure 4.4: The sampling evolution of a free parameter showing the presence of multiple clusters separated from one another. The sampling in multiple regions stops around half way of the total number of nested iterations, hence converging into a single region containing the likelihood mode. This situation, if more enhanced, could potentially lead to an MPD with false multimodal features.

that the initial sampling of the prior volume is just completely wrong and despite the process evolves and it is able to reach the termination condition, the sampling will not be improved. We recall that a similar argument was already mentioned at the beginning of Chapter 3 in the event of computations that could stop prematurely just by accident. Therefore sometimes a wrong initial sampling of the live points could still yield final outcomes but that are totally wrong. This is the case in which the computation starts with a set of live points unluckily thickened in a small portion of the entire parameter space. What happens is that a single and very small ellipsoid is computed out of the identified cluster of live points, which then evolves by becoming even smaller due to Equation 2.1. This causes the nested sampling to reach the termination condition very quickly, and without computational failures, but produces MPDs that have a spike-like structure, as that shown in Figure 4.5.

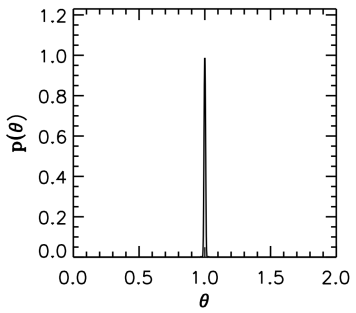


Figure 4.5: Example of an incorrect marginal probability distribution with an extremely narrow shape, similar to a spike or a sinc^2 function. This occurs when the initial sampling in the parameter space accidentally leads to a single and very small cluster of live points.

This spike-like structure will be similar for all the parameters inferred, and it does not mean that we have extremely precise parameter estimates or that we set too large prior boundaries for all of them, but it is instead a clear sign that we need to repeat the computation at least once more. If the problem persists, then we should proceed as follows: (i) try to increase the initial enlargement fraction by at least a few steps (e.g. 0.3, 0.4) until the MPDs appear fairly similar to one of the cases presented in Section 4.1; (ii) if by enlarging f_0 a failure error of the

type shown in Section 3.1 or 3.3 occurs, then it means that our choice of the prior PDFs was wrong from the very beginning and we need to inspect the evolution of the parameter sampling to understand in which way we have to correct the intervals, similarly to what explained in Section 3.1.

4.4 Truncated MPD and the role of uniform priors

The last example that we need to discuss is not caused by a problem in configuration of the ellipsoidal sampler but it is directly stemming from a wrong choice of the prior PDFs in the case of uniform distributions. This concerns MPDs that fall at the edge of our uniform prior range, and that could be truncated on one side of the interval (or both in the worst cases). Figure 4.6 depicts a MPD that is truncated on the left side due to a wrong choice of the left range limit for the free parameter θ. In the example, the MPD was still computed because luckily the Bayesian estimators could fall within the interval considered. If more pronounced, this situation would cause an assertion failure error, as seen in Section 3.1 and the related Figure 3.1. Such a MPD prevents us from considering the corresponding Bayesian estimates as reliable and, unless we are obliged to maintain our priors for specific reasons, we recommend to relaunch the code with a new set of priors that allows for more room toward the side of the truncated edge. This let us verify that we are able to correctly sample the mode of the likelihood distribution. If the priors cannot be changed and the MPD rises too close to the edge of the parameter interval, by following the same arguments provided in Section 3.1 we suggest to try converting the scale of the free parameter into a logarithmic one.

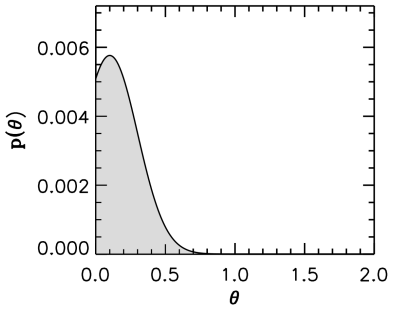


Figure 4.6: Example of a truncated marginal probability distribution, caused by a bad choice of the left prior boundary in a uniform prior PDF. This shows that the sampling algorithm has the tendency to sample new points very close to the edge of the prior range, meaning that the maximum of the likelihood distribution is likely to lie outside the given interval.

4.5 How to extend the methodology to multiple analyses

In this user guide we have often talked about situations where we need to apply a specific analysis to multiple datasets showing similar properties. What shown in Chapter 2 and in Sections 4.1, 4.2, 4.3, 4.4, provided us with a load of information on how to tackle the ellipsoidal sampler and obtain reliable results by using a general approach that can be applied to wide variety of applications. This paves the way to extend the methodology to two cases of interest:

- The analysis of multiple datasets with a single model and a fixed dimensionality
- The analysis of an individual dataset by means of mixture models with varying dimensionality

We conclude this manual by showing simple recipes on how to proceed in each of the two cases. For the first situation we have already mentioned the typical example of studying the background components in the Fourier domain of an ensemble of stars, where we use a single model for all of them. An example of this analysis was performed by [8]. What is most important in this context is the documentation provided in Section 2.5. Thus to perform this type of analysis on multiple datasets, we advise to adhere to the following steps:

1. Run the analysis on an initial dataset allowing a large range of clusters in the computation (e.g. $1 \leq N_{\rm clust} \leq 10$), allowing for a higher degree of flexibility in finding the optimal configuring parameters.

2. Once a reliable solution is found, according to what explained in Sections from 4.1 to 4.4, note down the optimal set of configuring parameters and how many clusters are being used on average by this first process. This last information can be obtained either from the Row #10 of the output ASCII file `configuringParameters.txt` or from the printed information on the terminal screen of the parameter `Ncl` at the last iterations of the nested sampling (see Section 1.7).

3. Restrict the range of clusters to this final value $\pm$ 1 or 2 clusters at most, or even fix it to the final value itself if you notice that $N_{\rm clust}$ remains very stable throughout the computation (meaning that `minNclusters`=`maxNclusters`).

4. Repeat the analysis for a few datasets and check whether the results remain reliable, according to what discussed in the previous sections.

5. Adopt the new range of clusters and the set of the other configuring parameters found so far for the remainder of the computations to be performed.

The second situation that we mentioned at the beginning of this section is probably less common and is well represented by the peak bagging analysis, namely the extraction of stellar oscillations from the Fourier spectrum of an individual star. This particular analysis usually involves Bayesian inferences with a relatively large number of dimensions (up to a few hundreds) and can alternatively be conducted by considering chunks of the same dataset, hence splitting the problem into separated subsets and performing the computation for a smaller number of oscillations per time (implying fewer dimensions). A clear real example of this approach is shown by [7] and [8]. The model used in this context is a so called *mixture model*, consisting of multiple terms stacked together, in this case a peak profile function for each oscillation peak to be fitted. What we need to apply to this case is essentially related to the effect of the number of dimensions on the initial enlargement fraction, which we discussed in detail in Sections 2.3 and 4.1. For this application we therefore suggest to follow the same steps #1, #2, #3 listed above, and then add the following ones:

4b. Compute the correction factor for the initial enlargement fraction f_0 according to Equation 2.9 by using the number of clusters identified in step #2 for your application (if the number of clusters is 4, no correction is needed).

5b. Check what is the dimensionality of the new analysis (e.g. depending on how many oscillation peaks you consider in a new chunk of the Fourier spectrum of the star), and note down the number.

6b. Evaluate the corresponding initial enlargement fraction f_0 for $N_{\rm clust} = 4$ as given by Equation 2.6, using the given number of dimensions (see Section 2.3 for more details)

and subsequently correct this enlargement by rescaling it according to the correction factor computed in the step #4b, if applicable.

This second type of application that we considered could also be that of a Bayesian model comparison, in which we use the same dataset but we want to compare different models, possibly with different numbers of free parameters involved. In particular, we refer to the peak significance test explained by [7], where two different mixture models are compared for every oscillation peak that needs to be tested (see also [8] for more details), using exactly the same dataset for both models.

For applying the methodologies discussed in this section, we assumed that prior PDFs were already available before running the inference for all the models and datasets used. Although the steps provided in this section are not ensured to perfectly work in all the cases, they certainly represent a good starting point to perform automated analyses on large samples of datasets.

Bibliography

[1] Alfano S., Greer M. L., 2003, J. Guid. Control Dyn., 26, 106

[2] Bonanno A. & Fröhlich H.-E., 2015, A&A, 580, 130

[3] Feroz F., Hobson M. P., 2008, MNRAS, 384, 449

[4] Feroz F., Hobson M. P., Bridges M., 2009, MNRAS, 398, 1601

[5] Fröhlich H.-E., Küker M., Hatzes A. P., & Strassmeier K. G., 2009, A&A, 506, 263

[6] Corsaro E., Froehlich H.-E., Bonanno A., et al. 2013, MNRAS, 430, 2313

[7] Corsaro E. & De Ridder J., 2014, A&A, 571, 71

[8] Corsaro E., De Ridder J., García R. A., 2015, A&A, 579, 83

[9] Corsaro E., De Ridder J., García R. A., 2015, A&A, 578, 76

[10] Kass R. E. & Wasserman L., 1996, J. Am. Stat. Assoc., 91, 1343

[11] Keeton, Charles R., 2011, MNRAS, 414, 1418

[12] Mukherjee P., Parkinson, D., Liddle Andrew R., 2006, ApJ, 638,51

[13] Shaw, J. R., Bridges M., Hobson M. P., 2004, MNRAS, 378, 1365

[14] Sivia, D. & Skilling, J. 2006, Data Analysis: A Bayesian Tutorial, Oxford OUP

[15] Skilling, J., 2004, AIP Conf. Proc., 735, 395

[16] Trotta, R., 2008, Con. Ph., 49, 71